# *Pratique de .NET 2.0 et de C♯ 2.0*

PATRICK SMACCHIA

# *Pratique de .NET 2.0 et de C♯ 2.0*

Éditions O'REILLY
18 rue Séguier
75006 PARIS
http://www.oreilly.fr

O'REILLY®

Cambridge • Cologne • Farnham • Paris • Pékin • Sébastopol • Taïpeï • Tokyo

*Couverture conçue et réalisée par* Hanna Dyer & Marcia Friedman.

*Édition :* Xavier Cazin.

Les programmes figurant dans ce livre ont pour but d'illustrer les sujets traités. Il n'est donné aucune garantie quant à leur fonctionnement une fois compilés, assemblés ou interprétés dans le cadre d'une utilisation professionnelle ou commerciale.


ISBN 2-84177-339-6

# *Table des matières*

# *Avant-propos*

## *À propos de ce livre*

La documentation officielle de *Microsoft* sur .NET est très vaste et décrit en détail chaque membre de chaque type du *framework* .NET. Elle contient aussi de nombreux articles concernant la description ou l'utilisation de telle ou telle partie de .NET. En tant que développeur, je sais combien l'utilisation de cette documentation est fondamentale lorsque l'on développe avec les technologies *Microsoft*. Cependant, de part le volume de cette documentation, il est assez difficile d'acquérir une vision globale des possibilités de .NET. En outre, d'après ma propre expérience, les nouvelles idées s'acquièrent mieux à partir d'un livre. Certes, on pourrait imprimer les dizaines de milliers de pages des **MSDN**, mais vous auriez du mal à les transporter pour les lire au calme, dans un jardin ou sur votre canapé.

Ce livre a donc été conçu dans l'optique d'être utilisé conjointement avec les **MSDN**. Il n'est pas question ici d'énumérer les dizaines de milliers de membres des milliers de types .NET mais plutôt d'expliquer et d'illustrer avec des exemples concrets les multiples facettes du langage C#, de la plateforme .NET ainsi que de son *framework*. J'espère qu'il répondra aux problématiques que vous rencontrerez et qu'il vous accompagnera hors des sentiers battus dans votre découverte de la technologie .NET.

## *L'organisation de ce livre*

### *Partie I : L'architecture de la plateforme .NET*

La première partie décrit l'architecture sous-jacente à la plateforme .NET. C'est dans cette partie que vous trouverez les réponses aux questions du type :

- Quels sont les liens entre l'exécution des applications .NET et le système d'exploitation sous-jacent ?
- Quelle est la structure des fichiers produits par la compilation de mon programme ?
- Comment sont gérées la sécurité et la synchronisation des accès aux ressources ?
- Comment puis-je tirer parti de tout cela pour améliorer la qualité et les performances de mes applications ?
- Comment puis-je exploiter du code déjà développé sous *Windows* à partir de mes applications .NET ?

## *Partie II : Le langage C♯ 2.0 et la comparaison C♯ 2.0/C++*

La deuxième partie décrit complètement le langage C♯ 2.0. Ce langage est beaucoup plus proche du langage Java que du langage C++. Je me suis donc efforcé de décrire les similitudes et les différences entre C♯ et C++ pour chaque facette du langage C♯. J'espère que cette approche répondra rapidement aux questions des développeurs C++ qui migrent vers C♯.

## *Partie III : Le framework .NET*

La troisième partie décrit les classes de base du *framework* .NET. Les fonctionnalités de ces classes se répartissent principalement dans les catégories suivantes :

- Les collections.
- Les classes de base classiques type calculs mathématiques, dates et durées, répertoires et fichiers, traces et débogage, expressions régulières, console.
- La gestion des entrées/sorties au moyen de flots de données.
- Le développement d'applications graphiques fenêtrées.
- La gestion des bases de données avec ADO.NET 2.0.
- La gestion des transactions.
- La création et l'exploitation de documents XML.
- Les applications à objets distribués avec .NET Remoting.
- Le développement d'application web avec ASP.NET 2.0.
- Les services web.

## *Remarques*

Ce plan de présentation de la technologie .NET permet d'avoir une vision globale. Néanmoins il est bien évident qu'une technologie aussi vaste comporte de multiples facettes qui transcendent ce découpage. Par exemple, nous avons choisi de placer la gestion de la synchronisation des accès aux ressources dans l'architecture de la plateforme .NET du fait qu'elle se base sur les notions sous-jacentes de threads et de processus. Cependant, en tant qu'ensemble de classes utilisables dans vos applications, la synchronisation fait aussi partie des classes de base du *framework* .NET. En outre le langage C♯ comporte des mots-clés spécialisés pour simplifier l'utilisation de ces classes. Enfin .NET Remoting présente des techniques qui permettent une approche évoluée de la synchronisation.

Cet ouvrage contient donc de nombreuses références internes qui j'espère, faciliteront vos recherches sur les différents sujets exposés.

# *À qui s'adresse ce livre et comment l'exploiter*

Ce livre s'adresse à vous dès lors que vous êtes intéressé par le développement sous .NET, que vous soyez étudiant, développeur professionnel ou amateur, enseignant ou formateur, architecte ou chef de projet.

Chaque chapitre a été conçu pour être lu d'une manière linéaire mais il n'en est pas de même pour l'ouvrage dans sa globalité. La première partie, l'architecture de la plateforme .NET, est

considérée comme la plus ardue mais aussi comme la plus fondamentale (et à mon sens comme la plus passionnante). Il n'est pas possible de développer correctement avec la technologie .NET sans tenir compte des services de la plateforme sous-jacente.

Le lecteur débutant pourra commencer par l'apprentissage du langage C# et des technologies de développement objet, tout en découvrant petit à petit les possibilités de .NET. .NET est un sujet très vaste, qui a plus que doublé en volume avec la version 2.0. Aussi, nous nous sommes efforcé de rester précis et concis.

Le lecteur expérimenté dans d'autres technologies devrait pouvoir bénéficier des explications concernant les nombreuses possibilités réellement innovantes offertes par .NET.

Le lecteur expérimenté avec .NET 1.x se servira de l'Annexe B qui référence toutes les nouveautés apportées par .NET 2.0 présentées dans le présent ouvrage.

# *Support*

Le présent ouvrage est supporté sur mon site : `http://www.smacchia.com`

Vous pouvez y télécharger les exemples de code fournis dans cet ouvrage. Nous croyons que bien souvent un exemple vaut mieux qu'un long discours. Le livre contient 648 exemples dont 523 listings C# et 65 pages ASP.NET 2.0.

Vous pouvez également nous adresser vos remarques sur cet ouvrage en écrivant aux Éditions *O'Reilly :*

*18 rue Séguier, 75006 Paris*

*Email : info@editions-oreilly.fr*

# *Remerciements*

Je souhaite avant tout remercier mon amie **Eli Ane** pour son soutien qui m'a tellement apporté durant la rédaction de cet ouvrage. J'ai aussi beaucoup apprécié le soutien de **Francis**, **France**, **Michel**, **Christine**, **Mathieu**, **Julien**, **Andrée**, **Patrick**, **Marie-Laure** et **Philippe**.

Merci à **Xavier Cazin** des Editions *O'Reilly* pour son aide et son professionnalisme.

Mes remerciements vont aussi aux relecteurs et amis qui m'ont apportés chacun de précieux conseils :

**Alain Metge**, 18 ans d'expérience. Responsable de la cellule architecture logicielle aux autoroutes du Sud de la France

**Dr. Bertrand Le Roy**, 8 ans d'expérience, participe depuis 3 ans à la conception et au développement de la technologie ASP.NET à *Microsoft Corporation*.

**Bruno Boucard**, 18 ans d'expérience, architecte/formateur à la Société Générale depuis 8 ans. Microsoft Informed Architect.

**Frédéric De Lène Mirouze** (alias Améthyste), spécialiste en développement web, 20 ans d'expérience, collabore notamment avec ELF, Glaxo, Nortel, Usinor. MCAD.NET.

**Jean-Baptiste Evain,** 3 ans d'expérience, spécialiste du Common Language Infrastructure, contributeur aux projets Mono et AspectDNG.

**Laurent Desmons,** 10 ans d'expérience, architecte et consultant .NET, collabore notamment avec Péchiney, Arcelor, Sollac. MCSD.NET.

**Matthieu Guyonnet-Duluc**, 4 ans d'expérience, développeur chez France Télécom dans le domaine commercial.

**Dr Michel Futtersack**, Maître de Conférences en Informatique, Université René Descartes, enseigne depuis 10 ans la conception et la programmation OO.

**Nicolas Frelat**, consultant .NET, 4 ans d'expérience. Early adopter de la plateforme .NET2.

**Olivier Girard**, 6 ans d'expérience, spécialiste en EAI, architecte à la Banque de France.

**Patrick Philippot,** freelance, 30 ans d'expérience (dont 19 à IBM), MVP .NET `www.mainsoft.fr`.

**Sami Jaber,** 8 ans d'expérience, consultant senior et formateur pour Valtech, collabore notamment avec Airbus, webmaster de `www.dotnetguru.org`.

**Sébastien Ros,** 7 ans d'expérience, spécialiste en mapping O/R, auteur de l'outil DTM, CTO d'Evaluant `www.evaluant.com`.

**Sébastien Vaucouleur**, 8 ans d'expérience, spécialiste en langages, collabore notamment avec Bull, Fujitsu, assistant de recherche à l'université ETH (Zurich).

**Thibaut Barrère,** freelance, spécialiste des plateformes J2EE/.NET/C++, programme depuis 1984 et a récemment collaboré avec Calyon, PPR/Redoute et MCS. Contributeur open-source sur les projets CruiseControl.Net, NAnt et TestDriven.Net

**Thomas Gil,** 8 ans d'expérience, spécialiste en Programmation Orientée Aspect, chef de projet de AspectDNG, consultant et formateur indépendant, co-webmaster de `www.dotnetguru.org`.

**Vincent Canestrier**, ancien enseignant au Conservatoire National des Arts et Métiers, ancien directeur technique de division Cap Gemini Ernst & Young.

Aussi, je souhaiterais remercier **Brian Grunkemeyer**, **Florin Lazar, Krzysztof Cwalina** et **Michael Marucheck** tous ingénieurs à *Microsoft Corp*, pour leurs réponses zélées à mes interrogations.

Enfin je souhaiterais vous remercier pour avoir choisi mon ouvrage. J'espère sincèrement qu'il vous aidera dans votre tâche.

# 1

# *Aborder la plateforme .NET*

## *Qu'est ce que .NET ?*

### *La technologie de développement logiciel de Microsoft*

Sous l'appellation .NET on désigne la plateforme de développement logiciel principale de *Microsoft*. Le sujet est très vaste et englobe aussi bien l'architecture interne, le format des composants, les langages de programmation, les classes standard et les outils. Sous l'appellation .NET, on désigne donc une nouvelle ère dans le monde *Microsoft*, qui supplante petit à petit l'ère COM/win32/C++/VB/ASP.

Le nom .NET a été choisi du fait que l'internet et plus généralement les réseaux sont de plus en plus exploités par les logiciels. Les applications sont de plus en plus interconnectées. De ce fait, la technologie .NET présente des facilités pour faire communiquer les applications qui font l'objet des chapitres 22 et 24. Pour faciliter l'interopérabilité des applications dans un milieu hétérogène, la plateforme .NET a aussi pour caractéristique forte d'exploiter XML à tous les niveaux.

Peu à peu, tous les produits *Microsoft* présentent partiellement ou complètement leurs APIs avec des types .NET. Par exemple, la version 2005 de *SQL Server* permet d'injecter du code .NET exécuté au sein du processus du SGBD qui réalise des traitements sur les données. L'API de programmation de *Windows Vista,* la prochaine version de *Windows*, est accessible sous forme de types .NET. La technologie de construction de pages web de .NET, nommée ASP.NET est maintenant privilégiée par le serveur web IIS 7.0. La suite *Office* présente un modèle de programmation basé sur .NET qui supplante peu à peu le modèle VBA.

### *Un ensemble de spécifications*

La plateforme .NET se base sur de nombreuses spécifications, certaines maintenues par d'autres organismes que *Microsoft*. Ces spécifications définissent des nouveaux langages, comme le lan-

gage C# et le langage IL ou des protocoles d'échange de données, comme le format SOAP. Plusieurs autres initiatives d'implémentations de ces spécifications sont en cours de réalisation. À terme, .NET devrait donc être disponible sur plusieurs systèmes d'exploitation et ne se cantonnera pas qu'au monde *Microsoft*. .NET est donc un nouvelle ère dans le monde du développement logiciel, comparable à l'ère C, l'ère C++ et à l'ère Java. Il est intéressant de remarquer que ce phénomène semble se reproduire périodiquement, approximativement tous les 7 ans. À chaque nouvelle ère, la productivité des développeurs augmente grâce à l'introduction de nouvelles idées et les applications sont plus conviviales et traitent plus de données, notamment grâce à la puissance croissante du *hardware*. La conséquence est que l'industrie adopte ces nouvelles technologies pour développer des logiciels de meilleure qualité tout en réduisant les coûts.

## *Présentation de la technologie .NET*

La technologie .NET se compose principalement de trois parties :

- Un ensemble extensible de langages de développement dont le langage **C#** et le langage **VB.NET**. Ces langages doivent respecter une spécification nommée **CLS** (*Common Langage Spécification*). Les types de base utilisés par ces langages doivent eux aussi respecter une spécification, nommée **CTS** (*Common Type System*).
- Un ensemble de classes de base utilisables à partir de programmes développés dans ces langages. On les désigne parfois sous le terme de ***BCL*** (*Base Class Library*). C'est ce que nous appellerons ***framework* .NET** tout au long de cet ouvrage.
- Une couche logicielle respectant une spécification nommée **CLI** (*Common Langage Infrastructure*). Elle est responsable de l'exécution des applications .NET. Cette couche logicielle ne connaît qu'un langage nommé langage **IL** (*Intermediate Langage*). Cette couche logicielle est notamment responsable durant l'exécution d'une application, de la compilation du code IL en langage machine. En conséquence les langages supportés par .NET doivent chacun disposer d'un compilateur permettant de produire du code IL. L'implémentation *Microsoft* de la spécification CLI est nommée **CLR** (*Common Langage Runtime*).

Parallèlement à ces trois parties, il existe de nombreux outils facilitant le développement d'applications avec .NET. On peut citer ***Visual Studio*** qui est un IDE (*Integrated Development Environment*, ou environnement de développement intégré en français) permettant de travailler notamment avec les langages C# VB.NET et C++/CLI. La liste de ces outils est disponible dans l'article **.NET Framework Tools** des **MSDN**. La plupart de ces outils sont décrits dans cet ouvrage et sont listés dans l'Annexe C.

Le découpage du présent ouvrage se base principalement sur ces trois parties :

# *Historique*

## *Le passé*

Dès 1998, l'équipe en charge du développement du produit *MTS* (*Microsoft Transaction Server*) souhaitait développer un nouveau produit pour pallier les problèmes de la technologie COM. Ces problèmes concernaient principalement le trop fort couplage entre COM et le système sous-jacent ainsi que la difficulté d'utilisation de cette technologie, notamment au niveau du déploiement et de la maintenance.

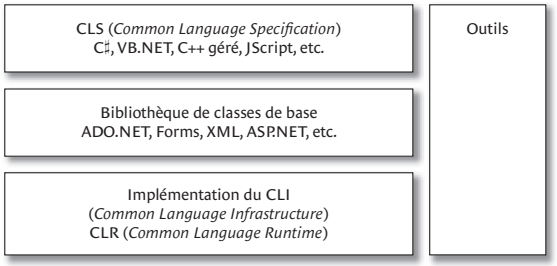


*Figure 1-1 : Vue générale de .NET*

Parallèlement, la communauté *Java* gagnait du terrain sur la scène du développement logiciel. De plus en plus d'entreprises étaient séduites par le concept de machine virtuelle permettant d'exécuter une application sur la plupart des systèmes sans efforts supplémentaires. De plus, les classes Java étaient bien plus faciles à utiliser que les MFC (*Microsoft Fundation Classes*) principalement grâce à l'absence de pointeurs et cela augmentait significativement la productivité des développeurs. Dès juin 2000 *Microsoft* annonça qu'il était en train de développer une nouvelle technologie qui comprenait notamment un nouveau langage, le langage C♯. Le 13 février 2002 était publiée la première version exploitable de .NET. Cet événement est décisif dans l'histoire de l'entreprise *Microsoft* et plus généralement, dans la scène du développement logiciel.

Parmi les ingénieurs en charge de ce projet, on peut citer *Anders Hejlsberg*, un des co-fondateurs de la société *Borland*. Cet ingénieur danois, concepteur du langage *Turbo Pascal* et du langage *Delphi*, fut débauché de *Borland* par *Microsoft* en 1996 pour travailler sur les *WFC* (*Windows Fundation Classes*), qui sont les classes utilisées par la machine virtuelle *Java* de *Microsoft*. Rapidement, il est placé dans l'équipe qui allait produire ce que l'on nomme aujourd'hui le CLR et le langage C♯.

En mars 2003, la version 1.1 de .NET est disponible. Elle contient notamment plus de classes sur le thème des fournisseurs de base de données (Oracle et ODBC), de la sécurité (cryptographie), de la technologie IP version6 et des technologies XML/XSLT. .NET 1.1 contient des outils pour développer des applications exécutables sous *Windows CE* (Pocket PC, smart phone...). La version 1.1 du *framework* .NET contient aussi le langage J#, destiné à aider les développeurs *Java* à migrer ver .NET.

## *Le présent*

Fin 2005, *Microsoft* publie la version 2.0 de .NET qui fait l'objet de cet ouvrage. Le nombre de types de base a plus que doublé couvrant maintenant de nombreux aspects initialement omis par les versions 1.x. Des améliorations, des évolutions et des optimisations sont apparues tant au niveau de la machine virtuelle en charge d'exécuter les applications .NET qu'au niveau des langages. Les outils de développement et notamment l'outil *Visual Studio* sont beaucoup plus complets et conviviaux. D'ailleurs, le sentiment général est que la qualité et l'intégration des outils est maintenant un élément prépondérant d'une plateforme de développement. La liste des nouveautés couvertes dans le présent ouvrage fait l'objet de l'Annexe B.

Parallèlement, on assiste au début de la concrétisation de deux méthodologies déjà bien intégrés dans les autres plateformes de développement : l'*eXtreme Programming* (ou *XP* à ne pas confondre avec *Windows XP*) et le développement à partir de modèles.

L'XP consiste à rationaliser les méthodologies utilisées pour développer un système d'information en coordonnant au mieux les activités de tous les acteurs impliqués. L'idée est de pouvoir faire face aux différentes évolutions et imprévus qui surviennent inexorablement dans le cahier des charges. Pour cela, on parle aussi parfois de méthode agile. L'agilité découle d'un certain nombre de contraintes. Il faut avant tout être à l'écoute du client en lui fournissant souvent et régulièrement une version testable. Il faut aussi faciliter la communication et le partage des connaissances entre les membres d'une équipe grâce à des outils polyvalents disponibles en plusieurs versions, chacune adaptée à une fonction précise. Le facteur humain est central dans l'XP. D'autres principes sont mis en œuvre tels que la conception d'une batterie de tests automatiques exécutée régulièrement afin de signaler au plus tôt les régressions dues aux évolutions. Cette batterie est en général exécutée après une compilation complète de l'application à partir des derniers sources. Le principe de *daily build* veut qu'une telle compilation soit effectuée quotidiennement, en général pendant la nuit. Toutes ces idées sont maintenant faciles à mettre en œuvre grâce aux nouvelles extensions *Team System* proposées par *Visual Studio 2005*.

Le développement à partir de modèles consiste à générer automatiquement le code source d'une application directement à partir d'un modèle. Ce modèle est exprimé en un langage de haut niveau, spécialement adapté aux besoins fonctionnels de l'application et donc très expressif. On parle de DSL (*Domain Specific Language*). L'avantage de cette approche est de permettre à l'équipe de travailler sur des sources proches des spécifications, réduisant ainsi les cycles d'évolution et la complexité du code. *Visual Studio 2005* propose des extensions spécialisées dans la conception et dans l'exploitation des DSLs. Ces extensions proposent aussi la visualisation de votre code source C# ou VB.NET sous la forme de diagrammes comparables à ceux fournis par UML.

## *Le futur*

*Microsoft* publiera *Windows Vista* durant l'année 2006. Cela marquera un tournant décisif pour la plateforme .NET puisque pour la première fois, l'environnement d'exécution .NET sera fourni de base avec un système d'exploitation. De nombreux nouveaux types .NET seront présentés par *Windows Vista* pour permettre d'accéder aux fonctionnalités de ce système d'exploitation directement à partir de votre code .NET. On peut citer notamment le nouveau *framework* de développement d'applications graphiques *WPF* (*Windows Presentation Framework*) et le nouveau *framework* de développement d'application distribuée *WCF* (*Windows Communication Framework*) présenté brièvement en page 1013.

Plus tard, en 2007 voire 2008, *Microsoft* publiera .NET 3.0 (nom de code *Orcas*). Cette version sera surtout axée sur une intégration poussée des technologies introduites par *Windows Vista* dans le *framework* et *Visual Studio*. Pour l'instant, seule l'équipe C# commence à faire part de ses travaux de recherches en ce qui concerne la version 3.0 du langage. Ils se focalisent sur un *framework* d'extension du langage et travaillent notamment sur une extension spécialisée pour la rédaction de requêtes sur un ensemble de données quelconques (objets, relationnelles ou XML). Des expressions lambda, qui sont un peu dans le même esprit que les méthodes anonymes de C# 2.0 mais en plus pratiques, viendront s'intégrer dans ces requêtes. D'autres nouveautés sont prévues telles que les types anonymes, le typage implicite des variables et des tableaux ainsi qu'une syntaxe efficace d'initialisation des objets et des tableaux.

Trois à quatre années plus tard, une version 4.0 de .NET sera publiée (nom de code *Hawaii*) mais aucune information n'est disponible pour l'instant.

# .NET hors du monde Microsoft / Windows

## L'organisation ECMA et .NET

En Octobre 2000 *Microsoft*, *Intel* et *Hewlett-Packard* ont proposé à l'*ECMA* (*European Computer Manufacturer's Association*) de standardiser un sous ensemble de .NET. Cette partie comprend principalement le langage C#, et le CLI. L'ECMA a accepté la demande et a créé une commission technique pour effectuer cette normalisation.

*Microsoft* n'est donc pas totalement propriétaire du langage C# et du CLI et le géant du logiciel a toléré jusqu'ici le fait que d'autres organisations implémentent ces spécifications. Pour plus d'information et pour obtenir les publications officielles de ces spécifications, vous pouvez consulter les URLs suivantes :

`http://www.ecma-international.org/`

`http://www.msdn.microsoft.com/net/ecma/`

`http://www.ecma-international.org/publications/standards/Ecma-334.htm` *(spécification C# 2.0)*

`http://www.ecma-international.org/publications/standards/Ecma-335.htm` *(spécification CLI)*

## Le consortium W3C

Le 9 mai 2000, *Microsoft* et 10 autres entreprises dont *IBM*, *Hewlett Packard*, *Compaq* et *SAP* ont proposé au consortium W3C de maintenir le standard SOAP. Le standard SOAP définit un format de message basé sur XML. Les services web peuvent communiquer au moyen de messages au format SOAP. L'idée de la standardisation de ce format est de rendre les services web complètement indépendants d'une entreprise ou d'une plateforme. Le format SOAP est décrit page 1000. Pour plus d'informations veuillez consulter la page `http://www.w3.org/TR/SOAP`.

Depuis, un certain nombre de spécifications visant à étendre les fonctionnalités des services web ont été soumis au W3C. Certaines sont en cours d'implémentation et certaines sont encore en cours de validation (voir page 989 et 1011).

## Le projet Mono

Le 9 juillet 2001, l'entreprise *Ximian*, fondée par *Miguel de Icaza*, a annoncé qu'elle développait une implémentation *open source* de .NET. La raison est que ses ingénieurs estiment que .NET représente la meilleure technologie de développement logiciel du moment. Le nom de ce projet est *Mono*.

À la mi 2003, l'entreprise *Novell* rachète *Ximian* récupérant ainsi Mono. Le 30 juin 2004, la version 1.0 du projet est publiée. Le projet Mono devrait être rapidement disponible en version .NET 2.0 et C# 2.0.

Le projet Mono comprend entre autres, un compilateur C# (distribué sous licence *GPL General Public Licence*), l'implémentation d'une bonne partie des bibliothèques .NET (distribuées sous licence *MIT/X11*) ainsi qu'une machine virtuelle qui implémente le CLI (distribuée sous licence *LGPL Lesser GPL*). Tout ceci est compilable sur *Windows*, sur *Linux* et sur plusieurs autres systèmes d'exploitation type *UNIX* tels que *Mac OS X*. La *home page* du projet est `http://www.mono-project.com`.

Malgré l'ombre que peut potentiellement faire le projet Mono à la version commerciale de .NET de *Microsoft*, ce dernier n'est pas forcément contre cette initiative. *John Montgomery*, responsable *Microsoft* de .NET a dit : « *...The fact that Ximian is doing this work is great. It's a validation of the work we've done, and it validates our standards activities. Also, it has caused a lot of eyeballs in the open source community to be directed to .Net, which we appreciate...* ». Le géant du logiciel n'est donc pas mécontent que le monde de l'*open source* ait une opportunité d'utiliser .NET. Cela représente autant de clients potentiels pour les produits développés autour de .NET.

## *Le projet SSCLI (Rotor) de Microsoft*

Le projet *Shared Source CLI* (aussi nommé *SSCLI* ou *Rotor*) de *Microsoft* consiste à distribuer du code source implémentant le CLI et certaines parties du *framework* .NET. Le projet SSCLI est surtout distribué à des fins académiques pour permettre aux étudiants et aux chercheurs de s'initier et de travailler sur les internes d'une machine virtuelle moderne (GC, JIT compilation etc). Vous pouvez néanmoins vous en servir pour comprendre le fonctionnement interne de .NET et pour déboguer vos applications. En revanche, vous n'avez pas le droit de vous en servir à des fins commerciales.

Concrètement, SSCLI c'est plusieurs millions de lignes de code, un compilateur C#, un compilateur JScript et de nombreux outils. Une version 2.0 devrait être disponible rapidement après la sortie de .NET 2.0. Les parties centrales du *framework* sont supportées telles que XML ou .NET Remoting. En revanche, d'autres domaines tout aussi importants mais plutôt orientés fonctionnel tels que ADO.NET, ASP.NET et Windows Form ne sont pas implémentés par SSCLI.

Contrairement à l'implémentation *Microsoft* commerciale de .NET, SSCLI peut fonctionner sur d'autres systèmes d'exploitation que *Windows*. À l'heure actuelle vous pouvez vous servir de SSCLI sur les systèmes d'exploitation *FreeBSD, Mac OS X* et *Windows*. Tout ceci est possible parce qu'en interne SSCLI n'appelle pas directement l'API win32. Le projet utilise une API proche de win32 nommée *PAL* (*Platform Abstraction Layer*). Or, cette API est supportée par ces trois systèmes d'exploitation.

La page officielle du projet est l'URL : `http://msdn.microsoft.com/net/sscli`

Vous pouvez consulter en ligne les fichiers sources à l'URL : `http://dotnet.di.unipi.it/SharedSourceCli.aspx`

# *Liens sur .NET*

Nous vous proposons des liens vers les principaux sites consacrés au développement sous .NET. Notez qu'avec le temps, je compte mettre de plus en plus de ressources sur mon site `http://www.smacchia.com` à propos du développement sous .NET.

## *Sites français*

Voici des sites qui en plus de contenir des articles parmi les plus intéressants disponible sur le web, sont entièrement en français :

`http://www.dotnetguru.org`

`http://www.microsoft.com/france/msdn`

`http://forums.microsoft.com/msdn/`

http://fr.gotdotnet.com

http://www.sharptoolbox.com/

http://www.dotnet-fr.org

http://www.c2i.fr

http://www.csharpfr.com

http://www.dotnet-news.com/

http://www.labo-dotnet.com/

http://www.techheadbrothers.com

http://www.blabladotnet.com

http://www.programmationworld.com

http://www.essisharp.ht.st

Précisons que le site *dotnetguru* (*DNG*) sort du rang grâce à ses nombreux articles de qualité et grâce aux entrées pertinentes des blogs de ses auteurs.

## *Newsgroup en français*

Sur le serveur msnews.microsoft.com :

microsoft.public.fr.dotnet

microsoft.public.fr.dotnet.adonet

microsoft.public.fr.dotnet.aspnet

microsoft.public.fr.dotnet.csharp

## *Sites anglais*

Voici d'autres sites en anglais :

http://www.msdn.microsoft.com

http://www.gotdotnet.com

http://msdn.microsoft.com/msdnmag/

http://www.theserverside.net

http://www.dotnet247.com

http://www.15seconds.com

http://www.codeproject.com/

http://www.eggheadcafe.com/

http://www.devx.com/

http://channel9.msdn.com/

http://dotnet.sys-con.com/

http://dotnet.oreilly.com http://www.ondotnet.com/

http://www.dotmugs.ch/

http://www.asp.net/

http://www.ondotnet.com

http://dotnetjunkies.com/

http://www.codeguru.com

http://www.c-sharpcorner.com http://www.csharp-corner.com

http://www.devhood.com

http://www.developer.com

http://www.4guysfromrolla.com (ASP.NET)

La section *download* du site http://www.idesign.net

## *Newsgroup en anglais*

Sur le serveur msnews.microsoft.com :

microsoft.public.dotnet.framework

microsoft.public.dotnet.framework.adonet

microsoft.public.dotnet.framework.aspnet

microsoft.public.dotnet.framework.clr

microsoft.public.dotnet.framework.performance

microsoft.public.dotnet.framework.remoting

microsoft.public.dotnet.framework.sdk

microsoft.public.dotnet.framework.webservices

microsoft.public.dotnet.general

microsoft.public.dotnet.languages.csharp

## *Blogs*

Enfin, voici quelques blogs à forte valeur ajoutée sur le développement logiciel orienté sur les technologies .NET :

BCL Team http://blogs.msdn.com/bclteam/

Bart De Smet (Divers) http://blogs.bartdesmet.net/bart/

Benjamin Mitchell (Web services, eXtreme Programming) http://benjaminm.net/

Brad Abrams (BCL, design) http://blogs.msdn.com/brada/default.aspx

Chris Brumme (CLR) http://blogs.msdn.com/cbrumme/

Chris Sells (Windows Form, divers) http://www.sellsbrothers.com/

Clemens Vaster (SOA, design, divers) http://staff.newtelligence.net/clemensv/

David M. Kean (FxCop, Windows Installer, divers) http://davidkean.net/

Dino Esposito (ASP.NET) http://weblogs.asp.net/despos/

Don Box (WCF, divers) http://www.pluralsight.com/blogs/dbox/default.aspx

Dotnetguru blog agrégé (Divers) http://www.dotnetguru.org/blogs/index.php

Eric Gunnerson (C#, divers) http://blogs.msdn.com/ericgu/

Frantz Bouma (Divers) http://weblogs.asp.net/fbouma

Fritz Onion (ASP.NET) http://pluralsight.com/blogs/fritz/default.aspx

Florin Lazar (Transactions) http://blogs.msdn.com/florinlazar/

Fredrik Normén (ASP.NET) http://fredrik.nsquared2.com/default.aspx

Jim Johnson (Transactions) http://pluralsight.com/blogs/jimjohn

Junfeng Zhang (GAC, versionning, CLR) http://blogs.msdn.com/junfeng/

Keith Brown (Sécurité) http://pluralsight.com/blogs/keith/default.aspx

Krzysztof Cwalina (Design) http://blogs.msdn.com/kcwalina/

Martin Fowler (Architecture, pattern) http://martinfowler.com/bliki/

Matt Pietrek (Win32, Windows) http://blogs.msdn.com/matt_pietrek/

Michele Leroux Bustamente (Divers) http://www.dasblonde.com/

Miguel de Icaza (Mono) http://tirania.org/blog/

Mike Stall (Debug) http://blogs.msdn.com/jmstall/

Mike Taulty (Divers) http://mtaulty.com/blog

Pluarlsight blog agrégé (Divers) http://pluralsight.com/blogs/default.aspx

Rico Mariani (Performances, GC) http://blogs.msdn.com/ricom/

Rockford Lhotka (Design, divers) http://www.lhotka.net/WeBlog/

Sahid Malik (ADO.NET, transactions) http://codebetter.com/blogs/sahil.malik/

Sam Gentile (Divers) http://samgentile.com/blog/

Scott Guthrie (ASP.NET) http://weblogs.asp.net/scottgu/

TheServerSide blog agrégé (Divers) http://www.theserverside.net/blogs/index.tss

Valery Pryamikov (Sécurité) http://www.harper.no/valery/

Wesner Moise (Divers, sujets pointus habituellement peu documentés) http://wesnerm.blogs.com/net_undocumented/

# 2

# *Assemblages, modules, langage IL*

Les *assemblages* sont les équivalents .NET des fichiers `.exe` et `.dll` de *Windows*. Ce sont donc les *composants* de la plateforme .NET.

## *Assemblages, modules et fichiers de ressource*

### *Assemblages et modules*

Un assemblage est une unité logique qui peut être définie sur plusieurs fichiers appelés *modules*. Tous les fichiers constituant un assemblage doivent se trouver dans le même répertoire.

On a tendance a considérer qu'un assemblage est une unité physique (i.e un assemblage = un fichier) car **la plupart des assemblages n'ont qu'un seul module**. Cela est en grande partie du au fait que l'environnement *Visual Studio* n'a pas la possibilité de générer des assemblage multi modules. Comme nous allons l'exposer, pour obtenir des assemblages multi modules, il faut faire l'effort de travailler avec des outils en ligne de commande tels que le compilateur C# `csc.exe` décrit page 317, ou l'outil `al.exe` décrit page 37).

Parmi les modules d'un assemblage, il y a un *module principal* qui joue un rôle particulier car :

- Tout assemblage en comporte un et un seul.
- En conséquence, un assemblage mono module se confond avec son module principal.
- Dans le cas d'un assemblage à plusieurs modules, c'est toujours ce module qui est chargé en premier par le CLR. Le module principal d'un assemblage multi modules référence les autres modules. Ainsi, l'utilisateur d'un assemblage n'a besoin de connaître que le module principal.

Le module principal est un fichier d'extension `.exe` ou `.dll` selon que son assemblage est un exécutable ou une bibliothèque de types. Un module qui n'est pas le module principal est un fichier d'extension `.netmodule`.

## Fichiers de ressource

En plus du code .NET compilé , un module peut physiquement contenir d'autres types de ressources telles que des images bitmap ou des documents XML. De telles ressources peuvent aussi être contenues dans leur fichier d'origine (par exemple d'extension `.jpg` ou `.xml`) et référencées par le module principal. Dans ce cas on dit que ces fichiers référencés sont des *fichiers de ressources* de l'assemblage. Dans ce chapitre, nous verrons que l'utilisation de ressources constitue une technique efficace pour globaliser une application.

## Assemblages, modules, types et ressources

La figure suivante utilise la notation UML pour résumer les relations entre assemblages, modules, types et ressources. On y voit qu'un même module peut être contenu dans plusieurs assemblages et qu'un même type peut être contenu dans plusieurs modules. Nous vous conseillons vivement d'éviter ces deux techniques de réutilisation de code et de préférer avoir recours à des assemblages bibliothèques de types.

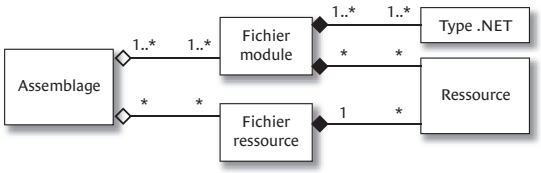


*Figure 2-1 : Diagramme UML : Assemblages, Modules, Ressources*

## Pourquoi morceler un assemblage en plusieurs modules/fichiers de ressources ?

On peut se demander quel est l'intérêt de morceler un assemblage en plusieurs modules. En effet, cette possibilité est rarement exploitée. Nous avons identifié les trois cas de figure suivants où cette fonctionnalité apporte un réel avantage :

- L'adage proféré il y a trois décennies qui disait qu'un logiciel passe 80% du temps dans 20% du code est toujours d'actualité. Si l'on isole la grosse partie du code peu utilisée dans des modules séparés, ces modules ne seront peut-être jamais utilisés tous ensembles. Donc la plupart du temps on économisera les ressources nécessaires au chargement total de l'assemblage dans le processus. Ces ressources sont la mémoire vive, les accès au disque dur, mais aussi la bande passante des réseaux si l'assemblage est stocké sur une machine distante.
- Un fichier de ressource ne sera réellement chargé que lorsque le programme en aura réellement besoin. Si une application tourne en français, on fait donc l'économie du chargement des fichiers de ressources en anglais.
- Si un même assemblage est développé par plusieurs développeurs, il se peut que certains préfèrent le langage VB.NET tandis que d'autres préfèrent C#. Dans ce cas chaque module peut être développé dans un langage différent.

### *L'outil ILMerge*

Sachez qu'à l'inverse vous pouvez réunir plusieurs assemblages au sein d'un même fichier d'extension .exe ou .dll. Pour cela, il vous faut avoir recours à l'outil ILMerge distribué gratuitement par *Microsoft* et téléchargeable à partir du web. Les fonctionnalités de cet outil sont aussi exploitables programmatiquement grâce à une API documentée. L'outil sait aussi tenir compte des assemblages signés.

## *Anatomie des modules*

### *Notion de fichier Portable Executable (PE)*

Un *fichier PE* (PE pour *Portable Executable*) est un fichier exécutable par *Windows*. Un fichier PE est en général d'extension `.exe` ou `.dll`. Les premiers octets d'un fichier PE constituent un en-tête formaté interprété par *Windows* au moment où l'exécutable est démarré. Ces octets contiennent des informations telles que le plus petit numéro de version de *Windows* sur lequel l'exécutable peut être utilisé ou le fait que l'exécutable est une application fenêtrée ou une application console.

Le formatage des fichiers PE est optimisé pour ne pas dégrader les performances. À quelques octets prés, l'image en mémoire d'un fichier PE qui est exploitée par *Windows* lors de l'exécution est identique au fichier PE. C'est pour cette raison que les fichiers PE sont parfois nommés *fichiers image*.

Les modules sont des fichiers PE car la plateforme .NET utilise les services de *Windows* en ce qui concerne le démarrage des applications. Nous expliquons en page comment le CLR est chargé par *Windows* lors du démarrage d'une application .NET.

Vous entendrez aussi parler de format *PE/COFF* (pour *Common Object File Format*). Le format COFF est utilisé par le compilateur C++ lorsqu'il lie les *fichiers objets*. L'extension COFF du format PE/COFF est ignorée par .NET.

### *Structure physique d'un module*

Chaque module comprend une section d'*entête CLR* qui contient la version du CLR pour laquelle l'assemblage a été compilée. Cette section contient éventuellement une référence vers le point d'entrée géré si le module en contient un.

Le module principal d'un assemblage contient une section appelée *manifeste* qui contient entre autres les références vers les autres modules et vers les fichiers de ressources. Les données du manifeste sont parfois nommées *métadonnées de l'assemblage*.

Chaque module contient une section qui décrit complètement son contenu (les types, les membres des types, les dépendances entre types, les ressources...). Les informations d'auto description contenues dans cette section sont nommées *métadonnées* (*metadata* en anglais).

Enfin, le module contient le code .NET compilé en IL ainsi que les ressources. Le schéma suivant représente la structure physique d'un module.

### *Structure du manifeste*

Le manifeste contient les informations d'auto description de l'assemblage. Il y a quatre types d'informations d'auto description et le manifeste contient une table pour chacun de ces types :

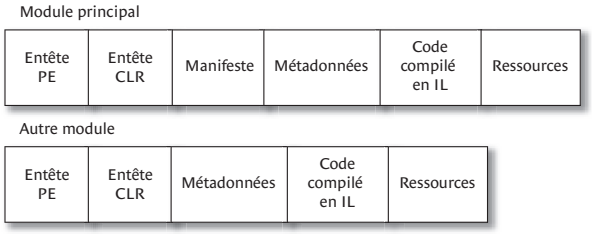


*Figure 2-2 : Structure physique d'un module*

- La table *AssemblyDef* : Cette table a une seule entrée qui contient :
  - Le nom de l'assemblage (sans extension, sans chemin).
  - La version de l'assemblage.
  - La culture de l'assemblage.
  - Des drapeaux décrivant certaines caractéristiques de l'assemblage.
  - Une référence vers un algorithme de hachage.
  - La clé publique de l'éditeur (qui peut être nulle éventuellement).
  - Toutes ces notions sont présentées dans les pages qui suivent.
- La table *FileDef* : Cette table contient une entrée pour chaque module et fichier de ressource de l'assemblage mis à part le module principal (donc si un assemblage n'a qu'un module, cette table est vide). Chaque entrée inclut le nom du fichier (avec l'extension), des drapeaux décrivant certaines caractéristiques du fichier et une *valeur de hachage* du fichier.
- La table *ManifestResourceDef* : Cette table contient une entrée pour chaque type et chaque ressource de l'assemblage. Chaque entrée contient un index vers la table *FileDef* pour indiquer dans quel fichier est le type ou la ressource. Dans le cas d'un type, l'entrée contient aussi un « offset » indiquant où se trouve physiquement le type dans le fichier. Une conséquence est que chaque compilation d'un module implique la reconstruction du manifeste, donc la recompilation du module principal.
- La table *ExportedTypeDef* : Cette table contient une entrée pour chaque type visible hors de l'assemblage. Chaque entrée contient le nom du type, un index vers la table *FileDef* et un « offset » indiquant où se trouve physiquement le type dans le fichier. Par souci d'économie de place, les types visibles hors de l'assemblage définis dans le module principal ne sont pas répétés dans cette table. En effet, nous allons voir que ces types sont déjà décrits dans la section métadonnées.

## *Structure de la section métadonnées de type*

Les métadonnées de type sont stockées dans des tables. Il existe trois sortes de tables de métadonnées de type : les tables de définitions, les tables de références et les tables de pointeurs.

### *Les tables de définition*

Chaque table de définition renferme des informations à propos d'une catégorie d'élément de module (i.e une table référençant les méthodes de toutes les classes, une table référençant toutes les classes etc). Nous ne présentons pas toutes ces tables mais voici les plus importantes :

- La table *ModuleDef* : Cette table contient une seule entrée qui définit le présent module. Cette entrée contient notamment le nom du fichier avec son extension mais sans son chemin.
- La table *TypeDef* : Cette table contient une entrée pour chaque type défini dans le module. Chaque entrée contient le nom du type, le type de base, les drapeaux du type (`public`, `internal`, `sealed` etc), et des index référençant les entrées des membres du type dans les tables *MethodDef*, *FieldDef*, *PropertyDef*, *EventDef*, ... (une entrée pour chaque membre).
- La table *MethodDef* : Cette table contient une entrée pour chaque méthode définie dans le module. Chaque entrée contient le nom de la méthode, les drapeaux de la méthode (`public`, `abstract`, `sealed` etc), l'offset permettant de situer physiquement dans le module le début du code IL de la méthode et une référence vers la signature de la méthode, qui est contenue dans un format binaire dans un tas appelé *#blob* décrit plus loin.

Il y a aussi une table pour les champs (*FieldDef*), une pour les propriétés (*PropertyDef*), une pour les événements (*EventDef*) etc. La définition de ces tables est standard et chaque table a un numéro codé sur un octet. Par exemple toutes les tables *MethodDef* de tous les modules .NET ont le numéro de table 6.

## *Les tables de références*

Les tables de références contiennent les informations sur les éléments référencés par le module. Les éléments référencés peuvent être définis dans d'autres modules du même assemblage ou définis dans d'autres assemblages. Voici quelques tables de références couramment utilisées :

- La table *AssemblyRef* : Cette table contient une entrée pour chaque assemblage référencé dans le module (i.e chaque assemblage qui contient au moins un élément référencé dans le module). Chaque entrée contient les quatre composantes du nom fort à savoir : le nom de l'assemblage (sans chemin ni extension), le numéro de version, la culture et le jeton de clé publique (éventuellement la valeur nulle si il n'y en a pas).
- La table *ModuleRef* : Cette table contient une entrée pour chaque module du même assemblage référencé dans le module (i.e chaque module qui contient au moins un élément référencé dans le module). Chaque entrée contient le nom du module avec son extension.
- La table *TypeRef* : Cette table contient une entrée pour chaque type référencé dans le module. Chaque entrée contient le nom du type et une référence vers là où il est défini. Si le type est défini dans ce module ou dans un autre module du même assemblage, la référence indique une entrée de la table *ModuleRef*. Si le type est défini dans un autre assemblage, la référence indique une entrée de la table *AssemblyRef*. Si le type est encapsulé dans un autre type, la référence indique une entrée de la table *TypeRef*.
- La table *MemberRef* : Cette table contient une entrée pour chaque membre référencé dans le module. Un membre peut être par exemple une méthode, un champ ou une propriété. Chaque entrée est constituée du nom du membre, de sa signature et d'une référence vers la table *TypeRef*.

La définition de ces tables est aussi standard et chaque table a un numéro codé sur un octet. Par exemple toutes les tables *MemberRef* de tous les modules .NET ont le numéro de table 10.

### *Les tables de pointeurs*

Les tables de pointeurs permettent à la compilation de référencer des éléments du code encore inconnus (un peu comme les déclaration en avant dans C++). En changeant l'ordre de définition des éléments de votre code, vous pouvez réduire le contenu de ces tables. On peut citer les tables *MethodPtr*, *ParamPtr* ou *FieldPtr*.

### *Les tas*

En plus de ces tables la section des métadonnées de type contient quatre tas (*heap*) nommés *#Strings*, *#Blob*, *#US* et *#GUID*.

- Le tas *#Strings* contient des chaînes de caractères comme le nom des méthodes. Ainsi les éléments des tables *MethodDef* ou *MemberRef* ne contiennent pas de chaînes de caractères mais référencent les éléments du tas *#string*.
- Le tas *#Blob* contient des informations binaires, comme la signature des méthodes stockée sous une forme binaire. Ainsi les éléments des tables *MethodDef* ou *MemberRef* ne contiennent pas les signatures des méthodes mais référencent des éléments du tas *#blob*.
- Le tas *#US* (pour *User String*) contient les chaînes de caractères définies directement au sein du code.
- Le tas *#GUID* contient les GUID définis et utilisés dans le programme. Un GUID est une constante de 16 octets, utilisée pour nommer une ressource. La particularité des GUID est que vous pouvez les générer à partir d'outils tels que `guidgen.exe`, de façon à être quasi-certain d'avoir fabriqué un GUID unique au monde. Les GUID sont particulièrement utilisés dans la technologie COM.

Les métadonnées de type sont très importantes dans l'architecture .NET. Elles sont référencées par les jetons de métadonnées du code IL qui font l'objet de la section page 46. Cette section présente un exemple qui souligne l'importance des tables et des tas de la section métadonnées de type. Les métadonnées de type sont aussi utilisées par la notion d'attributs .NET (décrite page 248) et le mécanisme de réflexion (décrit page 233).

### *Le tas #*

Certains documents se réfèrent parfois à un tas nommé # . Ce tas spécial contient en fait toutes les tables de métadonnées, y compris celles du manifeste s'il s'agit d'un module principal.

# *Analyse d'un assemblage avec ildasm.exe et Reflector*

## *Construction de l'assemblage à analyser*

Nous allons créer un assemblage avec :

- Un module principal `Foo1.exe`.
- Un module `Foo2.netmodule`.
- Un fichier de ressource `Image.jpg`.

Placez dans le même répertoire les deux fichiers sources C# suivant (`Foo1.cs` et `Foo2.cs`) ainsi qu'un fichier au format jpeg nommé `Image.jpg`.

*Exemple 2-1 :* *`Foo2.cs`*

```
namespace Foo {
   public class UneClasse {
      public override string ToString() {
         return "Bonjour de Foo2" ;
      }
   }
}
```

*Exemple 2-2 :* *`Foo1.cs`*

```
using System ;
using System.Reflection ;
[assembly: AssemblyCompany("L'entreprise")]
namespace Foo {
   class Program {
      public static void Main(string[] argv) {
         Console.WriteLine("Bonjour de Foo1") ;
         UneClasse a = new UneClasse() ;
         Console.WriteLine( a ) ;
      }
   }
}
```

Comme nous souhaitons construire un assemblage avec plus d'un module, nous n'avons pas d'autres choix que d'utiliser le compilateur `csc.exe` en ligne de commande (car l'environnement *Visual Studio* ne gère pas les assemblages multi modules). Le compilateur `csc.exe` est décrit en détails page 317.

Créer les fichiers `Foo2.netmodule` puis `Foo1.exe` en tapant dans l'ordre les lignes de commande suivantes (le compilateur `csc.exe` se trouve dans le répertoire <répertoire-d-installation-de-WINDOWS>\Microsoft.NET\Framework\v2.*) :

```
> csc.exe /target:module Foo2.cs
> csc.exe /Addmodule:Foo2.netmodule /LinkResource:Image.jpg  Foo1.cs
```

Lancez l'exécutable `Foo1.exe`, le programme affiche sur la console :

```
Bonjour de Foo1
Bonjour de Foo2
```

## *Analyse d'un module avec l'outil ildasm.exe*

On peut analyser le contenu d'un assemblage ou d'un module avec l'outil `ildasm.exe` fourni avec l'environnement de développement .NET. *ildasm* signifie désassembleur de code IL. Cet outil se trouve dans le répertoire `<répertoie-d-installation-de-VisualStudio>\SDK\v2.0\Bin`. Chargez le fichier `Foo1.exe` avec `ildasm.exe` :

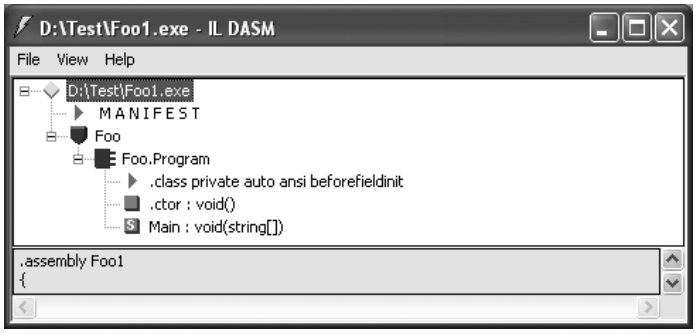


*Figure 2-3 : Vue générale de ildasm*

## *Vue du manifeste*

En double cliquant sur le manifeste le texte suivant apparaît dans une nouvelle fenêtre (certains commentaires ont été enlevés pour plus de clarté) :

```
.module extern Foo2.netmodule
.assembly extern mscorlib   {
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 2:0:0:0
}
.assembly Foo1   {
   .custom instance void
   [mscorlib]System.Reflection.AssemblyCompanyAttribute::.ctor(string) =
   ( 01 00 0E 4C E2 80 99 65 6E 74 72 65 70 72 69 73  65 00 00 ) //
   ...L...entreprise..

   .custom instance void [mscorlib]System.Runtime.CompilerServices.
   CompilationRelaxationsAttribute::.ctor(int32) =
      ( 01 00 08 00 00 00 00 00 )
   .hash algorithm 0x00008004
   .ver 0:0:0:0
}
.file Foo2.netmodule
    .hash = (80 CC 15 14 E2 AB E0 AF D6 BD 55 B9 1B 02 61 10  B4 CF AA 94 )
.file nometadata Image.JPG
    .hash = (0D 84 86 DE 03 E0 05 68 9D 38 F4 B0 B6 19 66 BB  3D 73 76 06 )
.class extern public Foo.UneClasse   {
   .file Foo2.netmodule
   .class 0x02000002
}
.mresource public Image.jpg   {
  .file Image.JPG at 0x00000000
}
.module Foo1.exe
```

```
// MVID: {3C680D21-A6C8-4151-A2A6-9B20B8FDDF27}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x04110000
```

On voit clairement que les fichiers `Foo2.netmodule` et `Image.jpg` sont référencés. On remarque aussi qu'un autre assemblage est référencé, c'est l'assemblage `mscorlib` qui contient entre autre la classe `object`. Tous les assemblages .NET référencent l'assemblage `mscorlib` car celui-ci contient les types de base. À ce titre, l'assemblage `mscorlib` joue un rôle prépondérant et particulier dans la plateforme .NET. Plus d'informations à son sujet sont disponibles en page 94.

## *Analyse de la classe Foo.Program*

En double cliquant sur la classe `Foo.Program`, on obtient la description de cette classe dans une nouvelle fenêtre :

```
.class private auto ansi beforefieldinit Foo.Program
       extends [mscorlib]System.Object
{
} // end of class Foo.Program
```

On voit clairement les drapeaux associés à la classe `Program` (`private`...) et que cette classe dérive de la classe `System.Object`.

## *Analyse du code de la méthode Main()*

En double cliquant par exemple la méthode `Main()`, on obtient le code en langage IL de cette méthode dans une nouvelle fenêtre. Nous présenterons brièvement le langage IL à la fin de ce chapitre :

```
.method public hidebysig static void  Main(string[] argv) cil managed{
  .entrypoint
  // Code size       31 (0x1f)
  .maxstack  1
  .locals init (class [.module Foo2.netmodule]Foo.UneClasse V_0)
  IL_0000:  ldstr      "Bonjour de Foo1"
  IL_0005:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000a:  nop
  IL_000b:  newobj ...
         ... instance void [.module Foo2.netmodule]Foo.UneClasse::.ctor()
  IL_0010:  stloc.0
  IL_0011:  ldloc.0
  IL_0012:  call       void [mscorlib]System.Console::WriteLine(object)
  IL_0017:  nop
  IL_0018:  call       int32 [mscorlib]System.Console::Read()
  IL_001d:  pop
  IL_001e:  ret
} // end of method Program::Main
```

### *Options de ildasm.exe*

L'outil `ildasm.exe` présente des options très intéressantes comme la visualisation du code IL binaire (option *Show Byte*) ou la présentation du code source correspondant en C# (option *Show Source Lines*).

Avec ildasm.exe 2.0 certaines options sont disponibles par défaut alors qu'en version 1.x il fallait utiliser le commutateur /adv en ligne de commande afin de les obtenir. Ces options sont la possibilité d'obtenir des statistiques quant à la taille en octets de chaque section d'un assemblage et quant à l'affichage des informations sur les métadonnées.

Par l'intermédiaire de `ildasm.exe` vous pouvez donc faire du « *reverse engineering* » sur un assemblage. Vous pouvez récupérer des informations comme le code IL des méthodes ou les noms des éléments d'un assemblage (classes, méthodes...). En revanche, les commentaires du code source d'un assemblage ne sont pas conservés dans l'assemblage. Ils ne peuvent donc pas être retrouvés avec `ildasm.exe`.

## *L'outil Reflector*

Depuis plusieurs années, l'utilitaire Reflector développé par *Lutz Roeder* a détrôné ildasm.exe et est devenu l'outil incontournable pour analyser des assemblages .NET. Cet outil est téléchargeable gratuitement à `http://www.aisto.com/roeder/dotnet/`. Voici une copie d'écran du traitement de notre exemple par Reflector :

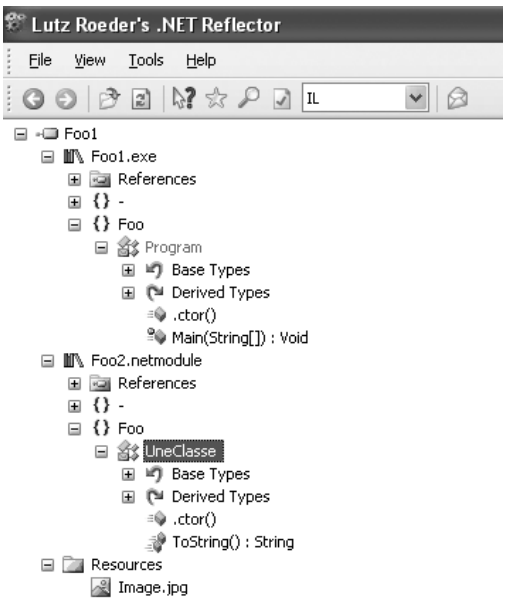


*Figure 2-4 : Vue générale de Reflector*

En plus d'un interface conviviale, Reflector présente un panel de fonctionnalités très intéressantes et absentes de ildasm.exe telles que :

- La décompilation en C# ou en VB.NET du code en IL d'un assemblage ; par exemple la décompilation en VB.NET de la méthode `Program.Main()` est :

```
Public Shared Sub Main(ByVal argv As String())
      Console.WriteLine("Bonjour de Foo1")
      Dim classe1 As New UneClasse
      Console.WriteLine(classe1)
      Console.Read
End Sub
```

- La possibilité de construire localement pour un élément du code le graphe des appelants et des appelés.
- De nombreux *addins* tel que *statement graph* de *Jonathan de Halleux*, qui permet de décompiler une méthode sous la forme d'un organigramme, *Reflector Diff* de *Sean Hederman* qui permet de mettre en évidence les différences entre deux versions d'un assemblage ou *file disassembler* de *Denis Bauer* qui permet de récupérer le code source d'une application à partir de ses assemblages. Une conséquence est que *file disassembler* permet de migrer une application, par exemple de VB.NET vers C#.

# *Attributs d'assemblage et versionning*

## *Attributs standard associables aux assemblages*

Pour aborder cette section il faut être familier avec la notion d'attribut .NET. Si vous ne savez pas ce qu'est un attribut .NET nous vous conseillons de lire la section page 248.

Vous avez la possibilité d'utiliser certains attributs du *framework* pour ajouter des informations telle que son numéro de version à l'assemblage courant. Ces attributs doivent être déclarés dans un fichier source de l'assemblage après les déclaratives `using` et avant les éléments du programme. Si vous éditez un projet sous l'environnement de développement *Visual Studio*, les attributs relatifs à l'assemblage se trouvent par défaut dans le fichier *AssemblyInfo.cs*.

Pour illustrer ceci, le code de l' utilise l'attribut `AssemblyCompanyAttribute` qui est un attribut d'assemblage permettant de préciser le nom de l'éditeur de logiciel qui a fabriqué l'assemblage. Si vous regardez les propriétés du module principal `Foo1.exe`, l'information « `L'entreprise` » est visible, comme le montre la figure 2-5 ci-dessous :

Voici la liste des attributs standard relatifs aux assemblages les plus utilisés :

- Les attributs d'information :
  - `AssemblyCompany` : associe à l'assemblage le nom de l'éditeur de logiciel qui l'a développé.
  - `AssemblyProduct` : associe à l'assemblage le nom du produit (de la solution) auquel il appartient.
  - `AssemblyTitle` : associe un titre à l'assemblage.
  - `AssemblyDescription` : associe une description à l'assemblage.
  - `AssemblyFileVersion` : associe un numéro de version du fichier produit par la compilation.
  - `AssemblyInformationalVersion` : associe le numéro de version du produit.

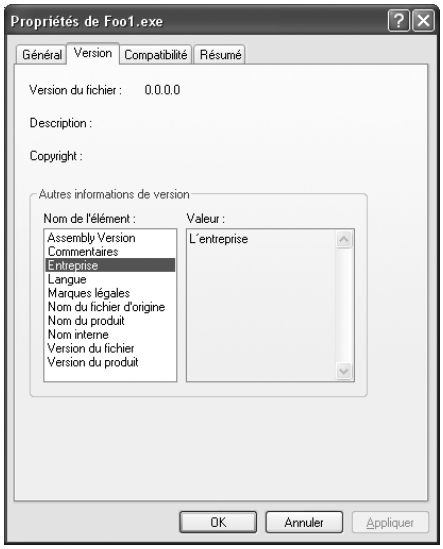


*Figure 2-5 : Propriétés d'un fichier d'un assemblage*

- Les attributs utilisés pour la constitution du nom fort (voir un peu plus loin pour la définition d'un assemblage à nom fort) :
  - `AssemblyVersion` : spécifie le numéro de version de l'assemblage utilisé pour constituer son nom fort.
  - `AssemblyCulture` : spécifie la culture de l'assemblage.
  - `AssemblyKeyFile` : spécifie le fichier d'extension `.snk` (*strong name key*) généré par l'exécutable `sn.exe`.
  - `AssemblyFlags` : spécifie si l'assemblage peut être exécuté côte à côte ou non. La notion d'utilisation côte à côte d'un assemblage est exposée page 65. Si l'exécution côte à côte est autorisée, cet attribut spécifie si l'utilisation côte à côte peut être faite dans un même domaine d'application, dans un même processus ou seulement sur la même machine.
- Les attributs relatifs à l'utilisation de l'assemblage au sein d'une application COM+. On peut citer les attributs `ApplicationID`, `Application-Name`, `ApplicationActivation`.

Dans l'article **Setting Assembly Attributes** des **MSDN** vous pouvez avoir plus de détails sur les attributs standard relatifs aux assemblages.

## *Numéro de version d'un assemblage*

Un numéro de version est composé de quatre numéros :

- Le numéro majeur.
- Le numéro mineur.
- Le numéro de compilation.

- Le numéro de révision.

Vous pouvez fixer le numéro de version de l'assemblage avec l'attribut `AssemblyVersion`. Il est possible d'utiliser dans cet attribut un astérisque qui laisse le choix au compilateur de fixer les numéros de compilation et de révision, par exemple « 2.3.* ». Dans ce cas le numéro de compilation sera le nombre de jours écoulés depuis le 1er janvier 2000, et le numéro de révision sera le nombre de secondes (divisé par deux car 24*60*60=86400 et 86400 > 65536) écoulées dans la journée. Ce processus de datation du numéro de version d'un assemblage est très utile pour obtenir des numéros de version croissants mais toujours différents.

Il existe en fait trois types de numéros de version pour un même assemblage ce qui entraîne une certaine confusion. Seul le numéro de version de l'assemblage est utilisé par le CLR pour permettre la possibilité de stockage d'un assemblage côte à côte. Les deux autres numéros de version, le numéro de version du produit (fixé par l'attribut `AssemblyInformationalVersion`) et le numéro de version du fichier (fixé par l'attribut `AssemblyFileVersion`) sont purement informatifs.

Lorsque vous estimez que plusieurs assemblages doivent constamment avoir le même numéro de version, vous pouvez faire en sorte que leurs projets référencent le même fichier qui contient ce numéro. Néanmoins, sachez que si vous avez rencontrez ce besoin, il est peut être judicieux de rassembler le code de vos assemblages dans un seul assemblage.

La définition des règles d'incrémentation des composantes des versions des assemblages est un sujet sensible qui doit être mûrement réfléchit. Il n'y a pas de bonnes solutions dans l'absolu car ces règles sont fonctions du modèle commercial de chaque l'entreprise. Elles varient selon qu'un assemblage est utilisé par un ou plusieurs produits, externes ou internes à l'entreprise, selon que l'entreprise vend des bibliothèques de classes ou des produits exécutables etc. Voici quelques recommandations :

- Le numéro majeur : à incrémenter lorsqu 'un ensemble significatif de fonctionnalité ont été ajouté.
- Le numéro mineur : à incrémenter lorsqu'une fonctionnalité a légèrement évolué ou lorsqu'un membre visible de l'extérieur (par exemple une méthode protégée d'une classe publique) a changé ou lorsqu'un bug majeur a été corrigé.
- Le numéro de compilation : à incrémenter à chaque recompilation du produit destinée à être utilisée à l'extérieur de l'entreprise (en général pour correction de bug mineur).
- Le numéro de révision : à incrémenter à chaque compilation.

## *Notion d'assemblages amis*

Bien souvent, lorsque vous développez un *framework* contenant plusieurs assemblages vous rencontrez le problème suivant : vous souhaitez que les types définis dans un assemblage du *framework*, soit accessibles à partir du code des autres assemblages du *framework* sans toutefois être accessibles à partir du code des assemblages qui utilisent le *framework*. Aucun des niveaux de visibilité des types du CLR public et internal sont à même de résoudre ce problème. Dans cette situation, il faut avoir recours à l'attribut d'assemblage System.Runtime.CompilerServices.InternalsVisibleToAttribute. Cet attribut permet de spécifier des assemblages qui ont accès aux types non publics de l'assemblage sur lequel il s'applique. On dit que se sont des *assemblages amis* de l'assemblage qui contient les types non publics.

L'utilisation de cet attribut permet de préciser zéro, une ou plusieurs composantes du nom fort d'un assemblage amis selon les différentes versions d'un assemblage avec lesquelles on souhaite « *être amis* » :

```
using System.Runtime.CompilerServices ;
...
[assembly:InternalsVisibleTo("AsmAmi1")]
[assembly:InternalsVisibleTo("AsmAmi2,PublicKeyToken=0123456789abcdef")]
[assembly:InternalsVisibleTo("AsmAmi3,Version=1.2.3.4")]
...
```

# *Assemblage à nom fort (strong naming)*

## *Introduction*

Depuis quelques années, les systèmes d'exploitation *Windows* présentent une technologie nommée *Authenticode* permettant d'authentifier un fichier exécutable. C'est-à-dire que lorsque vous exécutez un programme sur votre machine, vous pouvez être certains que ce programme est bien celui qui a été développé par l'entreprise dans laquelle vous avez confiance. Cette technologie fait l'objet d'une section en page 230.

La technologie du *nom fort* (*strong name* en anglais) que nous allons décrire dans la présente section peut potentiellement servir à authentifier les assemblages .NET dans le cas de grosses organisations telles que *Microsoft* ou l'*ECMA*. Cependant, elle est surtout destinée à nommer les assemblages d'une manière unique. Cette possibilité est inédite puisque jusqu'à présent, on utilisait exclusivement les noms des fichiers pour identifier les unités de déploiement. Avec .NET, lorsqu'un assemblage évolue vers de nouvelles versions, il y a plusieurs fichiers avec le même nom mais pas la même version. Il existe un répertoire spécial prévu pour stocker plusieurs versions d'un même assemblage. Ce répertoire, appelé le répertoire GAC, est présenté page 64.

Concrètement un assemblage acquiert un nom fort lorsqu'il est signé numériquement. Lorsqu'un assemblage a un nom fort, ce nom peut être formaté en une chaîne de caractères contenant ces quatre informations :

- Le nom du fichier ;
- le jeton de clé publique de la signature numérique (tous ces termes sont expliqués ci-dessous) ;
- la version de l'assemblage, celle spécifiée avec l'attribut `AssemblyVersion`, présenté dans la section précédente ;
- la culture de l'assemblage, que nous détaillons dans la prochaine section.

Le nom est fort dans le sens où il permet d'identifier l'assemblage d'une manière unique. Cette unicité est garantie par la seule composante « jeton de clé publique ».

Un nom fort est aussi sensé rendre un assemblage infalsifiable. *Richard Grimes* décrit à l'URL `http://www.grimes.demon.co.uk/workshops/fusionWSCrackThree.htm` une méthode permettant de craquer un nom fort d'un assemblage (i.e cette méthode permet de falsifier un assemblage signé). Cette méthode se base sur un bug du CLR version 1.1 corrigé en version 2.0. Il n'y a plus à notre connaissance de telles failles de sécurité en .NET 2.0.

Avant de nous pencher sur la création d'un assemblage à nom fort, nous vous conseillons de vous familiariser avec les notions de clés publiques/clés privées et de *signature numérique* présentées en page 223. En apposant une signature numérique à leurs assemblages, un auteur ou une entreprise peuvent l'authentifier. *Microsoft* et l'organisation ECMA ont chacun un couple de clé publique/clé privée qu'ils utilisent pour authentifier leurs assemblages.

## *L'outil sn.exe*

La première étape pour un développeur ou une entreprise qui désire créer des assemblages à noms forts est de créer un couple de clé privée/clé publique. Cette opération utilise des algorithmes mathématiques complexes. Heureusement le Framework .NET met à notre disposition l'outil `sn.exe` (`sn` pour *strong name*) qui effectue cette tâche. Cet outil peut fabriquer un fichier d'extension `.snk` (pour *strong name key* en anglais) qui contient un couple de clé privée/clé publique en utilisant l'option `-k`. Par exemple :

```
C:\Test>sn.exe -k Cles.snk

Microsoft (R) .NET Framework Strong Name Utility  Version 2.0.XXXXX
Copyright (C) Microsoft Corporation. All rights reserved.

Key pair written to Cles.snk

C:\Test>
```

`sn.exe` permet aussi de visualiser la clé publique et la clé privée contenues dans un fichier d'extension `.snk` avec l'option `-tp` :

```
C:\Test>sn.exe -tp Cles.snk

Microsoft (R) .NET Framework Strong Name Utility  Version 2.0.XXXXX
Copyright (C) Microsoft Corporation. All rights reserved.

Public key is
07020000002400005253413200040000010001005 1a7dadde83cf10e8b7c6cd99e4d062b
1aca430e11db76365ab29d6c31fc93a7bea6def9d7b2e8a7c568b0d5ada5e8e131cb98ea
3e9a876236b33b362e433fdd62bb4c5cc5ea23f1dfa76d35b5412d812f66d03e079009ea
76462392663bc08ab5f937524e794948532c679db5eda50210f8a8b2b8b186fcb342859c
48ea76d609d108b1957d3888f75b270cf85029ede8437c36b4ae59342c5fa7aacdb453c7
465cc7027405930627a5b153e5f48cdd0375840bf6feaa3548aa421ab5138fb095efa5d5
81ae61bd9248ac97293ee69b139ef9ae79d907c5cf2c194adf7c2723e269b5eef55157c4
095fccf436d7db1893aed8c63d57e9d5eba5c1dd88f8bda81b6c74b77899071823c85c86
2254865337d2b70d545d17de9b8471527bbd54d4e1bd6cb6b53fed9135c9c7b1b1af2b27
ab0414b423b61334c9c1adb0700145ba1354848b081e09e8a860d24fb9ea6c48ac2657f1
9ff1fab37a177744c377d9d7d09f34498901f4439bc6754b4ac0efcc4d84d4a6c22a05c2
eecaec3f7fabf8b45555d4788eaeda815cf743001477a8c31c24c04b4016f4ef3401617e
22441b95ca78265a0a6133150ca03c2886d4e3893f9d1dc6a3e2d8770a63b8fbd0db52d8
176bbda6e1f4074d9dfda916cf316294f0499eade4aa47d1b780627ab6fb7beb5aa48412
9062d3152e6b6585c865d319018727c1a34866484018f5f1c0ab0bf2b35e63a8a3bbf7a0
b6aeeb110f4b162426a977dc2034adf08ec41cc5d20f2d6beac92a1619aff0e25030e30a
02570eb9ad74eba0f2aba90b18789ae99f8da72
```

```
Public key token is f2bf46103b24f5f0
```

En fait cet outil affiche la clé publique (écrite en gras) puis la clé privée. La phrase « *public key is :* » est donc erronée. De même, ce qui est présenté comme étant le jeton de clé publique est une valeur de hachage du couple clé publique/clé privée. Cependant si *Microsoft* a fait le choix de ces termes, c'est parce qu'il suppose que la plupart du temps le mécanisme de signature retardée (exposé un peu plus loin) est utilisé. Dans ce cas ces termes prennent toute leur signification.

Lors de la fabrication d'une couple clé publique/clé privée par l'outil `sn.exe`, la clé publique fait 128 octets (grisé) + 32 octets d'en-tête (non grisé) et la clé privée fait 436 octets (non grisé).

## *Les jetons de clés publiques*

La clé publique étant volumineuse et difficile à manipuler, l'architecture .NET associe un *jeton de clé publique* (*public key token* en anglais) à chaque clé publique. Un jeton de clé publique est un numéro codé sur huit octets. Ce numéro est calculé à partir de la clé publique en utilisant un algorithme de hachage. Il identifie d'une manière quasi unique la clé publique, tout en étant beaucoup plus maniable de par sa taille réduite. Les jetons de clés publiques ont été créés pour :

- Réduire la taille des noms forts (qui incluent ces jetons à la place de la clé publique elle même).
- Réduire la taille des assemblages qui référencent de nombreux assemblages à noms forts.

Un assemblage signé avec la clé privée dont le jeton de la clé publique correspondante est `b03f5f7f11d50a3a`, est garanti avoir été produit par *Microsoft*. Un assemblage signé avec la clé privée dont le jeton de la clé publique correspondante est `b77a5c561934e089`, est garanti avoir été produit par l'organisme *ECMA*. Ces affirmations ne seraient plus valables, si un jour une de ces clés privées était piratée.

## *Signer un assemblage*

Pour attribuer un nom fort à un assemblage vous pouvez soit utiliser les options /keycontainer et /keyfile du compilateur csc.exe soit utiliser le menu *Propriété du projet* ▶ *Signature* ▶ *Signe l'assemblage* de *Visual Studio 2005*.

Vous pouvez aussi utiliser l'attribut d'assemblage `AssemblyKeyFile` dans le code source du module principal. Cet attribut accepte un argument qui est le nom d'un fichier d'extension `.snk`. Par exemple :

```
[AssemblyKeyFile("Cles.snk")]
```

Lorsque cet attribut est ajouté, le compilateur signe l'assemblage mais émet un avertissement vous mettant en garde qu'une des deux premières techniques citée est préférable.

Lorsqu'un assemblage doit être signé par une de ces trois techniques, le compilateur inclut une signature numérique et la clé publique dans le corps de l'assemblage. L'algorithme utilisé pour hacher le contenu de l'assemblage se nomme SHA-1 (*Secure Hash Algorithm*). Vous pouvez choisir un autre algorithme de hachage de l'assemblage, comme le très connu MD5 (MD pour Message Digest) au moyen de l'attribut d'assemblage `AssemblyAlgorithmId` qui prend en argument une valeur de l'énumération `AssemblyHashAlgorithm`. La taille d'une valeur de hachage créée par l'algorithme SHA-1 est de 20 octets. Cette taille peut varier selon l'algorithme utilisé.

Le compilateur encrypte la valeur de hachage avec la clé privée et obtient ainsi la signature numérique RSA de l'assemblage. La clé publique est insérée dans la table *AssemblyDef* du manifeste tandis que la signature numérique est insérée dans une partie spéciale de l'entête CLR. Voici le schéma décrivant le processus de signature d'un assemblage :

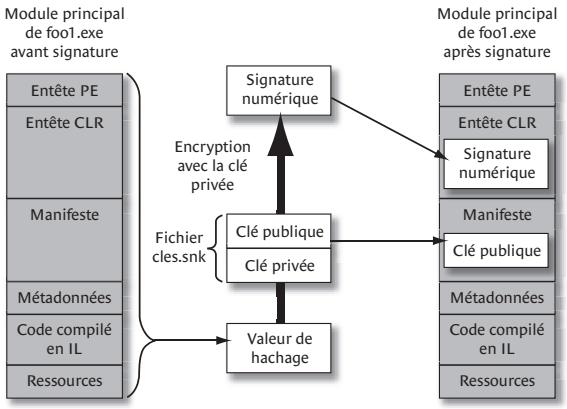


*Figure 2-6 : Signer un assemblage*

## *Un exemple*

Voici le code d'un assemblage Foo1 :

*Exemple 2-3 :* *Foo1.cs*

```
using System ;
namespace Foo {
   class Program {
      public static void Main( string[] argv ){
         Console.WriteLine( "Bonjour de Foo1" ) ;
      }
   }
}
```

Compilons et signons cet assemblage avec la clé `Cles.snk` :

```
>csc.exe /keyfile:Cles.snk Foo1.cs
```

Analysons le manifeste de l'assemblage `Foo1.exe` avec l'outil `ildasm.exe` (pour plus de clarté nous avons enlevé quelques commentaires) :

```
.assembly extern mscorlib   {
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )       // .z\V.4..
  .ver 2:0:0:0
}
.assembly Foo1   {
  .custom instance void
```

```
  [mscorlib]System.Reflection.AssemblyKeyFileAttribute::.ctor(string) =
     ( 01 00 08 43 6C 65 73 2E 73 6E 6B 00 00 ) // ...Cles.snk..
  [mscorlib]System.Diagnostics.DebuggableAttribute::.ctor
     (bool, bool) = ( 01 00 00 01 00 00 )

  .publickey = (   00 24 00 00 04 80 00 00 94 00 00 00 06 02 00 00
                   00 24 00 00 52 53 41 31 00 04 00 00 01 00 01 00
                   51 A7 DA DD E8 3C F1 0E 8B 7C 6C D9 9E 4D 06 2B
                   1A CA 43 0E 11 DB 76 36 5A B2 9D 6C 31 FC 93 A7
                   BE A6 DE F9 D7 B2 E8 A7 C5 68 B0 D5 AD A5 E8 E1
                   31 CB 98 EA 3E 9A 87 62 36 B3 3B 36 2E 43 3F DD
                   62 BB 4C 5C C5 EA 23 F1 DF A7 6D 35 B5 41 2D 81
                   2F 66 D0 3E 07 90 09 EA 76 46 23 92 66 3B C0 8A
                   B5 F9 37 52 4E 79 49 48 53 2C 67 9D B5 ED A5 02
                   10 F8 A8 B2 B8 B1 86 FC B3 42 85 9C 48 EA 76 D6 )
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module Foo1.exe
// MVID: {5DD7C72B-D1C1-49BB-AB33-AF7DA5617BD1}
.imagebase 0x00400000
.file alignment 512
.stackreserve 0x00100000
.subsystem 0x00000003
.corflags 0x00000009
// Image base: 0x03240000
```

La clé publique est bien la même que celle que l'on a surlignée plus haut, lors de l'analyse du fichier `Cles.snk`. L'outil `ildasm.exe` ne nous permet pas de visualiser ni le jeton de clé publique ni la signature numérique. L'outil `sn.exe` nous permet de visualiser le jeton de clé publique avec l'option -T :

```
C:\Code\CodeDotNet\ModuleTest>sn.exe -T Foo1.exe

Microsoft (R) .NET Framework Strong Name Utility  Version 2.0.XXXXX
Copyright (C) Microsoft Corporation. All rights reserved.

Public key token is c64b742bd612d74a
```

L'assemblage `mscorlib` est référencé et son jeton de clé publique est connu. C'est normal, puisque ce jeton fait partie intégrante du nom fort de l'assemblage. En fait le jeton de clé publique a été créé pour le référencement d'autres assemblages. Nous rappelons que la taille d'une clé publique étant de 128 octets, la taille des assemblages référençant beaucoup d'autres assemblages aurait été trop grosse sans la technique du jeton à clé publique.

Un assemblage à nom fort ne peut référencer un autre assemblage qui n'a pas de nom fort. En revanche, un assemblage sans nom fort, peut référencer un assemblage à nom fort.

Dans le cas où un assemblage est constitué de plusieurs modules, les modules autres que le module principal n'ont pas de clé publique. En revanche durant la compilation du module principal, le compilateur calcule une valeur de hachage pour chaque module. Ces valeurs sont

intégrées dans les entrées de la table *FileDef* du manifeste. Ces valeurs sont donc prises en compte lors du calcul de la valeur de hachage du module principal. Grâce à cette astuce les modules d'un assemblage à nom fort sont eux aussi nommés d'une manière unique et infalsifiables (en .NET 2.0), sans intégrer une signature numérique.

## *Le mécanisme de signature retardée*

Utiliser un système de signature numérique pour rendre ses assemblages infalsifiables est une formidable fonctionnalité qu'offre .NET. Cependant la mise en œuvre de cette technique oblige à fournir un accès en lecture aux fichiers contenants les couples clé publique/clé privée à tous ceux qui sont susceptibles de compiler un assemblage à nom fort dans l'entreprise. Dans une entreprise comme *Microsoft*, ils sont des milliers. Nul doute qu'après un certain temps, un employé finirait par divulguer la clé privée sur internet. La seule manière de limiter ce risque est de limiter le nombre d'employés qui ont accès à la clé privée. .NET fournit le mécanisme de *signature retardée* à cet effet. Concrètement un assemblage est signé avec la clé privée après avoir été compilé et testé et seulement avant d'être packagé et déployé chez les clients. Dans une grande entreprise, seul un ou quelques employés sont habilités à faire cette manipulation. Le mécanisme de signature retardée nécessite trois étapes :

- Il faut d'abord construire un fichier d'extension `.snk` qui ne contient qu'une clé publique. Pour cela il faut utiliser l'option `-p` de `sn.exe`.

```
C:\Code>sn -p Cles.snk ClePublique.snk

Microsoft (R) .NET Framework Strong Name Utility  Version 2.0.XXXX
Copyright (C) Microsoft Corporation. All rights reserved.

Public key written to ClesPubliques.snk

C:\Code>
```

- Le fichier `ClePublique.snk` est distribué aux développeurs qui l'utilisent durant le développement des assemblages à la place du fichier `Cles.snk` (donc dans l'attribut `AssemblyKeyFile`). L'attribut d'assemblage `AssemblyDelaySign` initialisé avec l'argument booléen `true`, doit être utilisé. Le compilateur comprend qu'il ne doit pas signer le module principal. Cependant le compilateur prévoit l'espace nécessaire à la signature, dans le module principal, insère la clé publique dans le manifeste et calcule la valeur de hachage des modules sans manifeste.
- Lorsque les équipes de développement fournissent des assemblages prêts à être packagés et déployés, il suffit de signer les assemblages avec l'option -R de `sn.exe`.

```
C:\Code>sn.exe -R Foo1.exe Cles.snk

Microsoft (R) .NET Framework Strong Name Utility  Version 2.0.XXX
Copyright (C) Microsoft Corporation. All rights reserved.

Assembly 'Foo1.exe' successfully re-signed

C:\Code>
```

Vous pouvez aussi retarder la signature d'un assemblage avec le compilateur csc.exe en utilisant conjointement les options /delaysign, /keycontainer et /keyfile.

La technique de la signature retardée peut être aussi utilisée pour signer des assemblages modifiés après la compilation, par exemple avec un framework de programmation orientée aspect.

# *Internationalisation et assemblages satellites*

## *Notion de culture et de globalisation d'une application*

Certaines applications doivent pouvoir être utilisées par des utilisateurs de différentes nationalités parlant différentes langues. Tous ce qui peut être présenté à l'utilisateur par l'application (chaînes de caractères, images, animations, sons, dates, nombres à virgules etc) doit alors être fournis pour chaque *culture* supportée par l'application. On appelle ce processus la *globalisation* ou parfois l'*internationalisation* voire la *localisation* de l'application.

Une culture est une association pays-langue/dialecte telle que « fr-FR » pour le français en France, « fr-CA » pour le français au Canada ou « en-US » pour l'anglais des Etats Unis. La liste des cultures standard du framework .NET est présentée dans l'article **CultureInfo Class** des **MSDN**.

Nous avons vu qu'une culture peut être précisée dans le nom fort d'un assemblage. Néanmoins, vous êtes obligé d'utiliser la culture neutre pour tout assemblage contenant du code. Concrètement, la chaîne de caractères spécifiée pour l'attribut `CultureInfo` doit toujours être vide lorsque votre assemblage contient du code. Pour indexer les ressources à partir d'une culture, il faut créer un assemblage pour chaque culture, qui ne contient que des ressources. Ces assemblages particuliers qui font l'objet de la présente section sont appelés les *assemblages satellites*.

## *Fichiers de ressources*

Concrètement, pour exploiter des ressources de type chaînes de caractères ou des ressources stockées dans un format binaire (images, animations, sons...) dans un assemblage, il faut procéder selon les quatre étapes suivantes :

- Editer le *fichier de ressources* dans un format exploitable par les humains (extension du fichier `.txt` ou `.resx`).
- Convertir le fichier de ressources dans un format binaire adapté à la lecture par un programme (extension du fichier `.resources`).
- Inclure le fichier d'extension `.resources` dans un assemblage (satellite ou non);
- Exploiter les ressources dans votre code source au moyen de la classe `System.Resources.ResourceManager`. Bien souvent, on se sert d'une classe générée qui encapsule l'accès à cette classe.

Nous commencerons par montrer comment faire ces manipulations « à la main » à l'aide d'outils en ligne de commande. Ensuite, nous verrons que *Visual Studio 2005* permet d'automatiser grandement ce processus.

Un fichier de ressources donné cible une seule culture. Il existe trois formats de fichiers de ressources, utilisés lors de différentes étapes du développement d'une application :

- Les fichiers de ressources d'extension `.txt` : Ces fichiers associent des identifiants aux chaînes de caractères d'une culture. Par exemple voici un tel fichier dont la culture destination est l'anglais :

  *Exemple 2-4 :* *`MyRes.txt`*

  ```
  Bonjour = Hello!
  AuRevoir = Bye...
  ```

- Les fichiers de ressources d'extension `.resx` : Ces fichiers sont au format XML. Ils associent des identifiants aux chaînes de caractères d'une culture. À la différence des fichiers de ressources d'extension `.txt`, les fichiers d'extension `.resx` peuvent aussi associer des identifiants à des ressources stockées dans un format binaire. Dans un fichier d'extension `.resx`, l'information binaire est convertie au format UNICODE en utilisant l'encodage *Base64*. L'utilitaire `resxgen.exe` sert à convertir une telle ressource (par exemple une image au format `bmp` ou `jpg`) en un fichier d'extension `.resx`. L'utilisation de *Visual Studio 2005* simplifie considérablement la visualisation, l'édition et la maintenance des fichiers d'extension `.resx` grâce à un éditeur dédié.
- Les fichiers de ressources d'extension `.resources` : Ces fichiers sont logiquement équivalents aux fichiers d'extension .resx. À la différence de ces derniers, ces fichiers sont dans un format binaire. Ce sont ces fichiers qui sont intégrés dans les assemblages à l'étape de la compilation.

Comme nous l'avons expliqué, seul un fichier de ressources au format `.resources` peut être intégré dans un assemblage. L'outil `resgen.exe` utilisable en ligne de commande permet de construire un fichier de ressources dans un de ces formats, à partir d'un autre fichier de ressources dans un autre format. `resgen.exe` connaît le format du fichier en entrée et du fichier en sortie grâce aux extensions de leurs noms. Voici un exemple d'utilisation :

```
>resgen.exe MyRes.txt MyRes.resources
```

Ne convertissez pas un fichier de ressources au format `resx` ou `resources` qui contient au moins une ressource qui n'est pas une chaîne de caractères, en un fichier de ressources au format `txt`.

## *Utiliser des ressources dans un assemblage*

La classe `System.Resources.ResourceManager` doit être utilisée pour exploiter les ressources selon la culture courante. Cependant, on préfère en général se servir d'une classe générée par l'outil resgen.exe pour encapsuler les accès à cette classe. Le fichier source d'une telle classe peut être généré à partir d'un fichier de ressources au format .txt, .resx ou .ressources avec l'option `/str` :

```
>resgen.exe MyRes.resources /str:cs
```

L'option `/str:cs` indique que nous souhaitons générer notre fichier source en C#. Nous aurions pu aussi spécifier `/str:vb` pour l'obtenir en VB.NET. Voici un extrait pertinent :

*Exemple :* *MyRes.cs*

```
internal class MyRes {
   private static System.Resources.ResourceManager resourceMan ;
   private static System.Globalization.CultureInfo resourceCulture ;
...
   internal static System.Resources.ResourceManager ResourceManager {
     get {
       if ((resourceMan == null)) {
         System.Resources.ResourceManager temp = new
           System.Resources.ResourceManager("MyRes",
                                        typeof(MyRes).Assembly) ;
           resourceMan = temp ;
         }
       return resourceMan ;
     }
   }
   internal static global::System.Globalization.CultureInfo Culture {
     get { return resourceCulture ; } set{ resourceCulture = value ; } }
...
   internal static string Bonjour {
     get{return ResourceManager.GetString("Bonjour", resourceCulture);}}
...
   internal static string AuRevoir {
     get {return ResourceManager.GetString(
             "AuRevoir", resourceCulture) ; } }
}
```

On comprend alors qu'en plus d'encapsuler l'accès à la classe ResourceManager, la classe MyRes présente des propriétés qui permettent d'accéder aux ressources d'une manière typée. Il ne nous reste plus qu'à utiliser ces propriétés pour pouvoir exploiter nos ressources. Le programme suivant affiche « hello » :

*Exemple 2-5 :* *Program.cs*

```
class Program {
   static void Main() {
      System.Console.WriteLine( MyRes.Bonjour ) ;
   }
}
```

Il est intéressant de comparer ce code au code d'un programme qui n'utilise pas la classe générée MyRes :

*Exemple 2-6 :* *Program.cs*

```
using System.Resources ;
class Program {
   static void Main() {
      ResourceManager rm = new ResourceManager( "MyRes" ,
                                              typeof(MyRes).Assembly) ;
      System.Console.WriteLine( rm.GetString("Bonjour") ) ;
```

```
    }
}
```

La classe `ResourceManager` présente plusieurs versions surchargées de `GetString()`. De plus, elle présente la méthode plus générale `GetObject()` qui peut être utilisée pour charger des chaînes de caractères. Cette méthode peut aussi être utilisée pour charger des ressources stockées dans un format binaire telles que des images. Par exemple :

```
...
   System.Drawing.Bitmap image = (Bitmap) rm.GetObject("UneImage") ;
   string s = (string) rm.GetObject("Bonjour") ;
...
```

Dans le cas d'utilisation de la classe générée par resgen.exe, une propriété de type System.Drawing.Bitmap représente l'accès à une ressource de type image.

```
...
   System.Drawing.Bitmap image = MyRes.UneImage ;
...
```

Intéressons nous maintenant à la compilation de ce programme. Dans le cas où l'assemblage contient du code, il faut préciser les noms des fichiers de ressources d'extension .resources à la compilation de l'assemblage. Le compilateur `csc.exe` du langage C# prévoit à cet effet les options `/resource` et `/linkresource`. Par exemple :

```
>csc.exe /resource:MyRes.resources,MyRes.resources Program.cs MyRes.cs
```

La syntaxe avec une virgule, qui sépare le nom physique du fichier de ressources (à gauche de la virgule) du nom logique (à droite de la virgule) qui sera utilisé dans le code de l'assemblage pour l'identifier. Les noms logique et physique sont ici identiques et égaux à `"MyRes.resources"`. Notez que l'option `/resource` copie physiquement le contenu du fichier ressource dans le corps du module compilé. Vous pouvez aussi utiliser l'option `/linkresource` pour référencer le fichier de ressources à partir du module principal.

## *Fabriquer un assemblage satellite*

Nous avons vu comment encapsuler des ressources dans un assemblage et comment les exploiter. Jusqu'ici, nous n'avons pas utilisé une culture en particulier. Les ressources que nous avons exploitées étaient relatives à ce que l'on nomme la culture invariante. C'est la culture prise par défaut lorsque aucune culture n'est spécifiée. Intéressons nous maintenant à la notion d'assemblage satellite qui permet de diversifier les cultures pour une même application.

L'outil `al.exe` (`al` pour *assembly linker*), utilisable en ligne de commande, permet de produire un assemblage bibliothèque (i.e un assemblage dont le module principal est dans un fichier d'extension `.dll`) à partir d'un ou plusieurs modules. `al.exe` permet donc de construire des assemblages tels que les assemblages satellites qui ne contiennent que des ressources. En page 67 nous expliquons que l'outil `al.exe` permet aussi la création d'assemblages de stratégie d'éditeur.

Un assemblage satellite donné contient des ressources relatives à une seule culture. Voici un exemple d'utilisation de `al.exe` pour construire un assemblage satellite. Notez l'utilisation de l'option `/c` pour préciser la culture (en l'occurrence, espagnole) :

```
>al.exe /out:es-ES\Program.Resources.dll /c:es-ES
/embed:es-ES\MyRes.es-ES.resources
```

Nous utilisons en entrée de al.exe avec l'option /embed un fichier nommé MyRes.es-ES.ressources. Ce fichier a été fabriqué avec l'outil resgen.exe.à partir du fichier MyRes.es-ES.txt suivant :

*Exemple 2-7 :* *MyRes.es-ES.txt*

```
Bonjour = ¡Hola!
AuRevoir = Adiós...
```

L'option /embed fait en sorte que le contenu du fichier MyRes.es-ES.ressources soit physiquement inclus dans le fichier Program.Ressources.dll.

## *Déployer et utiliser un assemblage satellite*

La classe `System.Resources.ResourceManager` sait gérer la notion d'assemblage satellite. Elle est notamment capable de charger un assemblage satellite à l'exécution. La culture qui est prise en compte est celle qui est précisée par la valeur de la propriété CultureInfo `Thread.CurrentUICulture{get;set}` du thread courant. Pour bénéficier de ce service, il est nécessaire de se plier à une certaine discipline :

- Le nom du module principal d'un assemblage satellite (celui créé avec l'outil `al.exe`) doit être `[nom de l'assemblage qui contient le code qui manipule les ressources].Resources.dll`.
- Un assemblage satellite correspondant à une culture `xx-XX` doit se trouver dans le sous répertoire `xx-XX` (par rapport au répertoire de l'assemblage qui contient le code qui manipule les ressources). On remarque donc que les assemblages satellites relatifs à un même assemblage ont tous le même nom. Seul les noms des répertoires qui les stockent permettent de déterminer pour chacun la culture à laquelle il se réfère.

Ces règles sont illustrées par cette organisation des fichiers de notre exemple :

```
\Program.exe                    // fichier fabriqué par csc.exe
\es-ES\Program.Resources.dll    // fichier fabriqué par al.exe
...
```

Si le programme suivant s'exécute dans le contexte ci-dessus, il affiche « ¡Hola! » :

*Exemple 2-8 :* *Program.cs*

```
class Program {
   static void Main() {
      MyRes.Culture =  new System.Globalization.CultureInfo("es-ES");
      System.Console.WriteLine( MyRes.Bonjour ) ;
   }
}
```

En analysant le code de la classe MyRes générée par resgen.exe, on s'aperçoit que la recherche de ressource ne se fait pas selon la culture précisée par la propriété Thread.CurrentUICulture mais selon la valeur de la propriété MyRes.Culture (qui correspond au champ MyRes.resourceCulture). On peut réécrire ce programme sans utiliser la classe MyRes comme ceci :

*Exemple 2-9 :* *Program.cs*

```
using System.Resources ;     // Pour la classe ResourceManager.
using System.Threading ;     // Pour la classe Thread.
using System.Globalization ; // Pour la classe CultureInfo.
class Program {
   static void Main() {
      Thread.CurrentThread.CurrentUICulture = new CultureInfo("es-ES");
      ResourceManager rm = new ResourceManager( "MyRes" ,
                                                typeof(MyRes).Assembly) ;
      System.Console.WriteLine(rm.GetString("Bonjour")) ;
   }
}
```

Les programmes précédents fonctionnent aussi si l'assemblage satellite `Program.Resources.dll` de culture `es-ES` se trouve dans le GAC. Comme tout assemblage situé dans le GAC, un tel assemblage satellite a un nom fort. Il est différencié de ces homologues relatifs à d'autres cultures grâce à la composante culture de ce nom fort.

Lors de la compilation de `Program.exe` vous n'avez aucunement besoin de référencer un des assemblages satellites. Il est ainsi possible de rajouter des assemblages satellites après la compilation de votre programme. Pour cela, vous devez faire en sorte que votre application récupère la culture avec laquelle elle doit s'exécuter. Celle-ci peut par exemple être récupérée à partir d'un fichier de configuration ou à partir de préférences utilisateurs.

## *Éviter les exceptions dues à l'échec de la recherche d'une ressource*

Il est fortement conseillé d'inclure un fichier de ressource correspondant à la culture invariante dans l'assemblage contenant le code. Ainsi, à l'exécution, si l'assemblage satellite correspondant à la culture en cours n'est pas trouvé ou si celui-ci ne contient pas la traduction de certaines ressources, la classe ResourceManager fera en sorte de fournir la valeur de la culture invariante. Si cette valeur n'était pas connue, une exception serait lancée.

Pour vous prémunir contre l'absence d'un assemblages satellites, vous pouvez vérifier son existence avec la méthode ResourceManager.GetResourceSet(). Par exemple :

*Exemple 2-10 :* *Program.cs*

```
using System ;
using System.Globalization ; // Pour la classe CultureInfo.
class Program {
   static void Main() {
      CultureInfo spainCulture = new CultureInfo("es-ES") ;
      if( MyRes.ResourceManager.GetResourceSet(
                     spainCulture , true, false )!= null ) {
        MyRes.Culture = spainCulture ;
        Console.WriteLine(MyRes.Bonjour) ;
      }
      else
         Console.WriteLine("Assemblage satellite es-ES non trouvé !") ;
   }
}
```

## *Visual Studio et les assemblages satellites*

Maintenant que nous avons compris le rôle des assemblages satellites et des outils resgen.exe et al.exe nous pouvons montrer comment exploiter *Visual Studio 2005* pour simplifier le processus de globalisation d'un assemblage.

Par défaut, un projet *Visual Studio 2005* n'a pas de fichier de ressources. Vous pouvez en ajouter avec le menu *Project* ▶ *Add* ▶ *New Item...* ▶ *Resources file*. Un fichier ressource d'extension .resx est alors ajouté au projet. À la compilation du projet, ce fichier sera automatiquement compilé en un fichier d'extension .resources. Ce fichier .resource sera intégré dans l'assemblage compilé ou dans un assemblage satellite selon que *Visual Studio* peut déterminer la culture à laquelle il se réfère en analysant son nom. Par exemple, un projet nommé MyProject qui contient les fichiers ressources suivants ...

```
Res1.resx
Res1.en-US.resx
Res1.fr-FR.resx
Res2.resx
Res2.es-ES.resx
```

...se compile en ceci :

```
~/MyProject.exe        contient Res1.resources et Res2.resources
~/en-US/MyProject.Resources.dll  contient Res1.en-US.resources
~/fr-FR/MyProject.Resources.dll  contient Res1.fr-FR.resources
~/es-ES/MyProject.Resources.dll  contient Res2.es-ES.resources
```

Par défaut, un fichier source nommé XXX.designer.cs est associé à chaque fichier ressources XXX.resx ajouté à un projet. Ce fichier source contient une classe nommée [Nom du projet].XXX similaire à celle qui est générée par l'option /str de l'outil resgen.exe. En général, seuls les fichiers de ressources relatifs à la culture invariante (i.e Res1.resx et Res2.resx dans notre exemple) ont besoin d'avoir une classe associée. Aussi, sachez que pour désactiver la génération de ce fichier source, il suffit de faire : *propriété du fichier* XXX.resx ▶ *Custom Tool* ▶ *Mettre une chaîne de caractères vide à la place de la valeur* ResXFileCodeGenerator

*Visual Studio 2005* contient un éditeur de fichier ressources d'extension .resx. Vous pouvez très facilement ajouter des couples *identifiant/chaînes de caractères* ou *identifiant/fichier*. Dans ce second cas le fichier peut être un fichier image, icone, son, texte ou autre.

En page 695 nous décrivons les facilités de *Visual Studio* 2005 pour localiser les fenêtres d'une application *Windows Forms 2.0*. En page 953 nous décrivons les facilités de *Visual Studio 2005* pour localiser les pages d'une application web ASP.NET 2.

## *Formatage et culture*

Le formatage de certains éléments présentés aux utilisateurs tels que les dates ou les nombres peut dépendre de la culture. Le framework .NET se sert de la valeur de la propriété CultureInfo `Thread.CurrentCulture{get;set}` pour déterminer quelle culture utiliser lors d'une opération de formatage. Par exemple, le programme suivant...

*Exemple 2-11 :*

```
using System.Threading ;     // Pour la classe Thread.
using System.Globalization ; // Pour la classe CultureInfo.
class Program {
   static void Main() {
      Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
      System.Console.WriteLine(System.DateTime.Now.ToString()) ;
      Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-FR");
      System.Console.WriteLine(System.DateTime.Now.ToString()) ;
   }
}
```

...affiche ceci :

```
6/20/2005 10:54:10 PM
20/06/2005 22:54:10
```

# *Introduction au langage IL*

Nous avons vu que les assemblages contiennent du code écrit en langage IL (*Intermediate Language*). Le langage IL est en fait un langage objet à part entière. Il constitue le plus petit dénominateur commun des langages .NET.

Il faut savoir que certaines documentations, notamment celles de *Microsoft*, utilisent le terme de langage *MSIL* pour nommer l'implémentation *Microsoft* du langage IL. D'autres documentations utilisent le terme *CIL* (*Common Intermediate Langage*) pour nommer le langage IL. Le langage IL est entièrement spécifié par l'organisation ECMA. Vous pouvez trouver les spécifications du langage IL sur le site de l'ECMA.

Les outils `ildasm.exe` et `Reflector` présentés précédemment permettent de visualiser le code en langage IL contenu dans un module. Nous allons voir et commenter du code IL bien qu'une présentation complète du langage IL dépasserait le cadre de cet ouvrage. Nous pensons qu'il est bon que les développeurs .NET aient une idée de ce qu'est le langage IL. Pour les programmeurs Java cela ne sera pas sans rappeler le *bytecode*.

Vous pouvez compiler vos propres sources écrites en langage IL, en assemblages, à l'aide du compilateur `ilasm.exe` fournis avec le framework .NET (à ne pas confondre avec l'outil `ildasm.exe`).

Comprenez bien que malgré sa ressemblance avec du langage machine, le langage IL n'est supporté par aucun processeur (pour l'instant du moins). Le code IL est compilé à l'exécution, en un langage machine cible. Ce langage machine est fonction du processeur de la machine. Ce mécanisme de compilation du code IL durant l'exécution, est décrit page 111. Cette idée d'utiliser un langage intermédiaire entre les langages de haut niveau et le langage machine n'est pas nouvelle. Cette idée est exploitée depuis longtemps sous Java et sous d'autres langages/compilateurs.

## *Présentation de la pile et des instructions IL correspondantes*

Pour comprendre cette section, il est nécessaire d'avoir assimilé la notion d'unité d'exécution (i.e la notion de thread) présentée page 136.

Comme beaucoup d'autres langages, IL est un langage avec une *pile* (*stack* en anglais). Une pile est un espace mémoire contigu qui a la particularité d'avoir seulement un point d'accès, appelé le sommet de la pile. Tout comme une pile d'assiette, vous pouvez ajouter une assiette au sommet de la pile ou enlever une assiette au sommet de la pile. En informatique les piles ne contiennent pas des assiettes mais des valeurs typées.

Une pile appartient à un thread. Pour chaque opérande d'une opération exécutée par le thread (opérations arithmétiques, appel de fonction...), il faut faire une copie de sa valeur au sommet de la pile.

En IL, la pile est gérée par le CLR. Dans ce cas le CLR joue le rôle d'un processeur « virtuel ». C'est pour cela que dans le monde Java, l'équivalent du CLR est nommé « machine virtuelle ». Le CLR ne fonctionne pas exactement comme un processeur classique. En effet, les processeurs travaillent avec une pile et des registres alors que le CLR remplit les mêmes fonctions seulement avec la pile.

## *Exemple 1 : les variables locales et la pile*

Le code d'une méthode C# suivant...

```
...
{
   int i1 =5 ;
   int i2 =6 ;
   int i3 = i1+i2 ;
}
...
```

...produit le code IL suivant :

```
.maxstack  2
  .locals ([0] int32 i1,
           [1] int32 i2,
           [2] int32 i3)
  IL_0000:  ldc.i4.5
  IL_0001:  stloc.0
  IL_0002:  ldc.i4.6
  IL_0003:  stloc.1
  IL_0004:  ldloc.0
  IL_0005:  ldloc.1
  IL_0006:  add
  IL_0007:  stloc.2
  IL_0008:  ret
```

Plusieurs remarques peuvent être faites :

- Le fait qu'à aucun moment cette méthode ne charge plus de deux valeurs sur la pile, est sauvé dans un attribut appelé `.maxstack`. La taille de la pile est donc bornée durant la compilation.
- Les variables locales sont typées et numérotées.
- Chaque instruction prend exactement un octet (IL_XXXX à gauche représente l'offset de l'instruction IL correspondante, par rapport au début de la méthode).

- `ldc.i4.5` (*load constant* 5/charge constante 5) est une instruction IL qui pousse la valeur constante entière 5 sur la pile, sous la forme d'un entier codé sur quatre octets (idem pour 6). Comprenez bien que l'entier 5 n'est pas un paramètre de cette instruction. En conséquence, l'instruction `ldc.i4.5` ne prend qu'un octet pour être stockée. À l'inverse, si l'on avait eu à pousser la valeur constante entière 12345678 stockée sur quatre octets sur la pile, on aurait utilisé l'instruction IL `ldc.i4`. Cette instruction IL prend en paramètre un entier codé sur quatre octets. Cette instruction avec son paramètre aurait alors pris cinq octets pour être stocké. On s'aperçoit donc, qu'à l'instar de ce qui se fait pour les langages machines, certaines instructions du langage IL ont été spécialement conçues pour optimiser le nombre d'octets utilisés pour stocker le code IL. De plus, avec cette pratique, la vitesse d'exécution est aussi optimisée, puisqu'on économise le temps de lecture du paramètre.
- `ldloc.N` (*load local*/charge locale) est une instruction IL qui pousse la valeur de la variable locale numéro N au sommet de la pile.
- `stloc.N` (*store local*/enregistre locale) est une instruction IL qui dépile la valeur au sommet de la pile et l'enregistre dans la variable locale numéro N.
- `add` (*add*/ajoute) est une instruction IL qui dépile les deux valeurs au sommet de la pile et les ajoute. Le résultat est alors stocké au sommet de la pile. Notez que l'autre instruction IL `add.ovf` teste le dépassement de valeur et lance, le cas échéant, une exception `OverflowException`. De plus, on ne peut combiner tous les types pour les valeurs d'entrés. Dans le cas où une combinaison de type n'est pas prévue, une exception est lancée.
- `ret` (return/retourne) est une instruction IL qui provoque le retour au code de la méthode appelante.

Plus généralement toutes les instructions IL, dont le nom commence par `ld`, chargent une valeur au sommet de la pile. À chacune de ces instructions IL est associée une instruction IL symétrique, dont le nom commence par `st`, qui dépile la valeur au sommet et l'enregistre.

## *Exemple 2 : les appels de méthodes et la pile*

La pile peut être vue comme un empilage de *fenêtres de pile* (*stack frames* en anglais). Chaque appel de méthode correspond à la création d'une fenêtre de pile sur le haut de la pile du thead courant. Chaque retour de méthode correspond à la destruction de la fenêtre de pile située sur le haut de la pile.

Le code C# de la méthode `Main()` suivante...

*Exemple 2-12 :*

```
class Program{
   static int f(int i1, int i2){
      return i1+i2 ;
   }
   public static void Main(){
      int i1 =5 ;
      int i2 =6 ;
      int i3 = f(i1,i2) ;
   }
}
```

...produit le code IL suivant pour la méthode `Main()` :

```
.maxstack  2
  .locals ([0] int32 i1,
           [1] int32 i2,
           [2] int32 i3)
  IL_0000:  ldc.i4.5
  IL_0001:  stloc.0
  IL_0002:  ldc.i4.6
  IL_0003:  stloc.1
  IL_0004:  ldloc.0
  IL_0005:  ldloc.1
  IL_0006:  call       int32 Program::f(int32,int32)
  IL_000b:  stloc.2
  IL_000c:  ret
```

On voit bien que, à l'instar de l'exemple précédent, les deux valeurs de `i1` et `i2` sont chargées sur la pile, avant l'appel de la méthode `Program.f()`, grâce à l'instruction IL `call` qui :

- dépile les arguments de l'appel et les stocke en mémoire ;
- crée une nouvelle fenêtre de pile en haut de la pile du thread ;
- effectue l'appel à la méthode `f()`.

Le code suivant est produit par le compilateur C# pour la méthode `f()` :

```
.maxstack  2
  .locals init ([0] int32 CS$00000003$00000000)
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  add
  IL_0003:  stloc.0
  IL_0004:  br.s       IL_0006
  IL_0006:  ldloc.0
  IL_0007:  ret
```

L'instruction IL `ldarg` sert à charger la valeur d'un argument sur la pile. Rappelez-vous que les valeurs des arguments ont été sauvées en mémoire par l'instruction `call`. Attention, les arguments sont indexés à partir de zéro dans une méthode statique (comme `f()`) et à partir de un dans une méthode non statique. En effet lors de l'appel d'une méthode non statique, l'argument indexé par zéro est réservé implicitement pour la référence `this`.

La valeur de retour de la méthode doit être la seule valeur présente dans la fenêtre de pile de la méthode, juste avant l'appel à l'instruction `ret`. L'instruction `ret` indique un retour de la méthode (donc la destruction de la fenêtre de pile qui lui est associée) mais garde la valeur de retour en haut de la pile de façon à ce que la méthode appelante puisse la récupérer.

Notez que le code suivant, qui ne fait pas intervenir de variables locales, aurait été valide aussi :

```
.maxstack  2
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  add
  IL_0003:  ret
```

## *Instructions IL de comparaison et de branchement*

Il existe beaucoup d'instructions IL de comparaison des deux valeurs au sommet de la pile. Leurs noms est, une fois de plus, emprunté aux langages machines existants : `ceq` (*compare val1 equal val2*), `cgt` (*compare val1 greater than val2*), `clt`(*compare val1 less than val2*).

Nous précisons que `val2` est au sommet de la pile, donc `val1` est juste en dessous. De plus `val1` et `val2` sont dépilées et une valeur entière est placée au sommet, 1 si la comparaison est vraie sinon 0.

Il existe une famille d'instructions IL pour le branchement. La plupart commencent par « `b` » ou « `br` » et prennent en paramètre l'offset localisant l'instruction IL, sur laquelle, l'unité d'exécution doit (éventuellement) se brancher. L'instruction de branchement inconditionnel s'appelle `br`. Toutes les autres instructions sont des branchements conditionnels, c'est-à-dire que le branchement ne se fait qu'à une condition. Par exemple l'instruction `brtrue` n'effectue le branchement qu'à la condition que la valeur au sommet de la pile soit non nulle. L'instruction `beq` n'effectue le branchement qu'à la condition que les deux valeurs au sommet de la pile soient égales. Les significations des instructions `brfalse` (Branch if False), `blt` (Branch if Lower Than), `ble` (Branch if Lower or Equal), `bgt` (Branch if Greater Than), `bge` (Branch if Greater or Equal), `bne` (Branch if Not Equal) ... découlent naturellement de tout ceci.

## *IL et la programmation orientée objet*

Analysons le code IL généré à partir d'un petit programme C# objet.

*Exemple 2-13 :*

```
class Program {
   public override string ToString() { return "Program m_i=" + m_i ; }
   int m_i = 9 ;
   Program(int i) { m_i = i ; }
   public static void Main() {
      object obj = new Program(12) ;
      obj.ToString() ;
   }
}
```

Voici le code IL de la méthode `Main()` :

```
.maxstack  2
  .locals init ([0] object obj)
  IL_0000:  ldc.i4.s   12
  IL_0002:  newobj     instance void Program::.ctor(int32)
  IL_0007:  stloc.0
  IL_0008:  ldloc.0
  IL_0009:  callvirt   instance string [mscorlib]System.Object::ToString()
  IL_000e:  ret
```

On voit qu'une instruction IL spéciale nommée `newobj`, crée un objet, appelle le constructeur de la classe et place la référence de l'objet sur la pile. On voit que l'appel à la méthode virtuelle `Program.ToString()` (qui redéfinie la méthode `object.ToString()`) se fait avec l'instruction IL `callvirt`. Avant cet appel, la référence de l'objet sur lequel la méthode est appelée, est placée sur la pile. Le polymorphisme est donc supporté par l'instruction IL `callvirt`.

Notez qu'en page 498 nous exposons les modifications principales du langage IL pour le support la généricité.

## *Les jetons de métadonnées*

Les tables des métadonnées de type sont souvent référencées par le code IL. Lorsqu'il désassemble le code IL, `ildasm.exe` met les noms et les signatures des méthodes directement dans le code IL désassemblé. En fait, au niveau binaire, le code IL n'a pas de chaînes de caractères définissant le nom des méthodes. Pour référencer une méthode, le code IL contient des valeurs de quatre octets qui pointent vers les tables de la section métadonnées de type. On appelle ces valeurs de quatre octets les *jetons de métadonnées* (*metadata token* en anglais).

Le premier octet d'un jeton de métadonnées référence la table de métadonnées. Les trois autres octets référencent un élément dans cette table. Par exemple la table référençant les membres utilisés par un module (la table *MethodRef*) ayant le numéro 10, un jeton de métadonnées vers le membre 5 de cette table est : 0x0A000005.

Les jetons de métadonnées peuvent être vus en choisissant l'option *Show token values* de `ildasm.exe`. Par exemple le code de la méthode `Main()` du programme suivant...

*Exemple 2-14 :*

```
using System ;
class Program{
   public static void Main(){
      Console.WriteLine( "Hello world!" ) ;
   }
}
```

...contient le code IL suivant : (vu avec la visualisation des jetons de métadonnées sous `ildasm.exe`) :

```
.method /*06000001*/ public hidebysig static
        void  Main() cil managed
{
  .entrypoint
  // Code size       11 (0xb)
  .maxstack  1
  IL_0000:  ldstr      "Hello world!" /* 70000001 */
  IL_0005:  call       void [mscorlib/* 23000001 */]
          System.Console/* 0100000F */::WriteLine(string) /* 0A00000E */
  IL_000a:  ret
} // end of method Program::Main
```

Les jetons de métadonnées vus dans cet exemple sont :

- 0x06000001 : référence l'entrée dans la table *MethodDef* (0x06) représentant la méthode `Main`(0x000001).
- 0x70000001 : référence l'entrée dans le tas *#userstring* (0x70) représentant la chaîne de caractères `"Hello world!"` (0x000001).
- 0x23000001 : référence l'entrée dans la table *AssemblyRef* (0x23) représentant l'assemblage `mscorlib` (0x000001).

- 0x0100000F : référence l'entrée dans la table *TypeRef* (0x01) représentant la classe `System.Console` (0x00000F).
- 0x0A00000E : référence l'entrée dans la table *MemberRef* (0x0A) représentant la méthode `WriteLine` (0x00000F). Notez qu'ici, le jeton de métadonnées de la méthode `WriteLine` n'est pas dans la table *MethodDef* car cette méthode est définie dans un autre module (et même, un autre assemblage).

# 3

# *Construction, configuration et déploiement des applications .NET*

## *Construire vos applications avec MSBuild*

La plateforme .NET 2.0 est livrée avec un nouvel outil nommé `msbuild.exe`. Cet outil sert à construire les applications .NET. Il accepte en entrée des fichiers XML qui décrivent l'enchaînement des tâches du processus de construction, un peu dans le même esprit que des fichiers `makefile`. D'ailleurs, au début du développement de ce projet, *Microsoft* l'avait baptisé `XMake`. L'exécutable `msbuild.exe` est situé dans le répertoire d'installation de .NET à savoir `[Rep_d'installation_de_Windows]\Microsoft.NET\Framework\v2.0.XXXX\`. Il est prévu que MSBuild fasse partie du système d'exploitation *Windows Vista*. Son rayon d'action augmentera alors et il pourra être utilisé pour construire tous types d'application.

Jusqu'ici, pour construire vos applications .NET, vous deviez :

- Soit utiliser la commande *Build* de *Visual Studio*.
- Soit utiliser l'exécutable de *Visual Studio* `devenv.exe` en ligne de commande.
- Soit avoir recours à une tierce technologie telle que l'outil *open-source Nant*, ou même, utiliser des fichiers *batch* qui appèlent le compilateur C# `csc.exe`.

MSBuild vise à unifier toutes ces techniques. Ceux qui connaissent Nant ne seront pas dépaysés car MSBuild reprend beaucoup de concepts de cet outil. L'atout majeur de MSBuild sur Nant est d'être exploité par *Visual Studio 2005*. MSBuild n'a aucune dépendance par rapport à *Visual Studio 2005* puisque, rappelons le, MSBuild fait partie intégrante de la plateforme .NET 2.0. En revanche, les fichiers d'extension `.proj` , `.csproj`, `.vbproj` etc générés par *Visual Studio 2005* pour construire les projets sont rédigés au format XML MSBuild. À la compilation, *Visual Studio 2005* utilise les services de MSBuild. En outre, le format XML MSBuild est pleinement supporté

et documenté. Le support de MSBuild est donc une évolution conséquente de *Visual Studio* qui jusqu'ici, utilisait des scripts de compilation non documentés.

# *MSBuild : Cibles, tâches, propriétés, items et conditions*

## *Fichier .proj, cibles et tâches*

L'élément racine de tous documents XML MSBuild est `<Project>`. Cet élément contient des éléments `<Target>`. Ces éléments `<Target>` constituent les unités de construction nommées cibles. Un script MSBuild peut contenir plusieurs cibles et `msbuild.exe` est capable d'enchaîner les exécutions de plusieurs cibles. Lorsque vous lancer `msbuild.exe` en ligne de commande, il prend en entrée le seul fichier d'extension `.proj` du répertoire courant. Si plusieurs fichiers `.proj` sont présents, il faut préciser en argument de ligne de commande le fichier `.proj` que `msbuild.exe` doit utiliser. Un seul fichier peut être précisé.

Une cible MSBuild est un ensemble de tâches MSBuild. Chaque élément enfant de l'élément `<Target>` constitue la définition d'une tâche. Les tâches d'une cible sont exécutées en série dans leur ordre de déclaration. Une quarantaine de types de tâches sont fournis par MSBuild comme par exemple :

| Type de tâche | Description |
|---|---|
| Copy | Copie des fichiers d'un répertoire source vers un répertoire destination. |
| MakeDir | Construit un répertoire. |
| Csc | Exploite le compilateur C# `csc.exe`. |
| Exec | Exécute une commande système. |
| AL | Exploite l'outil `al.exe` (*Assembly Linker*) |
| ResGen | Exploite l'outil `resgen.exe` (*Resources Generator*). |

La liste complète de ces types de tâches standard est disponible dans l'article **MSBuild Task Reference** des **MSDN**. Un aspect particulièrement intéressant de MSBuild est qu'un type de tâche est matérialisé par une classe .NET. Il est ainsi possible d'étendre MSBuild avec de nouveaux types de tâches en fournissant vos propres classes. Nous reviendrons sur ce point un peu plus loin.

Reprenons notre exemple d'assemblage multi modules de la page 20. Rappelons que pour construire cet assemblage constitué des trois modules `Foo1.exe`, `Foo2.netmodule` et `Image.jpg` nous avions exécutés les deux lignes de commande suivantes :

```
> csc.exe /target:module Foo2.cs
> csc.exe /Addmodule:Foo2.netmodule /LinkResource:Image.jpg  Foo1.cs
```

En plus, nous désirons ici qu'à l'issue de la construction de l'assemblage, les trois modules se trouvent dans un sous répertoire `\bin` du répertoire courant. Voici le script MSBuild `Foo.proj` capable de réaliser tout ceci :

*Exemple 3-1 :* *Foo.proj*

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
   <Target Name="FooCompilation">
      <MakeDir Directories= "bin"/>
      <Copy SourceFiles="Image.jpg" DestinationFiles=".\bin\Image.jpg"/>
      <Csc Sources="Foo2.cs"       TargetType="module"
           OutputAssembly=".\bin\Foo2.netmodule" />
      <Csc Sources="Foo1.cs"       TargetType="exe"
           AddModules=".\bin\Foo2.netmodule" LinkResources="Image.jpg"
           OutputAssembly=".\bin\Foo1.exe" />
   </Target>
</Project>
```

On voit que la cible nommée `FooCompilation` est constituée de quatre tâches :

- Une tâche de type `MakeDir` qui construit le répertoire `\bin`.
- Une tâche de type `Copy` qui copie le fichier `Image.jpg` dans le répertoire `\bin` ;
- Les deux tâches de type `Csc` qui invoquent chacune le compilateur `csc.exe`.

Pour exécuter ce script de compilation, il faut constituer un répertoire ayant le contenu suivant :

```
.\Foo.proj
.\Foo1.cs
.\Foo2.cs
.\Image.jpg
```

Allez dans ce répertoire avec la fenêtre de commande *SDK Command Prompt* (Menu *Démarrer* ▶ *Microsoft .NET Framework SDK v2.0* ▶ *SDK Command Prompt*) puis lancer la commande `>msbuild.exe`. Chaque cible est obligatoirement nommée. Par défaut `msbuild.exe` exécute seulement la première cible. Vous pouvez spécifier une liste de noms de cibles séparés par des points virgules en ligne de commande avec l'option `/target` (raccourcis `/t`). Vous pouvez aussi spécifier une telle liste avec l'attribut `DefaultTarget` de l'élément `<Project>`. Si plusieurs cibles sont spécifiées, l'ordre d'exécution n'est pas défini.

Le comportement par défaut de MSBuild est de stopper la construction dés qu'une tâche émet une erreur. Vous pouvez souhaiter avoir un script de construction tolérant aux erreurs. Aussi, chaque élément représentant une tâche peut contenir un attribut `ContinueOnError` qui est positionné à `false` par défaut mais qui peut être positionné à `true`.

## *Notion de propriété*

Pour vous permettre de paramétrer vos scripts, MSBuild présente la notion de propriété. Une propriété est un couple clé/valeur défini dans un élément `<PropertyGroup>`. Les propriétés MSBuild fonctionnent comme un système d'alias. Chaque occurrence de `$(clé)` dans le script est remplacée par la valeur associée Typiquement, le nom du répertoire `/bin` est utilisé à cinq reprises dans notre script `Foo.proj`. Il constitue un bon candidat pour définir une propriété :

*Exemple 3-2 :* *Foo.proj*

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
   <PropertyGroup>
      <OutputPath>.\bin</OutputPath>
   </PropertyGroup>
   <Target Name="FooCompilation">
      <MakeDir Directories= "$(OutputPath)"/>
      <Copy SourceFiles="Image.jpg"
            DestinationFiles="$(OutputPath)\Image.jpg"/>
      <Csc Sources="Foo2.cs"        TargetType="module"
           OutputAssembly="$(OutputPath)\Foo2.netmodule" />
      <Csc Sources="Foo1.cs"        TargetType="exe"
           AddModules="$(OutputPath)\Foo2.netmodule"
           LinkResources="Image.jpg"
           OutputAssembly="$(OutputPath)\Foo1.exe" />
   </Target>
</Project>
```

Vous pouvez en outre exploiter des propriétés définies par défaut par MSBuild telles que :

| Type de tâche | Description |
|---|---|
| `MSBuildProjectDirectory` | Répertoire qui stocke le script MSBuild courant. |
| `MSBuildProjetFile` | Nom du fichier script MSBuild courant. |
| `MSBuildProjectExtension` | Extension du fichier script MSBuild courant. |
| `MSBuildProjectFullPath` | Chemin complet du fichier script MSBuild courant. |
| `MSBuildProjectName` | Nom du fichier script MSBuild courant sans l'extension. |
| `MSBuildPath` | Répertoire qui stocke le fichier `msbuild.exe`. |

Lors de l'édition des propriétés avec *Visual Studio 2005*, vous vous apercevrez qu'un certains nombre de clés vous sont proposées par l'intellisense. `OutputPath` constitue une telle clé. Vous pouvez utiliser ces clés mais rien ne vous empêche de définir vos propres clés. Nous aurions ainsi pu choisir pour clé `RepDeSortie` à la place de `OutputPath`.

## *Notion d'item*

La base de la construction d'un projet par un script est la manipulation de répertoires, de fichiers (sources, ressources, exécutables etc) et de références (vers des assemblages, vers des services, vers des classes COM, vers des fichiers ressources, vers des projets etc). On utilise le terme *item* pour désigner ces entités qui constituent les entrées et les sorties de la plupart des tâches. Dans notre exemple, le fichier `Image.jpg` est un item consommé à la fois par la tâche `Copy` et par la seconde tâche `Csc`. Le fichier `Foo2.netmodule` est un item produit par la première tâche `Csc` et consommé par la seconde tâche `Csc`. Réécrivons notre script avec cette notion d'*item* :

*Exemple 3-3 :* *Foo.proj*

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
   <PropertyGroup><OutputPath>.\bin</OutputPath></PropertyGroup>
   <ItemGroup>
      <Fichier_Image Include="$(OutputPath)\Image.jpg"/>
      <NetModule_Foo2 Include="$(OutputPath)\Foo2.netmodule"/>
   </ItemGroup>
   <Target Name="FooCompilation">
      <MakeDir Directories= "$(OutputPath)"/>
      <Copy SourceFiles="Image.jpg"
            DestinationFiles="@(Fichier_Image)"/>
      <Csc Sources="Foo2.cs"        TargetType="module"
           OutputAssembly="@(NetModule_Foo2)" />
      <Csc Sources="Foo1.cs"        TargetType="exe"
           AddModules="@(NetModule_Foo2)"
           LinkResources="@(Fichier_Image)"
           OutputAssembly="$(OutputPath)\Foo1.exe" />
   </Target>
</Project>
```

Nous remarquons que l'on utilise la syntaxe `@(nom de l'item)` pour se référer à un item. En outre un *item* peut définir un ensemble de fichier grâce à la syntaxe *wildcard*. Par exemple, l'*item* suivant fait référence à tous les fichiers sources C# du répertoire courant sauf `Foo1.cs` :

```
<cs_source Include=".\*.cs" Exclude=".\Foo1.cs" />
```

## *Poser des conditions*

On peut souhaiter qu'un même script MSBuild se décline sous plusieurs versions. Par exemple, il serait dommage de devoir écrire et maintenir deux scripts pour chaque projet, un pour la construction en mode `Debug` et un pour la construction en mode `Release`. Aussi, MSBuild introduit la notion de condition. Un attribut `Condition` peut être appliqué à pratiquement n'importe quel élément d'un script MSBuild (propriétés, *item*, cible, tâche, groupe de propriétés, groupe d'*items* etc). Si à l'exécution la condition d'un élément n'est pas réalisée, le moteur MSBuild l'ignorera. L'article **MSBuild Conditions** des **MSDN** décrit la liste des expressions de condition qui peuvent être précisées.

Dans l'exemple suivant, nous exploitons les conditions de type *test de l'égalité de deux chaînes de caractères* pour faire en sorte que notre script exemple supporte le mode `Debug` et `Release`. Nous utilisons aussi une condition de type *test de l'existence d'un fichier ou d'une répertoire* pour n'exécuter la tâche `MakeDir` que si le répertoire à créer n'existe pas. Cette condition n'est là que pour des besoins pédagogiques car la tâche `MakeDir` ne s'exécute pas si le répertoire à créer existe déjà :

*Exemple 3-4 :* *Foo.proj*

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
   <PropertyGroup Condition="'$(Configuration)'=='Debug'">
      <Optimize>false</Optimize>
      <DebugSymbols>true</DebugSymbols>
```

```
      <OutputPath>.\bin\Debug</OutputPath>
   </PropertyGroup>
   <PropertyGroup Condition="'$(Configuration)'=='Release'">
      <Optimize>true</Optimize>
      <DebugSymbols>false</DebugSymbols>
      <OutputPath>.\bin\Release</OutputPath>
   </PropertyGroup>
   <ItemGroup>
...
   <Target Name="FooCompilation">
      <MakeDir Directories= "$(OutputPath)"
               Condition="!Exists('$(OutputPath)')"/>
...
```

Lorsqu'elles sont définies, les propriétés standard `Optimize` et `DebugSymbols` sont automatiquement prises en comptes par les tâches de type `Csc`.

Avant de lancer ce script, il faut préciser comme argument en ligne de commande la valeur de la propriété `Configuration`. Cela peut se faire avec l'option `/property` (raccourcis `/p`) :

```
>msbuild /p:Configuration=Release
```

Avec un peu d'astuce, il est possible d'utiliser une condition pour définir la valeur par défaut de la propriété `Condition` :

*Exemple 3-5 :* *Foo.proj*

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
   <PropertyGroup Condition="'$(Configuration)'==''">
      <Configuration>Debug</Configuration>
   </PropertyGroup>
   <PropertyGroup Condition="'$(Configuration)'=='Debug'">
...
```

# *Concepts avancés de MSBuild*

## *Construction incrémentale et dépendances entre cibles*

Dans un environnement réel, l'exécution d'un script MSBuild peut prendre plusieurs minutes voire plusieurs heures pour s'exécuter. Pour l'anecdote, sachez que depuis ses débuts le système d'exploitation *Windows* met environs 12 heures à se compiler. Cela signifie que le volume grandissant de code à compiler vient compenser la montée en puissance des machines.

Il n'est pas souhaitable de relancer complètement une construction pour un changement mineur effectué dans un fichier source sur lequel aucun autre composant ne dépend. Aussi, vous pouvez utiliser la notion de *construction incrémentale*. Pour cela vous devez préciser la liste des *items* en entrée et des *items* en sortie d'une cible au moyen des attributs `Inputs` et `Outputs`. Si MSBuild détecte qu'au moins un *item* en entrée est plus vieux qu'au moins un *item* en sortie, il prend la décision d'exécuter la cible.

Cette technique de construction incrémentale vous oblige à partitionner l'ensemble de vos tâches en plusieurs cibles. Nous avons vu que si vous précisez plusieurs cibles à exécuter à

MSBuild, par exemple avec l'attribut `DefaultTargets`, vous ne pouvez présumer d'aucun ordre d'exécution. Vous pouvez cependant définir un système de dépendance entre cibles avec l'attribut `DependsOnTargets`. MSBuild n'exécute une cible que lorsque l'ensemble des cibles sur lesquelles elle dépend a été exécuté. Naturellement, le moteur de MSBuild détecte et sanctionne d'une erreur les dépendances circulaires entre cibles :

*Exemple 3-6 :* *Foo.proj*

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
   DefaultTargets="FooCompilation">
...
   <Target Name="CréeOutputPath" Condition="!Exists('$(OutputPath)')">
      <MakeDir Directories= "$(OutputPath)"/>
   </Target>
   <Target Name="FooCompilation" DependsOnTargets="CréeOutputPath"
           Inputs="Foo2.cs;Foo1.cs"
           Outputs="@(NetModule_Foo2);$(OutputPath)\Foo1.exe">
      ...
   </Target>
</Project>
```

## *Transformations MSBuild*

Vous avez la possibilité d'établir une correspondance biunivoque entre l'ensemble des *items* en entrée et l'ensemble des *items* en sortie d'une cible. Pour cela, vous devez utiliser les transformations MSBuild détaillées dans l'article **MSBuild Transforms** des **MSDN**. L'avantage d'utiliser des transformations est que MSBuild ne décide d'exécuter la cible que si au moins un *item* en entrée est plus vieux que l'*item* en sortie qui lui correspond. Logiquement, une telle cible est moins souvent exécutée d'où un gain de temps.

## *Fractionner un script MSBuild sur plusieurs fichiers*

Nous avons vu que l'outil `msbuild.exe` ne peut traiter plus d'un fichier script à chaque exécution. Cependant, un fichier script MSBuild peut importer un autre fichier script MSBuild au moyen de l'élément `<Import>`. Dans ce cas, tous les éléments enfants de l'élément racine `<Project>` du fichier importé sont copiés en lieu et place de l'élément `<Import>` responsable de l'importation. Notre script exemple peut ainsi être fractionné sur deux fichiers `Foo.proj` et `Foo.target.proj` comme ceci :

*Exemple 3-7 :* *Foo.proj*

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
   DefaultTargets="FooCompilation">
   <PropertyGroup Condition="'$(Configuration)'==''"> ...
   <PropertyGroup Condition="'$(Configuration)'=='Debug'"> ...
   <PropertyGroup Condition="'$(Configuration)'=='Release'"> ...
   <ItemGroup> ...
   <Import Project="Foo.target.proj"/>
</Project>
```

*Exemple 3-8 :* *Foo.target.proj*

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >
   <Target Name="CréeOutputPath"   ...
   <Target Name="FooCompilation"   ...
</Project>
```

## *Intégration de MSBuild avec Visual Studio*

Nous avons déjà précisé que les fichiers d'extension `.proj`, `.csproj`, `.vbproj` etc générés par *Visual Studio 2005* pour construire les projets sont rédigés au format XML MSBuild. Si vous analysez un tel fichier, vous vous apercevrez qu'aucune cible n'y est explicitement définie. En fait, les fichiers scripts générés par *Visual Studio 2005* importent des fichiers d'extensions `.targets` qui contiennent des cibles génériques. Par exemple, un fichier d'extension `.csproj` contient l'élément suivant :

```
<Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
```

Ce fichier `Microsoft.CSharp.targets` contient deux cibles génériques :

- Une cible nommée `CreateManifestResourceNames` qui s'occupe de l'organisation des fichiers ressources (transformation des fichiers `.resx` en `.resources` etc).
- Une cible `CoreCompile` qui contient une tâche `Csc` qui réalise effectivement la compilation.

Nous vous invitons à consulter ces fichiers `.targets` situés dans le répertoire `$(MSBuildBinPath)` qui est le répertoire d'installation de .NET 2.0 (i.e `[Rep d'installation de Windows]\Microsoft.NET\Framework\v2.0.XXXXX`).

En plus d'importer un tel fichier `.targets`, les fichiers générés par *Visual Studio 2005* contiennent essentiellement les définitions des propriétés et items qui servent à paramétrer les cibles génériques.

## *Créer vos propres tâches MSBuild*

Un aspect particulièrement intéressant de MSBuild est que chaque type de tâche est matérialisé par une classe .NET. Il est ainsi possible d'étendre MSBuild avec de nouveaux types de tâches en fournissant vos propres classes. Une telle classe doit supporter les contraintes suivantes :

- Elle doit implémenter l'interface `ITask` définie dans l'assemblage `Microsoft.Build.Framework.dll` ou dériver de la classe *helper* `Task` définie dans l'assemblage `Microsoft.Build.Utilities.dll`.
- Elle doit implémenter la méthode `bool Execute()` de l'interface `ITask`. Cette méthode contient le corps de la tâche et doit retourner `true` si elle a été exécutée avec succès.
- Elle peut présenter des propriétés dont les valeurs seront positionnées par MSBuild à partir des valeurs des attributs de la tâche, avant l'appel à la méthode `Execute()`. Seules les propriété marquées avec l'attribut `Required` doivent être obligatoirement initialisées.

Voici en exemple le code d'une tâche nommée `MyTouch` qui met à jour la date des fichiers spécifiés par l'attribut `Files` (nous précisons qu'une telle tâche nommée `Touch` est présentée par le *framework* MSBuild) :

*Exemple 3-9 :* *MyTask.cs*

```
using System ;
using Microsoft.Build.Framework ;
using Microsoft.Build.Utilities ;
namespace MyTask {
   public class MyTouch : Task {
      public override bool Execute() {
         DateTime now = DateTime.Now ;
         Log.LogMessage(now.ToString() +
            " est la nouvelle date des fichiers suivants:") ;
         try {
            foreach(string fileName in m_FilesNames) {
               Log.LogMessage("   " + fileName) ;
               System.IO.File.SetLastWriteTime(fileName, now) ;
            }
         }
         catch (Exception ex) {
            Log.LogErrorFromException(ex, true) ;
            return false ;
         }
         return true ;
      }
      [Required]
      public string[] Files {
         get { return (m_FilesNames) ; } set { m_FilesNames = value ; }
      }
      private string[] m_FilesNames ;
   }
}
```

Notez l'utilisation de la propriété `TaskLoggingHelper Task.Log{get;}` qui permet d'afficher des informations concernant le déroulement de la tâche.

Notre tâche `MyTouch` doit être enregistrée auprès de tous les scripts MSBuild qui sont susceptibles d'y avoir recours. Cela se fait au moyen d'un élément `<UsingTask>`. Voici un tel script qui met à jour les dates des fichiers sources C# du répertoire courant (l'assemblage `MyTask.dll` doit être situé dans le répertoire `C:\CustomTasks\`) :

*Exemple 3-10 :* *Foo.proj*

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
   <UsingTask AssemblyFile="C:\CustomTasks\MyTask.dll"
              TaskName="MyTask.MyTouch"/>
   <ItemGroup>
      <FichierSrcCs Include="*.cs"/>
   </ItemGroup>
   <Target Name="ToucheLesFichiersSrcCs" >
      <MyTouch Files= "@(FichierSrcCs)"/>
   </Target>
</Project>
```

Il est intéressant de noter que toutes les tâches standard sont déclarées par des éléments <UsingTask> dans le fichier Microsoft.Common.Tasks. Ce fichier est automatiquement et implicitement importé par msbuild.exe à chaque exécution. En analysant ce fichier, on voit que les classes correspondantes aux tâches standard sont définies dans l'assemblage Microsoft.Build.Tasks.dll. Vous pouvez ainsi avoir accès au code de ces tâches en utilisant un outil tel que *Reflector*.

# *Fichiers de configuration*

Un assemblage exécutable .NET a la possibilité d'avoir un fichier de configuration XML qui peut être modifié après l'installation. Le nom d'un tel fichier de configuration doit obligatoirement commencer par le nom du module contenant le manifeste (extension .exe comprise) auquel on ajoute l'extension .config (par exemple Foo.exe.config). Il doit être placé impérativement dans le même répertoire que l'assemblage.

*Visual Studio* présente des facilités pour l'édition et la maintenance du fichier de configuration d'une application. Vous pouvez cliquer droit sur le projet qui définit un assemblage exécutable ► *Add* ► *New Item...* ► *Application Configuration File* ► Gardez le nom App.Config. À la compilation, *Visual Studio* effectuera une copie du contenu de ce fichier dans un fichier nommé [Nom de l'assemblage exécutable avec extension].config situé dans le même répertoire que l'assemblage exécutable produit.

## *Le fichier machine.config*

La plupart des éléments de configuration d'une application .NET peuvent être déclarés dans un fichier nommé machine.config situé dans le sous répertoire /CONFIG du répertoire d'installation de .NET ([Rep_d'installation_de_Windows]\Microsoft.NET\Framework\v2.0.XXXXX). La valeur d'un paramètre défini dans le fichier machine.config n'est prise en compte par une application .NET que si ce paramètre n'est pas initialisé dans son fichier de configuration. En outre, certains paramètres tels que le modèle de processus utilisé par ASP.NET n'ont de sens qu'au niveau de la machine. Ils ne peuvent ainsi qu'être initialisés dans le fichier machine.config.

Il n'est pas conseillé de modifier le fichier de configuration de la machine. En effet, vous pouvez altérer par inadvertance le comportement des applications installées sur votre machine. Cependant, dans le cas d'une machine en production (un serveur ou une machine dédiée à une application) ce fichier peut constituer un moyen efficace pour définir une stratégie de configuration globale à la machine.

## *Les paramètres de configuration standard*

Tout fichier de configuration .NET admet un élément racine <configuration> :

```
<configuration
  xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
```

Voici les principaux sous éléments standard de l'élément <configuration> :

- <appSettings> : Contient les éléments de configuration propres à l'application.
- <configSections> : Permet de définir les sections de configuration que nous présentons un peu plus loin.

- `<connectionStrings>` : Stocke les chaînes de connexion aux bases de données (voir page 717).
- `<location>` : Permet de définir les éléments de configuration ASP.NET pour chaque page web (voir page 890).
- `<mscorlib>` : (seulement `machine.config`) Contient un élément `<cryptographySettings>` qui contient les paramètres des mécanismes de cryptographie (voir page 219).
- `<protectedData>` : Permet de définir des sections d'information confidentielle dans un fichier de configuration (voir page 718).
- `<runtime>` : Contient une section `<gcConcurrent>` qui permet de déterminer si le ramasse miettes s'exécute d'une manière concurrente (voir page 120). Contient aussi une section `<assemblyBinding>` qui positionne les paramètres de l'algorithme de localisation des assemblages (voir page 103).
- `<system.Data.[Founisseur de données]>` : Permet de paramétrer les fournisseurs de données ADO.NET disponibles avec notamment des fichiers XML consommés par le *framework* de schéma (voir page 718).
- `<system.Diagnostics>` : Contient des informations relatives aux traces (voir page 620).
- `<system.runtime.remoting>` : Permet de configurer les services et canaux .NET Remoting (voir page 808).
- `<system.transactions>` : Permet de paramétrer les transactions (voir page 751).
- `<system.web>` : Définit les paramètres utilisés par ASP.NET. La façon de configurer les applications ASP.NET est un vaste sujet auquel nous consacrons la section 23 « Configuration d'une application ASP.NET », page 888.

## *Définir vos propres paramètres de configuration avec l'élément <appSettings>*

Vous pouvez vous servir de l'élément `<appSettings>` du fichier de configuration d'une application pour stocker les valeurs des paramètres qui lui sont propres. Par exemple, supposons que l'application `MyApp` ait un paramètre entier nommé `MyEntier` :

*Exemple 3-11 :* *`MyApp.exe.config`*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration
   xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <appSettings>
      <add key="MyEntier" value="1234"/>
   </appSettings>
</configuration>
```

Voici le code de l'application `MyApp`. À chaque exécution, ce code obtient la valeur du paramètre `MyEntier`, la multiplie par 10, puis sauvegarde la nouvelle valeur :

*Exemple 3-12 :* *MyApp.cs*

```
using System.Configuration ;
class Program {
   static void Main() {
      Configuration appCfg = ConfigurationManager.OpenExeConfiguration(
                                    ConfigurationUserLevel.None) ;
      AppSettingsSection appSettings = appCfg.AppSettings ;
      int myEntier ;
      if (int.TryParse( appSettings.Settings["MyEntier"].Value,
                        out myEntier)) {
         System.Console.WriteLine(myEntier) ;
         myEntier *= 10 ;
         appSettings.Settings["MyEntier"].Value = myEntier.ToString() ;
         appCfg.Save() ;
      }
   }
}
```

Si vous recompilez l'application `MyApp` avec *Visual Studio* et que vous utilisez le fichier `App.Config`, la valeur de `MyEntier` est réinitialisée à 1234 puisque le contenu du fichier de configuration `MyApp.exe.config` est écrasé par le contenu du fichier `App.Config`.

Il est clair que cette façon de procéder présente deux désavantages majeurs :

- La valeur d'un paramètre n'est pas typée. Il a fallu explicitement parser une chaine de caractère pour obtenir la valeur du paramètre `MyEntier` dans une variable entière.
- Le nom du paramètre n'est pas vérifié par le compilateur puisqu'il est fourni sous forme d'une chaîne de caractères. Cela nuit à la productivité des développeurs qui ne bénéficient pas non plus de l'intellisense.

## *Définir vos propres paramètres avec des sections de configuration*

Pour éviter ces deux problèmes, vous pouvez avoir recours à la notion de *section de configuration*. Une telle section est un ensemble de paramètres de configuration. Chaque paramètre est défini soit au niveau de l'application, soit au niveau de l'utilisateur *Windows* qui exécute l'application. Ainsi, vous pouvez vous servir simplement de ce mécanisme pour stocker les préférences de chaque utilisateur.

Le fichier de configuration suivant défini une section de configuration nommée `MySettings` qui contient un paramètre de configuration utilisateur nommé `MyEntierUsr` et un paramètre de configuration de l'application nommé `MyEntierApp` :

*Exemple 3-13 :* *MyApp.exe.config*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration
    xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <configSections>
      <sectionGroup name="userSettings"
                    type="System.Configuration.UserSettingsGroup,
                          System, Version=2.0.0.0, Culture=neutral,
```

```
                        PublicKeyToken=b77a5c561934e089" >
         <section   name="MySettings"
                    type="System.Configuration.ClientSettingsSection,
                          System, Version=2.0.0.0, Culture=neutral,
                          PublicKeyToken=b77a5c561934e089"
                    allowExeDefinition="MachineToLocalUser" />
      </sectionGroup>
      <sectionGroup name="applicationSettings"
                    type="System.Configuration.ApplicationSettingsGroup,
                          System, Version=2.0.0.0, Culture=neutral,
                          PublicKeyToken=b77a5c561934e089" >
         <section   name="MySettings"
                    type="System.Configuration.ClientSettingsSection,
                          System, Version=2.0.0.0, Culture=neutral,
                          PublicKeyToken=b77a5c561934e089" />
      </sectionGroup>
   </configSections>
   <userSettings>
      <MySettings>
         <setting name="MyEntierUsr" serializeAs="String">
            <value>1234</value>
         </setting>
      </MySettings>
   </userSettings>
   <applicationSettings>
      <MySettings>
         <setting name="MyEntierApp" serializeAs="String">
            <value>4321</value>
         </setting>
      </MySettings>
   </applicationSettings>
</configuration>
```

Voici le code de l'application `MyApp` correspondante. On voit que les deux problèmes de l'exemple précédent sont résolus grâce à l'introduction d'une classe nommée `MySettings` dérivée de la classe `System.Configuration.ApplicationSettingsBase`. Cette classe présente une propriété pour chaque paramètre. Les propriétés relatives aux paramètres utilisateurs sont marquées avec l'attribut `UserScopedSettingAttribute` tandis que les propriétés relatives aux paramètres de l'application sont marquées avec l'attribut `ApplicationScopedSettingAttribute`. On remarque que la propriété `MyEntierApp` n'a pas d'accesseur `set`. En effet, dans le cadre de cette technique les paramètres de l'application doivent être accessible en lecture seule :

*Exemple 3-14 :* *MyApp.cs*

```
using System.Configuration ;
class Program {
   static void Main() {
      MySettings mySettings = new MySettings() ;
      int myEntierUsr = mySettings.MyEntierUsr ;
      System.Console.WriteLine(myEntierUsr) ;
```

```
      myEntierUsr *= 10 ;
      mySettings.MyEntierUsr = myEntierUsr ;
      mySettings.Save() ;
      System.Console.WriteLine(mySettings.MyEntierApp) ;
   }
}
class MySettings : ApplicationSettingsBase {
   [UserScopedSetting()]
   public int MyEntierUsr {
      get { return (int)(this["MyEntierUsr"]) ; }
      set { this["MyEntierUsr"] = value ; }
   }
   [ApplicationScopedSetting()]
   public int MyEntierApp {
      get { return (int)(this["MyEntierApp"]) ; }
   }
}
```

À l'exécution, le CLR se sert de la réflexion pour :

- Mapper le nom des classes dérivées de la classe `ApplicationSettingsBase` avec les noms des sections de configuration.
- Mapper le nom des propriétés de ces classes avec les paramètres de configuration.

## *Créer vos sections de configuration avec Visual Studio 2005*

Malgré les avantages à utiliser les sections de configuration, vous pouvez être rebuté par la fait de créer une classe dédiée et par le nombre d'éléments ajoutés dans votre fichier de configuration. Heureusement, *Visual Studio 2005* présente un éditeur de configuration qui gère complètement cette charge supplémentaire.

Pour ajouter une nouvelle section il vous suffit de faire : Click droit sur le projet qui va être paramétré par la section ► *Add* ► *New Items...* ► *Settings File* ► Fournir le nom `XXX` de la section ► *OK*. Un nouvel élément `XXX.settings` est ajouté au projet. Si vous sélectionné cet élément, l'éditeur de paramètres de configuration apparaît. Pour chaque paramètre, vous pouvez choisir son nom, son type, s'il est un paramètre utilisateur ou un paramètre de l'application et une valeur par défaut.

Un fichier source `XXX.Designer.cs` est associé à l'élément `XXX.settings`. C'est un fichier généré qui contient la définition de la classe nommée `XXX` qui dérive de `ApplicationSettingBase`. Cette définition est constamment synchronisée avec les modifications effectuées dans l'éditeur. Aussi, comme dans tous fichier généré il ne faut pas y inséré votre propre code.

## *Remarques sur les sections de configuration*

En page 718 nous expliquons comment se servir de l'API de protection des données par encryption pour sauvegarder des paramètres de configuration sous une forme encryptée.

Vous avez la possibilité de valider les modifications apportées aux paramètres de configuration avant leur sauvegarde effective. Ceci est expliqué dans l'article **How to : Validate Application Settings** des **MSDN**.

Il est intéressant d'analyser votre fichier `machine.config` pour s'apercevoir que toutes les sections standard présentées un peu plus haut sont déclarées par des éléments `<section>`. À chacune de ces sections correspond un type standard tel que `ConnectionStringsSection`, `AppSettingsSection` ou `ProtectedConfigurationSection`. L'Exemple 3-12 montre comment avoir accès à une instance de `AppSettingsSection` afin de modifier le contenu de la section `<appSettings>`. Remarquez que certains de ces types tels que `SystemDiagnosticsSection` ne doivent être utilisés que par le CLR. Aussi, ils ne vous sont pas accessibles.

Les valeurs des paramètres utilisateurs ne sont pas stockées dans le fichier de configuration de l'application mais dans un fichier nommé `[nom de l'utilisateur].config`. Ce fichier est stocké dans le répertoire spécifié par la propriété statique `System.Windows.Forms.Application.LocalUserAppDataPath`.

Dans le cas d'une application déployée avec la technologie *ClickOnce* (présentée dans la suite du présent chapitre), les fichiers de configuration (de l'application et des utilisateurs) sont déployés dans le répertoire *ClickOnce* de l'application (voir page 83).

Par défaut, les classes dérivées de `ApplicationSettingBase` utilisent en interne la classe `System.Configuration.LocalFileSettingsProvider` pour avoir accès en lecture et en écriture aux fichiers de configuration. Cette classe dérive de la classe `System.Configuration.SettingsProvider`. Vous pouvez construire vos propres classes dérivées de `SettingsProvider` afin d'implémenter vos propres mécanisme de persistance des paramètres d'une application (dans une base de données, dans la base des registres, par l'intermédiaire d'un service web etc). Il suffit ensuite d'établir le lien entre vos classes dérivées de `ApplicationSettingBase` et vos classes dérivées de `SettingsProvider` en marquant les premières avec des attributs `SettingsProviderAttribute`.

Le contrôle `ToolStrip` du *framework Windows Form 2.0* présente des facilités pour stocker directement son état dans les paramètres de l'application. En outre, vous pouvez prévoir le même genre de facilité pour vos propres types de contrôles. Ceci fait l'objet de l'article **Application Settings for Custom Controls** des **MSDN**.

# *Déploiement des assemblages : XCopy vs. Répertoire GAC*

.NET propose deux types de stratégies pour déployer les assemblages d'une application.

- La stratégie XCopy (qualifiée aussi de stratégie de déploiement d'assemblage à titre privé) consiste simplement à copier les fichiers de l'application dans un répertoire. Parce qu'on ne peut pas faire plus simple, c'est la stratégie à choisir par défaut.
- Une autre stratégie doit être utilisée pour déployer les assemblages qui sont partagés par plusieurs applications. Cette stratégie présente des fonctionnalités très intéressantes tel qu'une gestion performante du *versionning*.

## *Déploiement XCopy*

La stratégie de déploiement d'assemblages *XCopy* est la plus simple qu'il puisse exister. Elle consiste à copier tous les assemblages concernant l'application à déployer, dans un même répertoire. En général les développeurs créent une arborescence de répertoires. Ni la base des registres ni les 'Active Directory' de *Windows* ne sont sollicités lors d'un déploiement de type XCopy.

Pour ne perdre aucun fichier, il est conseillé d'encapsuler l'arborescence et ses fichiers dans un seul fichier (par exemple une archive *cab*, *zip* ou *rar*, comme expliqué plus loin). La désinstallation de l'application consiste à détruire l'arborescence du disque dur. Le changement de version consiste à détruire l'arborescence du disque dur et à la remplacer par la nouvelle.

Ceux qui on déjà eu à gérer un déploiement sous *Windows*, que ce soit en tant que développeur, qu'administrateur ou en tant que simple utilisateur, mesurent le gain d'effort apporté par les déploiements de type XCopy.

Pour accéder à des fonctionnalités avancées, telles que la création de raccourcis bureaux ou d'icones dans la barre des tâches, vous pouvez utiliser le service d'installation de programmes *Windows* nommés MSI, qui fait l'objet d'une section un peu plus loin.

## *Assemblages partagés et répertoire GAC*

Le déploiement d'assemblages XCopy ne convient pas aux assemblages qui sont partagés par plusieurs applications installées sur la même machine. En effet, si l'on utilisait cette stratégie pour un *assemblage partagé*, il serait présent sur le disque dur autant de fois qu'il y a d'applications qui l'utilisent. Hormis le gaspillage d'espace, cette stratégie a pour principal défaut d'obliger à changer plusieurs fichiers en plusieurs endroits lorsque l'assemblage partagé évolue. La stratégie à appliquer pour les assemblages partagés consiste à utiliser un répertoire spécialement conçu pour les stocker. Ce répertoire s'appelle le *cache global des assemblages*. Il est aussi nommé répertoire *GAC* (*Global Assembly Cache* en anglais). Le répertoire GAC est en général le répertoire : « `C:\Windows\Assembly\GAC` ». Ce répertoire spécial est connu du CLR et par conséquent, il est connu à l'exécution de toutes les applications .NET.

> Comprenez bien que la grande majorité des assemblages ne sont pas partagés entre plusieurs applications. Vous rencontrerez rarement la nécessité de stocker un assemblage dans le répertoire GAC.

Un assemblage doit avoir un nom fort pour pouvoir être stocké dans le GAC. La notion de nom fort est exposée page 28. Un assemblage sans nom fort ne peut pas être stocké dans le GAC. En revanche un assemblage à nom fort peut ne pas être stocké dans le GAC (i.e déployer avec la stratégie de déploiement XCopy).

Chaque fois qu'un assemblage à nom fort est chargé dans un processus, la vérification de la validité de sa signature numérique doit être faite. Cependant, pour les assemblages partagés du répertoire GAC, ce n'est pas nécessaire. En effet, lorsqu'un assemblage partagé est inséré dans le répertoire GAC, la validité de sa signature numérique est automatiquement vérifiée. Si la signature numérique n'est pas valide, l'assemblage ne peut être inséré dans le répertoire GAC. Il y a donc un gain de performance lorsqu'un assemblage est chargé à partir du répertoire GAC.

Il est intéressant de remarquer que lorsque vous installez la plateforme d'exécution .NET sur une machine, les assemblages contenant les classes du *framework* .NET (`System.dll`, `System.Data.dll`, `System.Security.dll` etc) sont installés dans le GAC. Une copie de chacun de ces assemblages existe dans le répertoire d'installation de .NET (i.e `[Rep d'installation de Windows]\Microsoft.NET\framework\v2.0.XXXXX\`). Si vous installez *Visual Studio*, il aura recours à ces copies lors de l'analyse des références vers les assemblages standard du *framework* d'un projet. En effet, *Visual Studio* ne sait pas référencer les assemblages stockés dans le GAC. Seul le CLR sait exploiter les assemblages du GAC grâce à une couche de code nommée *assembly loader* qui fait l'objet de la section page 103.

## *Le stockage côte à côte dans le GAC résout l'enfer des DLLs*

Si on le considère comme un répertoire *Windows*, le répertoire GAC a une structure interne élaborée dont vous n'avez heureusement pas à vous préoccuper. Cette structure est nécessaire du fait que le GAC peut contenir plusieurs versions d'un même assemblage. Cette particularité s'appelle le stockage *côte à côte* (*side by side* en anglais) des différentes versions d'un même assemblage. Lorsqu'un assemblage référence un assemblage partagé, il utilise son nom fort. Or, le numéro de version fait partie du nom fort. Il n'y a donc pas de risque de charger une mauvaise version.

Le stockage côte à côte résout le problème connu sous le nom d'*enfer des DLLs*. (*DLL hell* en anglais). Ce problème survient lorsque l'on remplace une librairie dynamique *Windows* (i.e une DLL pour *Dynamic Link Library*) par sa nouvelle version. Le moindre problème de compatibilité entre les versions des DLLs implique que les application qui utilisent cette DLL et qui fonctionnaient correctement jusqu'alors avec l'ancienne version rencontrent des problèmes d'exécution avec la nouvelle. Ce problème est si important qu'il a une grande part de responsabilité dans la réputation d'instabilité des systèmes d'exploitation *Windows*. En plus de réduire l'instabilité du système, le stockage côte à côte dissipe l'énorme contrainte de la compatibilité ascendante souvent imposée au processus de développement d'une nouvelle version.

Le seul problème avec cette technique de stockage côte à côte est que le disque dur garde potentiellement toutes les versions de chaque assemblage. Pour les assemblages de tailles raisonnables, cela n'est pas vraiment un problème. Pour des assemblages volumineux, il est du devoir de l'éditeur de l'application de faire en sorte de partager un maximum de données entre les différentes versions.

## *Exécution côte à côte*

L'exécution côte à côte d'un assemblage consiste à autoriser l'exécution simultanée de plusieurs versions de cet assemblage. L'attribut .NET `AssemblyFlags` autorise les versions d'un assemblage à être exécutées côte à côte dans un même domaine d'application ou dans un même processus ou seulement sur la même machine.

Il vaut mieux éviter d'autoriser l'exécution côte à côte d'un même assemblage dans un domaine d'application ou même dans un processus. En effet, cette possibilité est surtout utile pour permettre à plusieurs applications d'utiliser simultanément un assemblage, chacune ayant recours à la version qui lui convient. Il est plutôt maladroit d'en arriver à ce que de telles applications s'exécutent au sein d'un même processus ou pire, au sein d'un même domaine d'application.

En outre, l'exécution côte à côte d'un assemblage peut, dans certains cas, introduire des problèmes très difficiles à corriger. Ces problèmes surviennent lorsqu'une même ressource est utilisée simultanément par plusieurs versions d'un assemblage. Il y a un risque que les différentes versions de l'assemblage ne s'accordent pas sur la façon d'utiliser la ressource. La conséquence est un comportement imprévisible.

## *Visualiser et manipuler le GAC*

Pour visualiser le GAC à partir d'un explorateur classique il existe l'extension du shell `ShFusion.dll` installée automatiquement lors du déploiement de la plateforme .NET sur une machine. Cette extension permet de visualiser et de manipuler d'une manière claire tous les assemblages partagés dans le GAC. Pour utiliser cette extension, il suffit de visualiser le répertoire GAC avec un explorateur *Windows*, comme sur la Figure 3-1 :

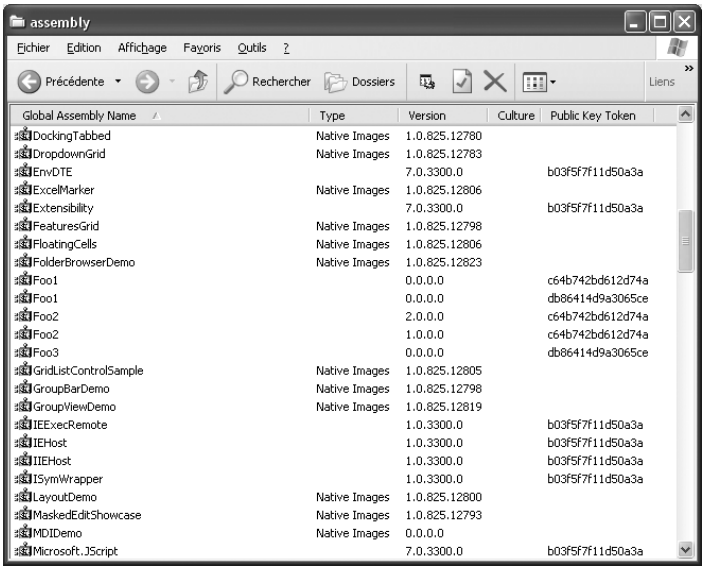


*Figure 3-1 : Visualisation du répertoire GAC*

L'outil `GACUtil.exe` utilisable en ligne de commande, permet aussi de visualiser et de manipuler le répertoire GAC. `GACUtil.exe` a de nombreuses options bien décrites dans les **MSDN**. Par exemple l'option `-l` permet de lister tous les assemblages partagés et l'option `-i` suivie du nom du fichier d'un assemblage contenant le manifeste permet d'insérer un assemblage à nom fort dans le répertoire GAC.

Seuls les administrateurs de la machine peuvent modifier le répertoire GAC. Vous pouvez effectivement visualiser et modifier la structure interne du répertoire GAC à partir d'une fenêtre de commande. Cependant il est impératif de ne jamais chercher à modifier la structure du répertoire GAC sans passer par l'outil `GACUtil.exe` ou l'extension du shell `ShFusion.dll` (bien que cette manipulation soit possible avec un peu d'astuce).

Le *cache des images natives* décrit page 113, est inclus dans le répertoire GAC. Ainsi lorsque l'on visualise le répertoire GAC, que ce soit avec `GacUtil.exe` ou l'extension `ShFusion.dll`, les images natives sont aussi visualisées.

# *Assemblage de stratégie d'éditeur (Publisher policy)*

## *La problématique*

Malgré le problème connu sous le nom d'enfer des DLLs, le modèle de partage des DLLs par plusieurs applications a un avantage par rapport au modèle de partage des assemblages basé sur la présence du numéro de version dans le nom fort.

Supposons qu'un éditeur redistribue une nouvelle version de l'assemblage partagé parce que des bugs ont étés corrigés. Cette version est complètement compatible avec la précédente (compatibilité ascendante). Cette nouvelle version va être installée côte à côte avec l'ancienne version dans le répertoire GAC. Cependant les applications ne vont pas utiliser cette nouvelle version car elles contiennent le numéro de l'ancienne version *codé en dur* dans leurs manifestes.

Ce problème n'existe pas avec le modèle de DLL classique puisque l'ancienne version de la DLL est purement et simplement remplacée par sa nouvelle version.

## *La solution*

On pourrait résoudre ce problème en recompilant et en réinstallant toutes les applications clientes de l'assemblage partagé. La recompilation permettrait aux manifestes des assemblages clients de mettre à jour le numéro de version de l'assemblage partagé. Clairement cette solution lourde et fastidieuse est impraticable et il a fallu trouver une autre solution.

La solution proposée par *Microsoft* est de créer un *assemblage de stratégie d'éditeur* (*publisher policy* en anglais). Cet assemblage, placé dans le répertoire GAC, n'est là que pour créer une redirection sur le numéro de version d'un autre assemblage partagé, lui aussi stocké dans le répertoire GAC. Lorsqu'une application demande la version 3.3.0.0 de l'assemblage `XXX` elle utilisera en fait la version 3.3.1.0 de l'assemblage `XXX`, car l'assemblage de stratégie d'éditeur associé à `XXX` contient l'information de redirection de 3.3.0.0 vers 3.3.1.0.

L'assemblage de stratégie d'éditeur n'est qu'un fichier de configuration. La redirection est effectuée par le CLR. En effet, c'est le CLR qui est chargé de localiser et de charger les assemblages durant l'exécution. Or, le CLR tient compte des assemblages de stratégie d'éditeur lors de cette opération.

Un assemblage de stratégie d'éditeur ne redirige que les appels destinés à un assemblage. De plus il peut y avoir plusieurs assemblages de stratégie d'éditeur pour un même assemblage. Dans ce cas, seule la version la plus récente de l'assemblage de stratégie d'éditeur sera prise en compte par le CLR.

Le comportement par défaut du CLR est de tenir compte des assemblages de stratégie d'éditeur. Cependant pour une application donnée, qui utilise un assemblage partagé donné, l'utilisateur de l'application peut décider que le CLR ne doit pas tenir compte de l'assemblage de stratégie d'éditeur. Ceci est bien pratique dans le cas où l'utilisateur se rend compte que la nouvelle version crée plus de problèmes qu'elle n'en résout (et nous savons bien que ce genre de situation arrive !). Nous exposons en page 106 comment réaliser cette manipulation.

Comprenez bien que les assemblages de stratégie d'éditeur sont nécessaires lorsqu'un éditeur a légèrement modifié un assemblage (pour une correction de bug en général) et n'a pas modifié la compatibilité ascendante. Les assemblages de stratégie d'éditeur ne doivent être utilisés que dans le cadre précis de cette problématique.

## *Création d'un assemblage de stratégie d'éditeur*

Concrètement un assemblage de stratégie d'éditeur est composé de deux modules. Un module est un fichier de configuration XML d'extension `.config` qui contient les informations de redirection souhaitées par l'éditeur. L'autre module ne contient qu'un manifeste et est construit par

l'outil al.exe, qui est présenté en page 37. Le fichier de configuration XML doit ressembler à celui-ci :

*Exemple 3-15 : Foo2.config*

```
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
         <dependentAssembly>
            <assemblyIdentity name="Foo2" culture= "neutral"
                              publicKeyToken="C64B742BD612D74A" />
            <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0"/>
         </dependentAssembly>
      </assemblyBinding>
   </runtime>
</configuration>
```

Remarquez l'élément <assemblyIdentity> qui reprend les composantes du nom fort de l'assemblage sur lequel doit se faire la redirection (à part le numéro de version).

Remarquez l'élément <bindingRedirect> qui redirige le numéro de version. Cet élément peut rediriger un intervalle de numéro de version, vers un numéro de version, avec la syntaxe suivante : "0.5.0.0-1.2.23.3". Vous pouvez aussi avoir recours à plusieurs éléments <bindingRedirect>.

La création du module de l'assemblage de stratégie d'éditeur contenant le manifeste, se fait avec l'outil al.exe utilisé avec la syntaxe suivante :

```
>al.exe /out:policy.1.0.Foo2.dll /version:1.0.0.0 /keyfile:Cles.snk
/linkresource:Foo2.config
```

- L'option /out précise le nom qu'aura le module contenant le manifeste. Ce nom est formé par des règles bien précises.
  - policy indiquera au CLR que c'est un assemblage de stratégie d'éditeur ;
  - 1.0 indiquera au CLR que cette stratégie d'éditeur s'applique aux demandes de l'assemblage Foo2.dll avec la version 1.0. Seul les numéros majeurs et mineurs sont précisés ici ;
  - Foo2 indiquera au CLR que cette stratégie d'éditeur s'applique aux demandes de l'assemblage Foo2.dll.
- L'option /version s'applique à l'assemblage de stratégie d'éditeur lui-même et non à Foo2.dll. Le CLR prendra toujours la stratégie d'éditeur avec la version la plus élevée pour un couple de numéros majeur/mineur donné.
- L'option /keyfile précise le fichier contenant la paire de clés publique/privé qui signera l'assemblage de stratégie d'éditeur. Ces clés doivent être les mêmes que celles qui signent Foo2.dll pour prouver au CLR que c'est bien le même éditeur qui fournit Foo2.dll et sa stratégie d'éditeur. De plus, en tant qu'assemblage partagé qui doit être stocké dans le GAC, l'assemblage de stratégie d'éditeur doit avoir un nom fort et doit donc être signé.
- L'option /linkresource précise le nom du fichier de configuration au format XML qui contient les informations de redirection de la stratégie d'éditeur. Comprenez bien que

`Foo2.config` devient un module de l'assemblage de stratégie d'éditeur grâce à cette option. Concrètement, le fichier `Foo2.config` n'est pas inclus physiquement dans le module `policy.1.0.Foo2.dll` mais est inclus logiquement dans l'assemblage `policy.1.0.Foo2`.

# *Introduction au déploiement d'applications .NET*

L'environnent de développement *Visual Studio 2005* permet la création de projets spécialement dédiés aux déploiements d'applications. Six types de projets de déploiement sont disponibles dans le menu *File* ▶ *New* ▶ *Project...* ▶ *Other Project Types* ▶ *Setup and Deployment* :

- **Cab Projet** : crée un fichier cab, qui rassemble les fichiers à installer et les compresse dans un fichier `.cab` (i.e une archive *cab*). Une section du présent chapitre est consacrée au déploiement par fichier `.cab`.
- **Smart Device Cab Projet** : crée un fichier cab spécialement adapté au déploiement sur *Windows CE* sur des machines type *Pocket PC* ou *Smart Phone*.
- **Merge Module Project** : crée un *module de déploiement*. Un tel module peut être intégré dans d'autres projets de déploiement. Ainsi un même module de déploiement peut être commun à plusieurs projets de déploiement. La notion de modules de déploiement n'est pas la même que la notion de modules d'assemblages.
- **Setup Project** : Crée un projet de déploiement exploitant la technologie MSI. Cette technologie permet d'effectuer des actions en plus d'installer des fichiers. Une section est consacrée à la technologie MSI un peu plus loin.
- **Web Setup Project** : Permet de déployer un projet d'application web en installant les fichiers dans des répertoires virtuels IIS. Le déploiement d'application web ASP.NET est un sujet particulier. Plus d'information à ce sujet sont disponibles en page 871.
- **Setup Wizard** : Ceci est une aide pas à pas au moyen d'un assistant, pour construire un projet de déploiement d'un de ces types.

Depuis sa première version la plateforme .NET présente une technologie nommée *No Touch Deployment* (*NTD*) spécialement conçue pour le déploiement d'application à partir d'internet. Cette technologie est toujours supportée par la version 2.0. Elle n'a pas évoluée car *Microsoft* a préféré miser sur une technologie nommée *ClickOnce* qui est beaucoup plus adaptée aux fortes contraintes d'un déploiement internet (sécurité, bande passante, mises à jour etc). Ces technologies font chacune l'objet d'une section de ce chapitre.

Il n'y a pas de type de projets de déploiement spéciaux aux technologies NTD et *ClickOnce*. NTD se gère à partir de fichier de configuration et de déploiement XCopy sur le serveur permettant le téléchargement de l'application. En revanche, nous verrons que *Visual Studio 2005* présente des facilités pour permettre le déploiement d'une application avec la technologie *ClickOnce*. Pour cela, il suffit que l'application soit représentée par un projet de type application fenêtrée *Windows Forms* ou application console.

## *MSI vs. .cab vs. XCopy vs. ClickOnce vs. NTD*

Vous avez principalement le choix entre ces cinq techniques pour déployer une application .NET qui n'est pas une application web ASP.NET. On peut diviser ce type de déploiement en deux catégories :

- Les **déploiements lourds** qui impactent profondément le système d'exploitation en installant des assemblages dans le GAC, en enregistrant des classes COM dans la base des registres, qui sont utilisables par plusieurs utilisateurs de la machine, qui requièrent des droits élevés type administrateurs pour leur exécution etc. Clairement, seule la technologie MSI est adaptée au déploiement de ces applications. De part leur taille et pour des raisons de sécurité, ce type de déploiement se fait parfois à partir d'un CD plutôt que par l'intermédiaire d'un réseau. L'utilisateur doit payer le prix d'un tel déploiement : une gestion de sécurité primaire avec la technologie authenticode (si déployé à partir d'un réseau), difficulté de mises à jour, exposition aux conséquences de l'enfer des DLLs, délais dus à la livraison d'un CD etc.
- Les **déploiements légers** d'application purement .NET qui impactent peu le système d'exploitation. Les quatre techniques *cab*, *XCopy*, *ClickOnce* et *NTD* peuvent être envisagées pour ce type de déploiement. *ClickOnce* est en général à privilégier de part ces fonctionnalités avancées notamment quant à la gestion de la sécurité, des mises à jour et de la bande passante. Mis à part la compatibilité ascendante, il n'y a pas de cas où la technologie NTD est préférable à *ClickOnce*. Les techniques *cab* et *XCopy* présentent l'avantage d'être très simples tant pour le développeur que pour l'utilisateur en général habitué au *copier/coller* de fichiers. Aussi, ces deux techniques ne sont pas dénuées d'intérêt quand il s'agit de réaliser des déploiements très simples. Dans ce cas, on préfèrera sûrement la technologie *cab* puisqu'un seul fichier est toujours plus facile à acheminer de l'éditeur à l'utilisateur que plusieurs.

## *MSI vs. ClickOnce*

Voici un tableau qui devrait vous aider à décider lorsque vous hésitez entre le déploiement avec MSI ou le déploiement avec *ClickOnce*. Ces deux technologies ne sont pas antagonistes. Il est courant que le socle/les prérequis d'une application se déploient avec MSI tandis la partie fonctionnelle, plus légère pour le système d'exploitation mais aussi plus évolutive, se déploie avec *ClickOnce* :

| Fonctionnalité | ClickOnce | MSI |
|---|---|---|
| Installation de fichiers. | X | X |
| Création de raccourcis dans le menu des programmes | X | X |
| Création de raccourcis bureau et autres. | | X |
| Installation de classes COM sans enregistrement dans la base des registres (voir page 287). | X | X |
| Installation de classes COM et de composants COM+. | | X |
| Association d'extension de fichiers. | | X |
| Installation de services *Windows*. | | X |
| Installation d'assemblages dans le GAC. | | X |
| Gestion de ODBC. | | X |
| Modification dans la base des registres. | | X |

| | | |
|---|---|---|
| *Self* réparation | | X |
| Interaction avec l'utilisateur à l'installation. | | X |
| Choix du répertoire d'installation des fichiers. | | X |
| Installation possible pour tous les utilisateurs. | | X |
| Actions spéciales au moment de l'installation ou de la désinstallation. | | X |
| Manipulation des droits sur les objets *Windows*. | | X |
| Conditions d'installation et vérification de la version et de l'état du système d'exploitation. | | X |
| Vérification et téléchargement automatique des mises à jour de l'application. | X | |
| Gestion de la sécurité évoluée basée sur le mécanisme CAS du CLR. | X | |
| Installation de partie de l'application à la demande. | X | |
| Possibilité de revenir à la version précédente après une mise à jour. | X | |

# *Déployer une application avec un fichier cab*

Vous avez la possibilité d'ajouter un projet de déploiement par fichiers *cab* à votre solution, comme le montre la Figure 3-2 :

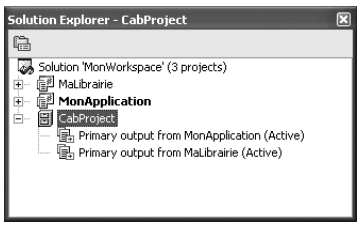


*Figure 3-2 : Un projet cab inséré dans une solution*

Vous pouvez ajouter des fichiers produits ou contenus par les autres projets de la solution (*Project output*) comme illustré sur la Figure 3-3. Ici nous avons ajouté les fichiers `MonApplication.exe` et `MaLibrairie.dll` en sélectionnant le type *Primary Output* pour chacun de ces projets.

Après compilation d'un projet de déploiement cab, un fichier d'extension `.cab` est produit. Avec un outil de visualisation d'archives vous pouvez examiner le contenu du fichier cab, et extraire les fichiers contenus comme le montre la Figure 3-4. Notez qu'un avantage de l'utilisation de ce type d'archive, est que les fichiers sont compressés.

Un fichier d'extension `.osd` est contenu dans le fichier cab. Ce fichier au format XML décrit le contenu du fichier cab :

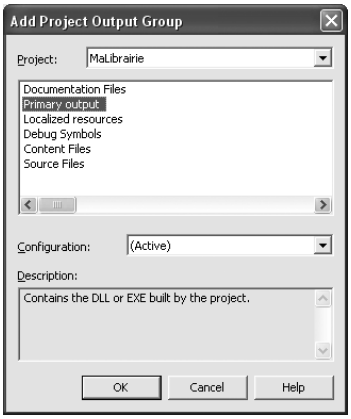


*Figure 3-3 : Ajout dans un projet de déploiement cab*

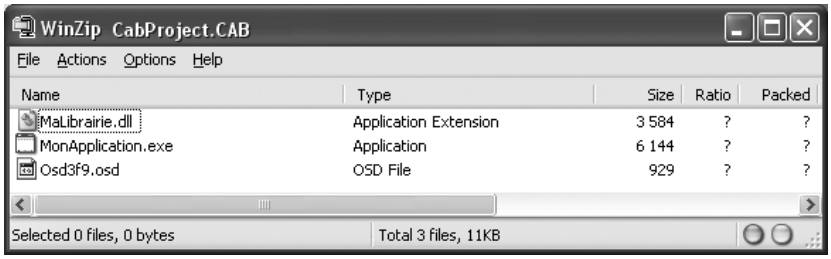


*Figure 3-4 : Utiliser Winzip pour l'analyse d'un fichier cab*

```
<?XML version="1.0" ENCODING='UTF-8'?>
<!DOCTYPE SOFTPKG SYSTEM "http://www.microsoft.com/standards/osd/osd.dtd">
<?XML::namespace href="http://www.microsoft.com/standards/osd/msicd.dtd"
    as="MSICD"?>
<SOFTPKG NAME="CabProject" VERSION="1,0,0,0">
   <TITLE> CabProject </TITLE>
   <MSICD::NATIVECODE>
      <CODE NAME="MonApplication">
         <IMPLEMENTATION>
            <CODEBASE FILENAME="MonApplication.exe">
            </CODEBASE>
         </IMPLEMENTATION>
      </CODE>
   </MSICD::NATIVECODE>
   <MSICD::NATIVECODE>
      <CODE NAME="MaLibrairie">
         <IMPLEMENTATION>
            <CODEBASE FILENAME="MaLibrairie.dll">
            </CODEBASE>
```

```
          </IMPLEMENTATION>
        </CODE>
      </MSICD::NATIVECODE>
</SOFTPKG>
```

# *Déployer une application avec la technologie MSI*

## *Ajouter les fichiers de votre application*

Lorsque vous avez ajouté le projet setup à votre solution, vous disposez de la vue « Système de fichiers » (*File System* en anglais). Cette vue contient par défaut trois répertoires : Le répertoire de l'application ; Le bureau ; Le menu démarrer.

Les fichiers que vous allez mettre dans ces répertoires durant la construction du projet *setup* seront automatiquement copiés dans les répertoires correspondants, sur la machine cible, lors de l'installation. En général on fait en sorte de placer dans ces répertoires les assemblages non partagés, produits des autres projets de la solution. Ceci est illustré sur la Figure 3-5. Remarquez que cette opération se fait de la même manière que dans un projet de déploiement utilisant un fichier cab.

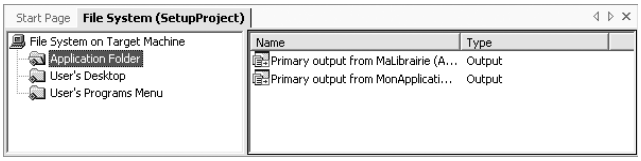


*Figure 3-5 : La vue système de fichiers*

## *Installer des raccourcis*

Il est très facile d'ajouter des raccourcis sur le bureau ou dans le menu *démarrer* à partir de la vue système de fichiers. Il suffit de cliquer droit sur l'exécutable, d'ajouter un raccourci et de déplacer le raccourci dans le répertoire souhaité (bureau ou « menu démarrer »).

## *Ajouter un assemblage partagé, dans le répertoire GAC*

Pour ajouter un assemblage partagé dans le répertoire GAC à partir d'un projet de déploiement « *setup project* », il faut d'abord voir le répertoire GAC dans la vue « *système de fichier* ». Pour cela, il suffit de cliquer droit dans l'espace réservé au répertoire, et d'ajouter le répertoire GAC, comme sur la Figure 3-6. Ensuite vous n'avez plus qu'à ajouter vos assemblages partagés dans ce répertoire. De nombreux autres répertoires spéciaux peuvent être ajoutés.

## *Les propriétés du projet*

Vous avez en fait les deux types de propriétés du projet.

- Les pages de propriétés, accessibles par *maj F4*. Elles permettent essentiellement de gérer les configurations du projet *setup* (Debug /Release...) et d'ajouter une signature *Authenticode* au projet.

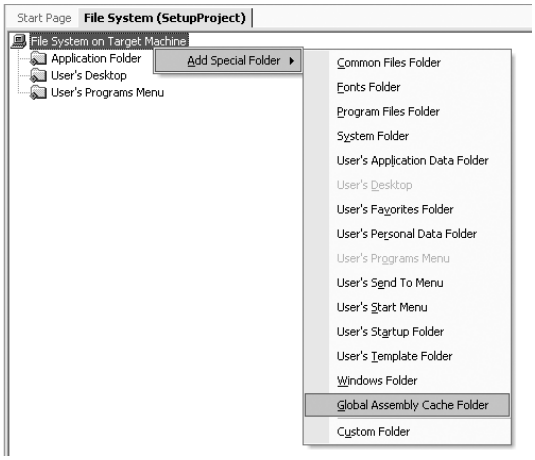


*Figure 3-6 : Ajout du répertoire GAC*

- La fenêtre de propriété du projet *setup* qui vous permet de configurer de nombreux attributs associés au projet, comme son nom, sa culture ou le nom de l'éditeur, comme le montre la Figure 3-7.

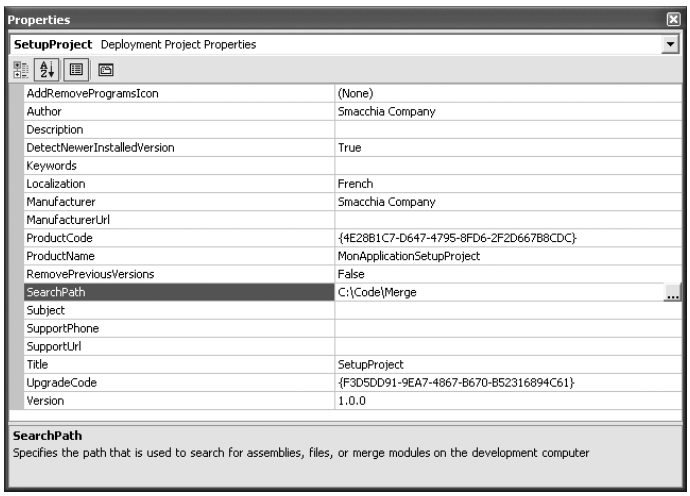


*Figure 3-7 : Propriétés d'un projet setup*

## *Manipuler la base des registres*

Vous avez la possibilité d'ajouter des clés dans la base des registres relatives à votre application lors de son installation. Il suffit pour cela de sélectionner la vue *Registre* et de modifier les clés,

comme sur la Figure 3-8. Comprenez bien que la plupart des applications .NET ne devraient pas utiliser la base des registres, aussi faites attention à vos modifications.

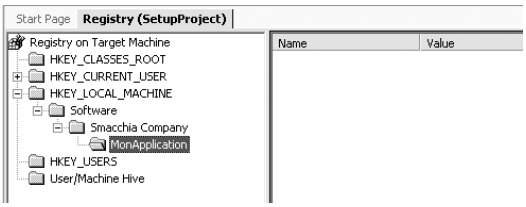


*Figure 3-8 : Configurer la base des registres*

## *Effectuer des actions personnalisées durant l'installation*

Il peut être très utile de lancer un exécutable durant l'installation ou la désinstallation d'une application. Par exemple, vous pouvez créer un exécutable qui vérifie certains paramètres de configuration sur votre machine, nécessaires au bon fonctionnement de l'application. Vous pouvez aussi souhaitez enregistrer un objet COM durant l'installation et le désenregistrer durant la désinstallation. Dans ce dernier exemple, il est nécessaire d'ajouter le fichier `regsvr32.exe` à la liste des fichiers qu'installe l'application. Ensuite, vous n'avez plus qu'à sélectionner la vue *actions personnalisées* et à ajouter des références comme illustré par la Figure 3-9 :

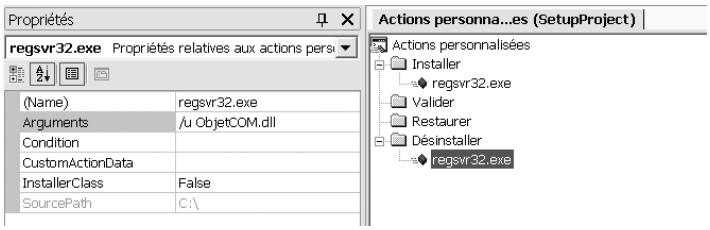


*Figure 3-9 : Ajouter des actions personnalisées*

## *Modifier l'interface utilisateur de l'installation*

Vous pouvez modifier simplement le cours de l'installation en modifiant les fenêtres de dialogue proposées par *Visual Studio* dans la vue *Interface Utilisateur*, comme illustré sur la Figure 3-10. Vous pouvez ajouter de nombreux dialogues, comme illustré sur la Figure 3-11.

# *Déployer une application avec la technologie ClickOnce*

Pour tirer partie de la technologie *ClickOnce*, il est absolument nécessaire d'avoir assimilé la technologie *Code Access Security* (CAS) exposée dans le chapitre 6 « La gestion de la sécurité ».

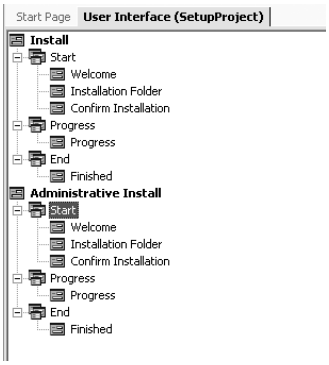


*Figure 3-10 : Enchaînement des interfaces utilisateurs*

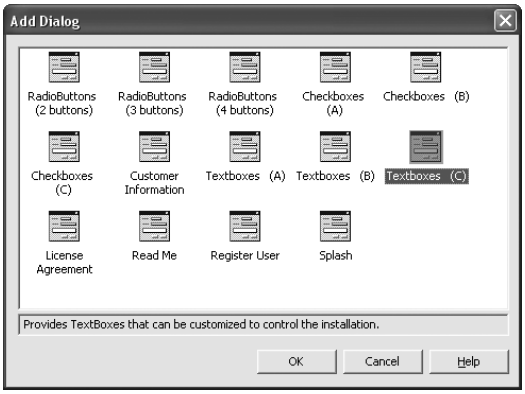


*Figure 3-11 : Dialogues possibles pour l'installation*

## *Organisation du répertoire de déploiement*

Commençons par décrire l'arborescence de fichiers qui permet le déploiement d'une application avec *ClickOnce*. Elle doit se situer dans un répertoire de déploiement qui peut être un répertoire virtuel d'IIS, un répertoire FTP, un répertoire partagé au sein d'un intranet ou un répertoire d'un CD. La technologie *ClickOnce* requiert deux fichiers particuliers en plus de l'ensemble des fichiers de l'application à déployer :

- Le *manifeste de l'application* : C'est un fichier XML d'extension `.manifest` qui référence l'ensemble des fichiers de l'application et qui définit l'ensemble des permissions CAS nécessaires pour son exécution. Son nom est la concaténation du nom de l'assemblage contenant le point d'entrée de l'application (extension `.exe` comprise) avec l'extension `.manifest`.
- Le *manifeste du déploiement* : C'est un fichier XML d'extension `.application`. qui référence un fichier manifeste de l'application. Il contient aussi les paramètres du déploiement.

Ces deux fichiers manifestes doivent être signés numériquement par la technologie *authenticode* pour pouvoir être utilisés. Cela se traduit par une présence d'un élément `<signature>` dans

chacun d'eux. La technologie *authenticode* est décrit en page 230. Vous pouvez aussi consulter l'article **ClickOnce Deployment and Authenticode** des **MSDN**.

L'organisation des fichiers dans le répertoire de déploiement est la suivante :

```
.\MyApp_1_0_0_0.application              // Manifeste de déploiement.
.\MyApp_1_0_0_0\MyApp.exe.manifest       // Manifeste de l'application.
.\MyApp_1_0_0_0\MyApp.exe.deploy         // Fichiers de l'application ...
.\MyApp_1_0_0_0\MyAppClassLib.dll.deploy // ... avec l'extension .deploy.
.\MyApp_1_0_0_0\en-US\MyApp.resources.dll.deploy
...
```

On remarque qu'il existe un sous répertoire par version de l'application. Le nom de ce sous répertoire ne contient pas nécessairement d'indication sur le numéro de version bien que ceci soit une bonne pratique. Chacun de ces sous répertoires contient le manifeste de l'application relatif à la version ainsi que tous les fichiers à déployer pour cette version. Une extension `.deploy` est rajoutée au nom de chacun de ces fichiers.

Le manifeste de déploiement est stocké dans le répertoire de déploiement. Puisqu'il référence un manifeste d'application qui est relatif à une version de l'application, il est lui-même relatif à une version de l'application. Aussi, il est judicieux que son nom soit la concaténation du nom de l'assemblage contenant le point d'entrée de l'application (extension `.exe` non comprise) avec une indication sur le numéro de version suivie de l'extension `.application`. Ainsi, plusieurs manifestes de déploiement relatifs à plusieurs versions d'une même application peuvent cohabiter dans le même répertoire de déploiement.

## *Concevoir un déploiement ClickOnce*

Il existe plusieurs possibilités pour créer les fichiers manifestes, énumérées ici de la plus pratique à la plus fastidieuse :

- Utiliser le menu *Properties* ▶ *Publish* de *Visual Studio 2005* sur un projet de type console ou application *Windows Forms*.
- Utiliser l'outil fenêtré `mageui.exe` (*mage* pour *Manifest Generation and Editing Tool*) disponible dans le répertoire SDK de l'installation de *Visual Studio 2005*.
- Utiliser l'outil en ligne de commande `mage.exe` disponible dans le même répertoire que `mageui.exe`.
- Construire ces fichiers XML à la main en se basant sur la documentation officielle.

L'outil `mage.exe` est particulièrement utile si vous souhaitez intégrer la création des fichiers manifestes dans un script MSBuild. L'outil `mageui.exe` est surtout pédagogique car il permet d'obtenir une vue précise et compréhensible du contenu de chacun des fichiers manifestes. L'utilisation de ces outils est décrite dans l'article **Walkthrough : Deploying a ClickOnce Application Manually** des **MSDN**. Hormis ces deux raisons, nous vous conseillons d'avoir recours à *Visual Studio 2005* qui présente l'interface suivante :

- **Application Files... :** Permet de choisir l'ensemble des fichiers qui constituent l'application. L'assemblage exécutable généré par le projet courant fait automatiquement partie de cet ensemble. Les assemblages référencés par ce projet, les assemblages satellites générés par ce projet ainsi que tous autres fichiers relatifs à ce projet font aussi partie de cet ensemble. Mis

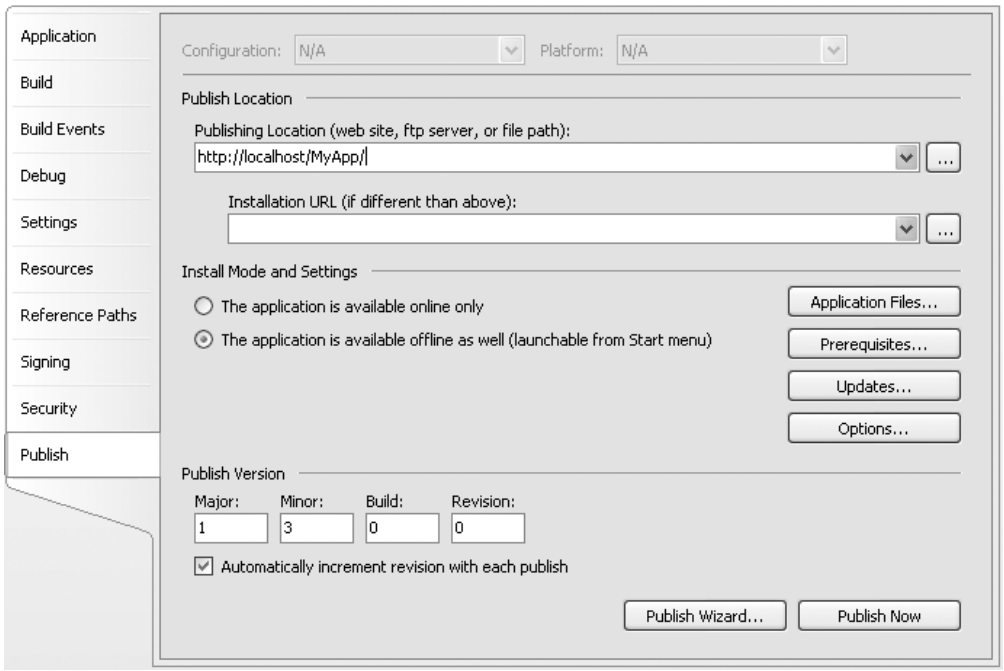


*Figure 3-12 : Visual Studio et ClickOnce*

à part l'assemblage exécutable qui est forcément requis, chaque fichier peut être marqué comme prérequis, requis ou optionnel. Dans le premier cas, le fichier doit être disponible avant l'installation de l'application. Dans le second cas le fichier doit être téléchargé lors de l'installation de l'application. Dans le troisième cas le fichier doit faire partie d'un groupe de fichiers optionnels. Ces groupes permettent un téléchargement à la demande des parties fonctionnelles d'une application. Nous revenons sur ce point un peu plus loin.

- **Prerequisites...** : Permet de choisir les prérequis de l'installation de l'application. Vous pouvez choisir notamment le *framework .NET 2.0*, *SQL Server 2005 Express Edition* mais aussi n'importe quel application, fichier ou *framework* à installer. *Visual Studio 2005* vous permet de générer un fichier `setup.exe` que la littérature anglo saxonne nomme *bootstrapper*. *ClickOnce* proposera au client de télécharger et d'exécuter le *bootstrapper* avant l'installation de l'application si sa machine n'a pas les prérequis. Vous pouvez spécifier d'où le bootstrapper doit télécharger chaque prérequis (site de déploiement, site officiel du composant etc). Techniquement l'utilisateur n'a pas besoin d'être administrateur de la machine pour exécuter le *bootstrapper*. En pratique, il a souvent besoin de l'être puisque le *bootstrapper* doit généralement installer des composants qui requièrent une installation avec la technologie MSI. Plus d'information concernant le *bootstrapper* sont disponible dans l'article **Use the Visual Studio 2005 Bootstrapper to Kick-Start Your Installation** de *Sean Draine* consultable en ligne dans le **MSDN Magazine d'octobre 2004**.
- **Updates...** : Permet de positionner une politique de mise à jour de l'application. Une section est consacrée aux mises à jour un peu plus loin.

- **Options...** : Permet de positionner les paramètres de l'application tels que le nom de l'éditeur, du produit, la génération d'une page web d'aide au téléchargement, la culture cible du déploiement, l'utilisation de l'extension `.deploy` etc.
- **Publish Wizard...** et **Publish Now** : permet de publier l'application, c'est-à-dire de générer les deux fichiers manifestes et de construire l'arborescence de fichier que nous avons présenté. Il faut publier votre application dans un répertoire de déploiement spécifique pour chaque culture supportée. Vous pouvez choisir si le répertoire de déploiement est un répertoire virtuel IIS, un répertoire FTP ou un répertoire *Windows* classique que vous pouvez alors dupliquer sur un CD.

Cette interface vous permet aussi de choisir si l'application est disponible *offline*. Dans ce cas un raccourci vers l'application est installé dans le menu des programmes et l'utilisateur n'a pas besoin d'être connecté à internet pour la lancer.

## *Déploiement de l'application et impératifs de sécurité*

Pour déployer l'application sur une machine, il suffit d'exécuter le fichier manifeste de déploiement. Dans le cas d'un déploiement web, cela se fait en général par l'intermédiaire d'une page qui contient un lien vers ce fichier. Dans les autres cas il suffit de double cliquer ce fichier (à partir du répertoire FTP, du répertoire partagé intranet ou du répertoire du CD).

À moins que les deux fichiers manifestes aient été signés avec un certificat X.509 vérifiable par la machine cible, une boite de dialogue apparaît signalant à l'utilisateur que l'application qu'il est en train d'installer a été conçue par un éditeur inconnue. Il a alors le choix de continuer l'installation ou non.

Si l'éditeur de l'application a pu être vérifié grâce au certificat, l'ensemble des permissions demandées dans le manifeste de l'application lui est accordé. Il en est de même si l'application est installée à partir d'un CD. Sinon, un certain ensemble de permissions CAS peut lui être accordé par les stratégies de sécurité CAS. Tout dépend de la zone d'où l'installation est effectuée (*internet untrusted*, *internet trusted*, *intranet* etc).

Si l'ensemble des permissions décrit dans le manifeste de l'application n'est pas complètement inclus dans l'ensemble des permissions que les stratégies de sécurité CAS sont prêtent à lui accorder alors une boite de dialogue informe l'utilisateur. S'il choisit de continuer l'installation, l'ensemble des permissions décrit dans le manifeste de l'application sera accordé à chaque exécution de l'application, sinon l'installation est annulée. La littérature anglo saxonne qualifie cette étape de *permission elevation*.

Une fois ces deux barrières potentielles passées (au pire nous en sommes à trois clicks ☺), *ClickOnce* installe l'application dans un répertoire dédié à cette application. Nous reviendrons sur ce répertoire. Un raccourci vers l'application est installé dans le menu des programmes si l'application est disponible *offline*. L'application peut être désinstallée à partir du menu *Ajout/Suppression de programmes*. Le manifeste de déploiement a la possibilité de spécifier que l'application doit être démarrée dés que le l'installation est terminée.

L'application n'est installée que pour l'utilisateur courant. Si plusieurs utilisateurs d'une machine ont besoin de cette application, elle devra être installée pour chacun d'eux dans des répertoires différents.

L'application des stratégies de sécurité de la machine ne se fait qu'à l'installation et lors des mises à jour de l'application. Une fois que l'application est installée elle s'exécutera à chaque fois dans le cadre des permissions spécifiées dans le manifeste.

Contrairement à la technologie MSI, vous ne pouvez pas personnaliser le processus de déploiement d'une application *ClickOnce*. Concrètement, vous ne pouvez pas fournir de dialogues demandant à l'utilisateur des informations comme le répertoire où installer l'application. Cette contrainte est essentielle pour garantir une sécurité optimale.

## *Installation à la demande*

Nous avons vu qu'un fichier d'une application peut être marqué comme optionnel. Dans ce cas, il fait partie d'un groupe de fichiers optionnels. Tous les fichiers d'un tel groupe sont téléchargés explicitement par le code de l'application lorsque qu'à l'exécution elle a besoin d'un de ces fichiers pour la première fois. C'est l'installation à la demande.

Le *framework* représenté par l'espace de nom standard `System.Deployment` permet au code de votre application de télécharger un groupe de fichier de l'application non encore installé. Ce code doit être exécuté lors du déclenchement d'un événement type `AppDomain.AssemblyResolve` ou `AppDomain.ResourceResolve`. On parle parfois d'installation transactionnelle d'un tel groupe de fichiers puisque soit ils sont tous installés soit aucun n'est installé.

L'exemple suivant expose un assemblage exécutable `MyApp` qui référence un assemblage bibliothèque `MyLib`. Supposons que dans un projet *ClickOnce* de déploiement, `MyLib` fasse partie d'un groupe de fichiers optionnels nommé `MyGroup`. Le code de la méthode `AssemblyResolveHandler()` est déclenché lors de la première exécution de `MyApp`, lorsque `MyLib` n'est pas trouvé. Ce code télécharge les fichiers du groupe `MyGroup` puis récupère et retourne l'assemblage `MyLib`.

Notez la nécessité de la présence de la classe `Foo`. Si nous ne nous servions pas d'une classe intermédiaire pour invoquer `MyClass`, le compilateur JIT compilerait la classe `Program` avant d'avoir exécuté la méthode `Main()`. L'événement `AssemblyResolve` serait donc déclenché avant même que la méthode `AssemblyResolveHandler()` ait pu être abonnée :

*Exemple 3-16 :* *MyLib.cs*

```
public class MyClass {
   public override string ToString() {
      return "Bonjour de MyLib." ;
   }
}
```

*Exemple 3-17 :* *MyApp.cs*

```
using System ;
using System.Reflection ;
using System.Deployment.Application ;
class Program {
   public static void Main() {
      AppDomain.CurrentDomain.AssemblyResolve += AssemblyResolveHandler ;
      Console.WriteLine("Bonjour de MyApp.") ;
      Foo.FooFct() ;
   }
   static Assembly AssemblyResolveHandler(object sender,
                                          ResolveEventArgs args) {
      if ( ApplicationDeployment.IsNetworkDeployed ) {
         // La propriété CurrentDeployement retourne null
```

```
         // lorsque l'application n'est pas déployée
         // d'où le test de la propriété IsNetworkDeployed.
         ApplicationDeployment currentDeployment =
            ApplicationDeployment.CurrentDeployment ;
         currentDeployment.DownloadFileGroup("MyGroup") ;
         Console.WriteLine("Téléchargement...") ;
      }
      return Assembly.Load("MyLib") ;
   }
}
class Foo {
   public static void FooFct() {
      MyClass a = new MyClass() ;
      Console.WriteLine(a) ;
   }
}
```

## *Mises à jour d'une application installée avec ClickOnce*

Les outils *Visual Studio 2005*, `mageui.exe` et `mage.exe` vous permettent de paramétrer la façon dont une application déployée avec *ClickOnce* gère ses mises à jour. Vous pouvez spécifier quand l'application doit vérifier que des mises à jour sont disponibles (à chaque exécution ou périodiquement). Vous pouvez aussi spécifier quand ces mises à jour doivent être téléchargées :

- Après le démarrage de l'application pour un démarrage rapide mais une prise en compte des mises à jour qu'à la prochaine exécution.
- Ou avant le démarrage de l'application pour s'assurer que les utilisateurs exécutent toujours la version la plus récente au prix d'un démarrage parfois ralenti.

Lors de la mise à jour d'une application la technologie *ClickOnce* ne télécharge que les fichiers qui ont changé. En outre l'ensemble des permissions accordées lors de l'installation d'une application sera automatiquement hérité par les futures mises à jour. Enfin, dés lors qu'une application a été au moins une fois mise à jour sur une machine la technologie *ClickOnce* se met à stocker sur cette machine les données nécessaires pour éventuellement revenir à la version précédente. Ainsi, l'utilisateur ne prend pas de risques en mettant à jour une application puisqu'il a la possibilité dans le menu de désinstallation de programme de revenir à la version précédente.

## *Facilités pour manipuler l'ensemble des permissions CAS requises par une application*

Les projets *Visual Studio 2005* de type console ou application *Windows Forms* présentent un menu *Properties* ▶ *Security* qui facilite grandement la gestion des permissions CAS lors du développement d'une application :

Ce menu permet de spécifier si l'application a besoin de la permission `FullTrust` pour être exécutée ou non. Dans le second cas, une grille vous aide à construire l'ensemble exact des permissions dont l'application à besoin pour s'exécuter. C'est cet ensemble qui sera spécifié dans le manifeste de l'application. Lorsqu'un type de permission est sélectionné, vous pouvez cliquer le bouton *Properties* pour obtenir une fenêtre qui permet de configurer finement la permission.

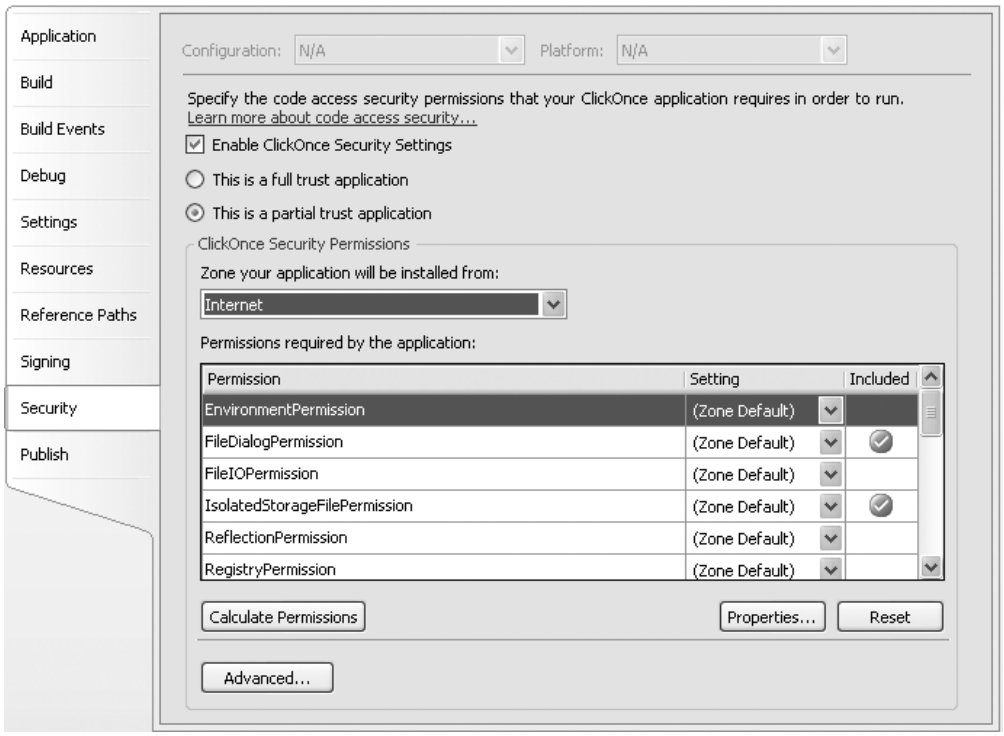


*Figure 3-13 : Visual Studio et les permissions CAS*

*Visual Studio 2005* vous aide à comparer cet ensemble avec les ensembles de permissions accordés par défaut par les stratégies de sécurité. Pour cela, il faut choisir à partir de quelle zone l'application sera déployée. Un avertissement signale quand une permission non accordée par défaut à la zone sélectionnée est demandée. Un avertissement signifie qu'au moment de l'installation à partir de cette zone, la technologie *ClickOnce* demandera à l'utilisateur de choisir s'il souhaite élever l'ensemble des permissions, ce qui fait toujours mauvais effet !

Il est fastidieux de déterminer l'ensemble exact de permissions dont une application a besoin pour s'exécuter correctement. L'outil `permcalc.exe` fourni avec *Visual Studio 2005* permet de calculer cet ensemble en analysant le code IL de l'assemblage exécutable ainsi que le code IL des assemblages référencés (récursivement). Selon la taille de votre application, ce calcul peut prendre un certain temps, de l'ordre de la minute. Vous pouvez vous servir de cet outil en ligne de commande pour produire l'ensemble des permissions au format XML. Ce contenu peut alors être intégré dans le manifeste de l'application. Le bouton *Calculate Permissions* du menu *Security* permet d'invoquer cet outil et d'obtenir son résultat dans la grille des permissions.

*Visual Studio 2005* vous permet de déboguer votre application dans le contexte de sécurité défini par l'ensemble de permissions que vous avez construit dans le menu *Security*. Cette possibilité est très pratique puisque c'est aussi le contexte de sécurité dans lequel s'exécutera l'application chez les clients. Pour cela, *Visual Studio 2005* génère à la compilation un assemblage nommé `[Nom de l'application].vshost.exe` dans le répertoire de sortie qui contient aussi l'assemblage `[Nom`

de l'application].exe. C'est cet assemblage exécutable *VSHost* qui est lancé par *Visual Studio 2005* lorsque vous demandez de déboguer l'application. Le code de cet assemblage positionne l'ensemble des permissions puis charge et exécute l'application.

Enfin, sachez que l'intellisense de *Visual Studio 2005* tient compte de l'ensemble des permissions défini dans le menu *Security*. Concrètement, il grise les noms des classes et méthodes du *framework* qui requièrent des permissions non accordées.

## *Détails sur l'installation et l'exécution d'une application déployée avec ClickOnce*

Si vous ouvrez le gestionnaire des tâches sur une machine qui est en train d'exécuter une application nommée XXX déployée avec *ClickOnce*, vous pourrez constater qu'il n'y a pas de processus en cours nommé XXX.exe. En effet, une application *ClickOnce* qui n'a pas la permission FullTrust s'exécute au sein d'un processus nommé AppLaunch.exe. Il s'occupe de la gestion de l'ensemble des permissions CAS accordées à l'application. Le menu ajouté pour une application déployée avec *ClickOnce* fait référence à un fichier *référence d'application Windows* d'extension .appref-ms. Lors de l'ouverture d'un tel fichier *Windows* exécute des objets COM définis dans le composant dfshim.dll. Ces objets sont responsables du lancement de AppLaunch.exe.

Lors du lancement d'une application déployée avec *ClickOnce* vous pouvez aussi observer l'apparition du processus dfsvc.exe. C'est le moteur d'exécution de *ClickOnce*. C'est ce processus qui s'occupe de la vérification et du téléchargement des mises à jour. Ce processus s'auto détruit après 15 minutes d'inactivité.

Les fichiers d'une application déployée avec *ClickOnce* sont installés dans un sous répertoire du *cache des applications déployées avec ClickOnce*. Ce cache est propre à chaque utilisateur et à une taille bornée (de l'ordre de 100 Mo). C'est le répertoire caché \Documents and Settings\[Nom de l'utilisateur]\Local Settings\Apps\. Le nom du répertoire d'une de ces sous répertoires est composé de valeurs de hachages calculées sur l'application lors de l'installation. Une association entre ce répertoire et le fichier référence d'application est présente dans la base des registres. Vous pouvez retrouver ce répertoire soit par l'intermédiaire de la propriété AppDomain.CurrentDomain.BaseDirectory qui représente le répertoire d'hébergement de l'assemblage de l'application, soit en lançant une recherche des fichiers de votre application sous le répertoire Apps. Lors de la mise à jour d'une application, un nouveau répertoire *frère* du précédent est créé et *ClickOnce* fait en sorte qu'il n'y ait que deux répertoires consacrés à cette application : celui de la version précédente et celui de la version courante.

Lors du déploiement d'une application, *ClickOnce* fait la différence entre les fichiers propres à l'application (essentiellement l'arborescence de référencement des assemblages obtenue à partir de l'assemblage exécutable de l'application, assemblages satellites compris) et les fichiers de données de l'application (les fichiers de configuration, les fichiers de base de données .mdb, les fichiers XML etc). Les fichiers de données sont déployés dans un sous répertoire du *cache des données des applications déployées avec ClickOnce* (à savoir \Documents and Settings\[Nom de l'utilisateur]\Local Settings\Apps\Data). Ce répertoire de données de l'application est programmatiquement accessible par la propriété string ApplicationDeployment.DataDirectory{get;} et par la propriété stringSystem.Windows.Forms.Application.LocalUserAppDataPath\{get;\}. Il faut que l'application ait les permissions CAS d'accès à ce répertoire pour pouvoir exploiter les données. Pour cette raison, vous pouvez préférer utiliser le mécanisme de stockage isolé décrit en page 205 qui ne requiert que la permission de faire du stockage isolé. En effet, cette permission est accordée par défaut aux zones *internet* et *intranet*.

Lors de la mise à jour d'une application, les anciens fichiers de données sont copiés dans le nouveau répertoire de données. Si *ClickOnce* télécharge une nouvelle version d'un fichier de données, celle-ci écrase l'ancienne version. Cependant, pour éviter la perte de données l'ancienne version est alors copiée dans un sous répertoire `inc`.

## *Déployer une application avec la technologie No Touch Deployment*

Lorsque le *framework* .NET est installé sur une machine, vous pouvez démarrer un assemblage téléchargeable sur le web directement à partir d'internet explorer. Cette possibilité est nommée *No Touch Deployment* (*NTD*).

Supposons que l'assemblage `Foo.exe` qui a le code suivant, soit téléchargeable à partir de l'URL : `\texttt{www.smacchia.com/Foo.exe}`. Nous utilisons le fait que la propriété `Assembly.CodeBase` contienne l'URL à partir de laquelle a été téléchargée un assemblage :

*Exemple 3-18 :* *Foo.cs*

```
using System.Reflection ;
using System.Windows.Forms ;
class Program {
   static void Main() {
      Assembly asm = Assembly.GetExecutingAssembly() ;
      MessageBox.Show("Mon CodeBase est:" + asm.CodeBase) ;
   }
}
```

L'ouverture d'*internet explorer* sur cette URL est illustrée par la figure suivante :


*Figure 3-14 : Exécution d'un assemblage à partir du web*

Vous remarquez qu'*internet explorer* ne vous a pas demandé votre permission pour télécharger et démarrer l'assemblage. En effet, le fait que l'exécutable soit un assemblage .NET a été détecté. Ainsi, *internet explorer* délègue la gestion de la sécurité au CLR. Le CLR est hébergé dans le processus de *internet explorer* par un hôte du moteur d'exécution particulier introduit en page 95.

L'exécution de l'assemblage `Foo.exe` ne requiert aucune permission sensible. Le CLR n'entrave pas son exécution car l'application des stratégies de sécurité permet par défaut l'exécution d'assemblages téléchargés à partir du web, tant qu'il ne demande pas de permissions particulières. Cependant, le CLR n'aurait pas permis à l'assemblage `Foo.exe` d'effectuer une action sensible, comme accéder à un fichier du disque dur ou modifier une clé de la base des registres, à moins bien sur, que la stratégie de sécurité accorde une certaine confiance au site `\url{www.smacchia.com}`. Un assemblage exécuté à partir du web peut ne pas avoir de nom fort.

## *Le cache de téléchargement*

Lorsqu'un assemblage est téléchargé pour la première fois à partir du web, il est automatiquement stocké dans le *cache de téléchargement* (*download cache* en anglais). Le cache de téléchargement est un répertoire entièrement géré par le CLR. Le cache de téléchargement est indépendant du cache d'*internet explorer* ainsi que des différents caches de la technologie *ClickOnce*. Les avantages du cache de téléchargement sont les suivant :

- Il évite de devoir télécharger un assemblage accessible à partir du web à chaque fois qu'il doit être exécuté, d'où la notion de cache.
- Il permet d'isoler physiquement les assemblages téléchargés du web des autres assemblages.

Le cache de téléchargement présente les deux différences suivantes avec le répertoire GAC :

- Le répertoire GAC est global à une machine. Il ne peut y avoir qu'un répertoire GAC par machine. En revanche, le CLR fait en sorte qu'il existe un cache de téléchargement par utilisateur d'une machine. Ainsi, il peut y avoir plusieurs caches de téléchargement sur une machine.
- Le CLR accorde une plus grande confiance (i.e accorde plus de permissions) aux assemblages contenus dans le répertoire GAC qu'aux assemblages contenus dans le cache de téléchargement.

Enfin, sachez que l'option `/ldl` de l'outil `gacutil.exe` vous permet de visualiser le contenu du cache de téléchargement :

```
C:>gacutil.exe /ldl
Microsoft (R) .NET Global Assembly Cache Utility.  Version 2.0.XXXXX
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.
The cache of downloaded files contains the following entries:
    Foo.exe, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null,
    Custom=null
Number of items = 1
```

Plus d'information sur la technologie *No-Touch Deployment* sont disponibles dans l'article **No-Touch Deployment in the .NET Framework** des **MSDN**.

# *Et si .NET n'est pas installé sur la machine cible ?*

Lors de la compilation d'un projet setup MSI le compilateur affiche cet avertissement :

```
WARNING: This setup does not contain the .NET Framework which must be
installed on the target machine by running dotnetfx.exe before this
setup will install. You can find dotnetfx.exe on the Visual Studio .NET
'Windows Components Update' media. Dotnetfx.exe can be redistributed
with your setup.
```

Cela signifie que lors de l'installation de l'application .NET, le *framework* .NET ne sera pas installé sur la machine cible. Toutes les applications .NET ont besoin que le *framework* .NET soit installé sur la machine pour pouvoir être exécutées.

L'avertissement nous apprend que pour installer le *framework* .NET sur la machine cible, il faut exécuter le fichier `dotnetfx.exe`. Ce fichier peut être redistribué librement.

- `dotnetfx.exe` se trouve dans le répertoire `\wcu\dotNetFramework\dotnetfx.exe` si vous avez obtenu .NET à partir d'un DVD.
- `dotnetfx.exe` se trouve dans le répertoire `\dotNetFramework\dotnetfx.exe` si vous avez obtenu .NET à partir d'un CD.
- `dotnetfx.exe` peut facilement être téléchargé à partir du site de *Microsoft*.

Ce fichier fait une taille d'environ 23 Mo. Il existe un tel fichier par version de .NET. Bien entendu, il faut que le *framework* .NET 2.0 soit installé sur une machine avant l'installation d'une application .NET, quelle que soit la technologie de déploiement utilisée. Dans le cas d'un déploiement avec la technologie *ClickOnce* nous avons vu que vous pouvez préciser en prérequis la présence du *framework* .NET. Sinon, il faut soit fournir `dotnetfx.exe` par exemple sur le CD distribué soit prévoir un lien pour permettre au client de le télécharger.

# 4

# *Le CLR (le moteur d'exécution des applications .NET)*

Le CLR (*Common Langage Runtime* en anglais et *moteur d'exécution* en français) est l'élément central de l'architecture de la plateforme .NET. Le CLR est une couche logicielle qui gère à l'exécution le code des applications .NET. Le mot « gérer » recouvre en fait une multitude d'actions nécessaires au bon déroulement de l'application. Listons en quelques-unes :

- hébergement de plusieurs applications dans un même processus *Windows* ;
- compilation du code IL en code machine ;
- gestion des exceptions ;
- destruction des objets devenus inutiles ;
- chargement des assemblages ;
- résolution des types.

Le CLR est conceptuellement proche de la couche logicielle communément nommé *machine virtuelle* en Java.

## *Les domaines d'application*

### *Notion de domaine d'application*

Un domaine d'application (*application domain* en anglais, en général nommé *AppDomain*) peut être vu comme un *processus léger*. Un processus *Windows* peut contenir plusieurs domaines

d'applications. La notion de domaine d'application est particulièrement utilisée pour qu'un même serveur physique héberge un grand nombre d'applications. Par exemple, la technologie ASP.NET utilise les domaines d'application pour héberger plusieurs applications web dans un même processus *Windows*. Les *stress test* de *Microsoft* créent jusqu'à 1000 applications web simples dans un même processus. Le gain de performance apporté par la notion de domaine d'application est double :

- La création d'un domaine d'application consomme beaucoup moins de ressources que la création d'un processus *Windows*.
- Les domaines d'application hébergés dans un même processus *Windows* partagent les ressources du processus telles que le CLR, les types de base de .NET, l'espace d'adressage ou les threads.

Lorsqu'un assemblage exécutable est démarré, le CLR crée automatiquement un *domaine d'application par défaut* pour l'exécuter. Chaque domaine d'application a un nom et le nom du domaine d'application par défaut est le nom du module principal de l'assemblage lancé (extension `.exe` comprise).

Si un même assemblage est chargé par plusieurs domaines d'application dans un même processus, il peut y avoir deux comportements :

- soit le CLR va charger plusieurs fois l'assemblage, une fois pour chaque domaine d'application du processus.
- soit le CLR va charger une seule fois l'assemblage hors de tous domaines d'application du processus. L'assemblage pourra néanmoins être utilisé par chaque domaine d'application du processus. On dit que l'assemblage est *domain neutral*.

On verra un peu plus loin dans ce chapitre que le choix de ce comportement est configurable. Le comportement par défaut est de charger plusieurs fois l'assemblage.

## *Domaines d'application et threads*

Ne confondez pas les termes « unité d'exécution » (les threads) et « unité d'isolation d'exécution » (les domaines d'application).

Il n'y a pas de notion d'appartenance entre les threads d'un processus et les domaines d'application de ce même processus. Rappelons que ce n'est pas le cas pour les processus, puisqu'un thread appartient à un seul processus, et chaque processus a un ou plusieurs threads. Concrètement un thread n'est pas confiné à un domaine d'application et à un instant donné plusieurs threads peuvent s'exécuter dans le contexte d'un même domaine d'application.

Soit deux domaines d'application `DA` et `DB` hébergés dans un même processus. Supposons qu'une méthode d'un objet `A`, dont l'assemblage contenant sa classe se trouve dans `DA`, appelle une méthode d'un objet `B`, dont l'assemblage contenant sa classe se trouve dans `DB`. Dans ce cas c'est le même thread qui exécute la méthode appelante et la méthode appelée. Ce thread traverse la frontière entre les domaines d'applications `DA` et `DB`.

Les concepts de thread et de domaine d'application sont donc orthogonaux.

## *Déchargement d'un domaine d'application*

Une fois qu'un assemblage est chargé dans un domaine d'application, vous n'avez pas la possibilité de le décharger de ce domaine. En revanche, vous pouvez décharger un domaine d'application tout entier. Cette opération est lourde de conséquences. Les threads en cours d'exécution dans le domaine d'application à décharger doivent être avortés par le CLR. Des problèmes se posent si certains d'entre eux sont en train d'exécuter du code non géré. Les objets gérés contenus dans le domaine d'application doivent être collectés.

Nous vous déconseillons d'avoir une architecture qui compte sur le fait de décharger régulièrement des domaines d'application. Nous verrons néanmoins que ce type d'architecture constitue parfois un mal nécessaire pour implémenter des serveurs qui nécessitent un grand taux de disponibilité (type 99,999% du temps comme *SQL Server 2005*).

## *L'isolation des domaines d'application*

L'isolation entre domaines application est le fruit des caractéristiques suivantes :

- Un domaine d'application peut être déchargé indépendamment des autres domaines applications.
- Un domaine d'application n'a pas d'accès direct aux assemblages et aux objets des autres domaines application.
- Un domaine d'application peut avoir sa propre stratégie de gestion d'exception. Du moment qu'il ne laisse pas « sortir » une exception hors de ses frontières, les problèmes d'un domaine d'application n'ont pas d'incidence sur les autres domaines d'application du même processus.
- Chaque domaine d'application peut définir sa propre stratégie de sécurité pour paramétrer le mécanisme CAS qu'utilise le CLR pour accorder des permissions au code d'un assemblage.
- Chaque domaine d'application peut définir ses propres règles pour paramétrer le mécanisme qu'utilise le CLR pour localiser ses assemblages avant de les charger.

## *La classe System.AppDomain*

Une instance de la classe `System.AppDomain` est une référence vers un domaine d'application du processus courant. La propriété statique `CurrentDomain{get;}` de cette classe vous permet de récupérer une référence vers le domaine d'application courant. L'exemple suivant illustre l'utilisation de cette classe pour énumérer les assemblages contenus dans le domaine d'application courant :

*Exemple 4-1 :*

```
using System ;
using System.Reflection ; // Pour la classe Assembly
class Program {
   static void Main() {
      AppDomain curAppDomain = AppDomain.CurrentDomain;
      foreach ( Assembly assembly in curAppDomain.GetAssemblies() )
         Console.WriteLine( assembly.FullName ) ;
   }
}
```

## *Lancer une nouvelle application dans le processus courant*

La classe `AppDomain` contient la méthode statique `CreateDomain()` qui permet de créer un nouveau domaine d'application dans le processus courant. Cette méthode est surchargée en plusieurs formes. Pour utiliser cette méthode, vous devez préciser :

- (obligatoire) Un nom pour le nouveau domaine d'application.
- (optionnel) Les règles de sécurité qui paramètrent le mécanisme CAS sur ce domaine d'application (par un objet de type `System.Security.Policy.Evidence`).
- (optionnel) Des informations qui paramètrent le mécanisme de localisation des assemblages de ce domaine d'application utilisé par le CLR (par un objet de type `System.AppDomainSetup`).

Les deux propriétés importantes d'un objet de type `System.AppDomainSetup` sont :

- `ApplicationBase` : Cette propriété définit le répertoire de base du domaine d'application. Ce répertoire est notamment utilisé par le mécanisme de localisation d'assemblages à charger dans ce domaine d'application.
- `ConfigurationFile` : Cette propriété référence un éventuel fichier de configuration du domaine d'application. Ce fichier doit être au format XML et contient des informations sur les règles de versionning et/ou la localisation des assemblages.

Maintenant que vous savez créer un domaine d'application, nous pouvons présenter comment charger et exécuter un assemblage exécutable dans un domaine en appelant la méthode `System.AppDomain.ExecuteAssembly()`. L'assemblage doit être de type exécutable et son exécution commence à son point d'entrée. C'est le thread qui appelle `ExecuteAssembly()` qui exécute le code de l'assemblage chargé. Cette constatation illustre le fait qu'un thread peut indifféremment traverser les frontières entre domaines d'application.

Voici un exemple de code en C#. Le premier code est l'assemblage qui va être chargé dans un domaine d'application par l'assemblage produit par la compilation du second code :

*Exemple 4-2 :* *`AssemblyACharger.exe`*

```
using System ;
using System.Threading ;
public class Program {
   public static void Main() {
      Console.WriteLine(
         "Thread:{0} Vous avez le bonjour du domaine : {1}",
         Thread.CurrentThread.Name,
         AppDomain.CurrentDomain.FriendlyName) ;
   }
}
```

*Exemple 4-3 :* *`AssemblyChargeur.exe`*

```
using System ;
using System.Threading ;
public class Program {
   public static void Main() {
```

```
      // Nomme le thread courant.
      Thread.CurrentThread.Name = "MonThread" ;
      // Crée un objet de type AppDomainSetup.
      AppDomainSetup info = new AppDomainSetup() ;
      info.ApplicationBase = "file:///"+ Environment.CurrentDirectory ;
      // Crée un domaine d'application sans paramètres de sécurité.
      AppDomain newDomaine = AppDomain.CreateDomain(
               "NouveauDomaine", null, info) ;
      Console.WriteLine(
  "Thread:{0} Appel à  ExecuteAssembly() à  partir du appdomain {1}",
         Thread.CurrentThread.Name,
         AppDomain.CurrentDomain.FriendlyName) ;
      // Charge l'assemblage 'AssemblyACharger.exe' dans
      // 'newDomain' puis l'exécute.
      newDomaine.ExecuteAssembly("AssemblyACharger.exe") ;
      // Décharge le nouveau domaine
      // ainsi que l'assemblage qu'il contient.
      AppDomain.Unload(newDomaine) ;
   }
}
```

Cet exemple affiche :

```
Thread:MonThread Appel à ExecuteAssembly() à partir du appdomain
AssemblyChargeur.exe
Thread:MonThread Vous avez le bonjour du domaine : NouveauDomaine
```

Notez la nécessité de l'ajout `"file:///"` pour préciser que l'assemblage se trouve en local. Si ce n'était pas le cas on aurait pu utiliser `"http:///"`. Dans ce cas, l'assemblage aurait pu être chargé à partir du web.

Cet exemple illustre aussi que le nom du domaine d'application par défaut est le nom du module principal de l'assemblage lancé (ici `AssemblyChargeur.exe`).

## *Exécuter du code dans le contexte d'un domaine d'application*

Grâce à la méthode d'instance `AppDomain.DoCallBack()` vous avez la possibilité d'exécuter du code d'un assemblage du domaine d'application courant dans le contexte d'un autre domaine d'application. Pour cela, ce code doit être contenu dans une méthode que vous référencez à l'aide d'un délégué de type `System.CrossAppDomainDelegate`. Tout ceci est illustré par l'exemple suivant :

*Exemple 4-4 :*

```
using System ;
using System.Threading ;
public class Program {
   public static void Main() {
      Thread.CurrentThread.Name = "MonThread" ;
      AppDomain newDomaine = AppDomain.CreateDomain(
               "NouveauDomaine") ;
```

```
        CrossAppDomainDelegate deleg = new CrossAppDomainDelegate(Fct);
        newDomaine.DoCallBack(deleg);
        AppDomain.Unload(newDomaine) ;
    }
    public static void Fct() {
        Console.WriteLine(
            "Thread:{0} Fct() exécutée dans  {1}",
            Thread.CurrentThread.Name,
            AppDomain.CurrentDomain.FriendlyName) ;
    }
}
```

Cet exemple affiche :

```
Thread:MonThread Fct() exécutée dans  NouveauDomaine
```

Soyez conscient que cette possibilité « d'injecter » du code dans un domaine d'application peut provoquer la levée d'une exception de sécurité si vous n'avez pas les droits suffisants.

## *Les événements d'un domaine d'application*

La classe `AppDomain` présente les événements suivants :

| Événement | Description |
|---|---|
| `AssemblyLoad` | Déclenché lorsqu'un assemblage vient d'être chargé. |
| `AssemblyResolve` | Déclenché lorsqu'un assemblage à charger n'a pu être trouvé. |
| `DomainUnload` | Déclenché lorsque le domaine d'application est sur le point d'être déchargé. |
| `ProcessExit` | Déclenché lorsque le processus se termine (déclenché avant `DomainUnload`). |
| `ReflectionOnlyPre-BindAssemblyResolve` | Déclenché juste avant le chargement d'un assemblage destiné à être utilisé par le mécanisme de réflexion. |
| `ResourceResolve` | Déclenché lorsqu'une ressource n'a pu être trouvée. |
| `TypeResolve` | Déclenché lorsqu'un type n'a pu être trouvé. |
| `UnhandledException` | Déclenché lorsqu'une exception n'est pas rattrapée dans le code du domaine d'application. |

Certains de ces événements peuvent être utilisés pour remédier au problème qui l'a déclenché. L'exemple suivant illustre comment exploiter l'événement `AssemblyResolve` pour faire en sorte de charger un assemblage à partir d'un emplacement qui n'a pas été pris en compte par le mécanisme de localisation des assemblages du CLR :

*Exemple 4-5 :*

```
using System ;
using System.Reflection ;  // pour Assembly
public class Program {
   public static void Main() {
      AppDomain.CurrentDomain.AssemblyResolve += AssemblyResolve ;
      Assembly.Load("AssemblyACharger.dll");
   }
   public static Assembly AssemblyResolve(object sender,
                                          ResolveEventArgs e) {
      Console.WriteLine("Assemblage {0} non trouvé : ", e.Name) ;
      return Assembly.LoadFrom(@"C:\AppDir\CetAssemblyACharger.dll");
   }
}
```

Si la seconde tentative de chargement marche, aucune exception n'est lancée. Le nom de l'assemblage chargé à la seconde tentative n'est pas nécessairement le même que celui que l'on a essayé de charger à la première tentative.

En page 519 nous présentons un programme exploitant l'événement `UnhandledException`.

En page 80 nous expliquons que les classes de l'espace de nom `System.Deployment` peuvent avoir recours au événements `AssemblyResolve` et `ResourceResolve` afin de télécharger dynamiquement un groupe de fichier lorsqu'une application déployée avec la technologie *ClickOnce* en a besoin pour la première fois.

## *Echanger des informations entre domaines*

Vous avez la possibilité de stocker des données dans un domaine d'application grâce aux méthodes `SetData()` et `GetData()` de la classe `AppDomain`. Comme le montre l'exemple suivant, chacune de ces données est indexée par une chaîne de caractères :

*Exemple 4-6 :*

```
using System ;
using System.Threading ;
public class Program {
   public static void Main() {
      AppDomain newDomaine = AppDomain.CreateDomain(
               "NouveauDomaine") ;
      CrossAppDomainDelegate deleg = new CrossAppDomainDelegate(Fct) ;
      newDomaine.DoCallBack(deleg) ;
      // Récupère dans l'appDomain par défaut l'entier indexé par la
      // string "UnEntier" dans l'appDomain 'NouveauDomaine'.
      int unEntier = (int) newDomaine.GetData("UnEntier");
      AppDomain.Unload(newDomaine) ;
   }
   public static void Fct() {
      // Cette méthode s'exécute dans l'appDomain 'NouveauDomaine'.
      // Indexe la valeur '691' par la string "UnEntier".
      AppDomain.CurrentDomain.SetData("UnEntier", 691);
```

```
    }
}
```

Cet exemple présente le cas simple car nous stockons un entier qui est un type valeur connu de tous les domaines d'application. La technologie *.NET Remoting* qui fait l'objet du chapitre 22 permet de partager des objets entre domaines d'application d'une manière plus évoluée mais plus complexe.

# *Chargement du CLR dans un processus grâce à l'hôte du moteur d'exécution*

## *Les DLLs mscorsvr.dll mscorwks.dll*

Chaque version du CLR est publiée avec deux DLLs :

- La DLL `mscorsvr.dll` contient une version du CLR spécialement optimisée pour les machines ayant plusieurs processeurs (« svr » est employé pour « serveur »).
- La DLL `mscorwks.dll` contient une version du CLR spécialement optimisée pour les machines ayant un seul processeur (« wks » est employé pour « workstation » ou « station de travail »).

Ces deux DLLs ne sont pas des assemblages. En conséquence, elles ne contiennent pas de code IL, et elles ne peuvent être analysées avec l'outil `ildasm.exe`. Chaque processus exécutant une ou plusieurs applications .NET contient une de ces deux DLLs. On dit que le processus héberge le CLR. Nous allons expliquer ici comment le chargement dans le processus d'une de ces DLLs se déroule.

## *L'assemblage mscorlib.dll*

Une autre DLL joue un rôle prépondérant dans l'exécution des applications .NET. C'est la DLL `mscorlib.dll` qui est en fait l'unique module de l'assemblage du même nom. Cet assemblage contient les implémentations des types de base du *framework* .NET (comme la classe `System.String`, la classe `System.Object` ou la classe `System.Int32`). Cet assemblage est référencé par tous les assemblages .NET. Cette référence est créée automatiquement par tous les compilateurs qui produisent du code IL. Il est intéressant d'analyser l'assemblage `mscorlib` avec l'outil `ildasm.exe`. Nous précisons que l'assemblage `mscorlib` réside à l'exécution hors de tous domaines d'application. En outre, il ne peut être chargé/déchargé qu'une fois durant la vie d'un processus.

## *Notion d'hôte du moteur d'exécution*

Le fait que .NET ne soit encore intégré à aucun système d'exploitation entraîne que le chargement du CLR à la création d'un processus, doit être pris en charge par le *processus* lui-même.

La tâche de charger le CLR dans le processus incombe à une entité appelée *l'hôte du moteur d'exécution* (*runtime host* en anglais). L'hôte du moteur d'exécution étant là pour charger le CLR, une partie de son code est forcément non géré, puisque c'est le CLR qui gère le code. Cette partie s'occupe de charger le CLR, de le configurer, et de passer le thread courant en mode géré.

Une fois le CLR chargé dans le processus, l'hôte du moteur d'exécution a d'autres responsabilités telles que la décision à prendre lorsqu'une exception n'est pas rattrapée. La figure suivante illustre les différentes couches de cette architecture. On voit que le CLR et l'hôte s'échangent des informations grâce à une API :

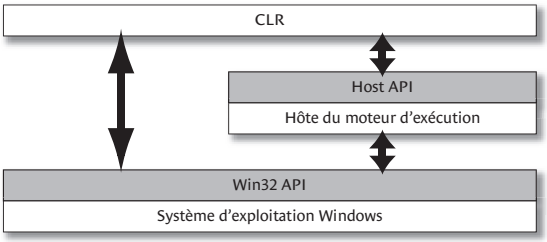


*Figure 4-1 : Découpage en couches de l'hébergement du CLR*

Il existe plusieurs hôtes du moteur d'exécution et vous avez même la possibilité de créer vos propres hôtes du moteur d'exécution. Le choix de tel ou tel hôte du moteur d'exécution pour une application a un impact sur les performances de l'application et l'étendue des fonctionnalités utilisables par l'application. Les hôtes du moteur d'exécution déjà existants fournis par *Microsoft* sont :

- L'hôte du moteur d'exécution des applications **Console** et **Winform** : L'assemblage exécutable est chargé dans le domaine d'application par défaut. Lorsqu'un assemblage est chargé implicitement, il est chargé dans le même domaine d'application que l'assemblage qui le sollicite. En général ce type d'application n'a pas à utiliser d'autres domaines d'application que le domaine d'application par défaut.
- L'hôte du moteur d'exécution **ASP.NET** : Crée un domaine d'application pour chaque application web. Une application web est identifiée par son répertoire virtuel racine ASP.NET. Si une requête web s'adresse à une application déjà chargée dans un domaine, la requête est automatiquement routée vers ce domaine par l'hôte.
- L'hôte du moteur d'exécution **Microsoft Internet Explorer** : Par défaut, il crée un domaine d'application par site web visité. Ainsi les assemblages de différents sites peuvent s'exécuter avec différents niveaux de sécurité. Le CLR n'est chargé que lorsque *Internet Explorer* a besoin d'exécuter un assemblage pour la première fois. Plus d'informations au sujet de cet hôte sont disponibles en page 84.
- L'hôte du moteur d'exécution de **SQL Server 2005** : Les requêtes vers la base peuvent être écrites en langage IL. Le CLR n'est chargé que la première fois qu'une telle requête doit être effectuée. Un domaine d'application est créé pour chaque couple utilisateur/base de données. Dans le chapitre courant, nous aurons l'occasion de revenir sur les possibilités et propriétés de cet hôte très particulier introduit avec .NET 2.0.

## *Plusieurs versions du CLR sur la même machine*

`mscorlib` étant un assemblage à nom fort, plusieurs versions peuvent cohabiter « côte à côte » sur la même machine. De plus, toutes les versions du CLR sont dans le répertoire « `%windir%\Microsoft.NET\Framework` ». Tous les fichiers concernant une version du *framework*

.NET sont dans un sous répertoire portant le numéro de version. Comme il existe un répertoire par version du *framework* .NET installée sur la machine, il peut y avoir aussi plusieurs versions des DLLs `mscorsvr.dll` et `mscorwks.dll` sur la même machine. Cependant une seule version du CLR peut être chargée et hébergée par chaque processus.

Le fait d'avoir potentiellement plusieurs versions du CLR entraîne l'existence d'une petite couche logicielle qui prend en paramètre la version désirée du CLR et la charge. Ce code est appelé *cale* (*shim* en anglais) et est stocké dans la DLL `mscoree.dll` (*MSCOREE* veut dire *Microsoft Component Object Runtime Execution Engine*).

Il ne peut y avoir qu'une seule DLL cale par machine. La cale est directement appelée par l'hôte du moteur d'exécution par la fonction `CorBindToRuntimeEx()`. La DLL `mscoree.dll` contient des interfaces et des classes COM. La fonction `CorBindToRuntimeEx()` crée un objet COM, instance de la classe COM `CorRuntimeHost`. C'est cet objet qui va s'interfacer avec le CLR. Pour manipuler cet objet la fonction `CorBindToRuntimeEx()` retourne l'interface COM `ICLRRuntimeHost`.

L'appel à `CorBindToRuntimeEx()` pour créer l'objet COM s'interfaçant avec le CLR, transgresse des règles fondamentales de COM : il ne faut pas appeler la fonction `CoCreateInstance()` pour créer un objet COM. De plus, les appels aux méthodes `AddRef()` et `Release()` sur l'interface `ICLRRuntimeHost` n'ont pas d'effet.

## *Chargement du CLR avec CorBindToRuntimeEx()*

Voici le prototype de la fonction `CorBindToRuntimeEx()` qui charge la DLL cale, qui charge à son tour la DLL contenant le CLR :

```
HRESULT CorBindToRuntimeEx(
   LPWSTR      pwszVersion,
   LPWSTR      pwszBuildFlavor,
   DWORD       flags,
   REFCLSID    rclsid,
   REFIID      riid,
   LPVOID *    ppv) ;
```

Ce prototype est spécifié dans le fichier `mscoree.h` et le code de cette fonction est dans la DLL cale `mscoree.dll`.

- `PwszVersion` : Le numéro de version du CLR sous la forme d'une chaîne de caractères commençant par « v » (exemple `"v2.0.50727"`). Si cette chaîne n'est pas précisée (i.e si on passe un pointeur nul), la version la plus récente disponible du CLR sera alors choisie.
- `PwszBuildFlavor` : Ce paramètre indique si l'on souhaite charger le CLR pour workstation (`mscorwks.dll`) en précisant la chaîne de caractères `"wks"` ou le CLR pour serveur (`mscorsvr.dll`) en précisant la chaîne de caractères `"svr"`. Si vous n'avez qu'un seul processeur, le CLR pour workstation sera chargé quelle que soit la valeur de ce paramètre. *Microsoft* prévoit d'autres types de CLR, donc d'autres valeurs pour ce paramètre.
- `flags` : Ce paramètre est constitué par un ensemble de drapeaux.
  En utilisant le drapeau `STARTUP_CONCURRENT_GC` on indique que l'on souhaite que le *ramasse-miettes* soit exécuté en mode concurrent, dit aussi mode simultané. Ce mode permet à un thread de réaliser une bonne partie du travail du ramasse-miettes sans avoir à interrompre les autres threads de l'application. Dans le cas d'une exécution non simultanée

du ramasse-miettes, le CLR utilise régulièrement les threads de l'application pour effectuer le travail du ramasse-miettes. Les performances globales du mode non concurrent sont meilleures. En revanche le mode concurrent est souhaitable dans le cas d'interfaces utilisateurs, car il permet une plus grande réactivité de l'interface.

On peut aussi positionner d'autres drapeaux pour indiquer que l'on souhaite que les assemblages soient chargés d'une manière neutre par rapport aux domaines (*loaded domain-neutrally* en anglais) ou non. Cela signifie que toutes les parties en lecture des assemblages (le code, les structures...) ne seront présentes physiquement qu'une seule fois dans le processus, même si plusieurs domaines d'application chargent le même assemblage (dans le même esprit que le *mapping* des DLLs entre processus sous les systèmes d'exploitation *Windows*). Un assemblage chargé d'une manière neutre par rapport aux domaines n'appartient donc pas à un domaine spécifique. Cela présente le principal inconvénient qu'il ne peut pas être déchargé. Il faut donc détruire puis relancer tout un processus lors de la mise à jour d'un tel assemblage. En outre il faut que le même ensemble de permissions lui soit accordé par la stratégie de sécurité de chaque domaine d'application sinon il est chargé plusieurs fois. Chaque constructeur de classe d'un assemblage chargé d'une manière neutre est invoqué pour chaque domaine d'application et il existe une copie de chaque champ statique de chaque classe pour chaque domaine d'application. En interne cela est possible grâce à une table d'indirection gérée par le CLR. Ceci implique une petite baisse des performances. Cependant, charger un assemblage d'une manière neutre par rapport aux domaines, permet de ne le charger qu'une seule fois. En conséquence, cette pratique est plus économique en terme de consommation de la mémoire, d'où un effet positif sur les performances. En outre, chacune des méthodes d'un tel assemblage n'est compilée par le compilateur JIT qu'une seule fois.

Vous avez trois drapeaux possibles :

- `STARTUP_LOADER_OPTIMIZATION_SINGLE_DOMAIN` : Aucun assemblage n'est chargé d'une manière neutre par rapport aux domaines. C'est ce comportement qui est pris par défaut.
- `STARTUP_LOADER_OPTIMIZATION_MULTI_DOMAIN` : Tous les assemblages sont chargés d'une manière neutre par rapport aux domaines. Aucun hôte du moteur d'exécution n'utilise cette option à ce jour.
- `STARTUP_LOADER_OPTIMIZATION_MULTI_DOMAIN_HOST` : Seuls les assemblages partagés contenus dans le GAC sont chargés d'une manière neutre par rapport aux domaines.

L'assemblage `mscorlib` a un traitement spécial puisqu'il est chargé d'une manière neutre une seule fois, quelle que soit la valeur de ce paramètre.

- `rclsid` : Le classe ID (`CLSID`) de la classe COM (`coclass`) qui implémente l'interface que vous cherchez. Seules les valeurs `CLSID_CorRuntimeHost`, `CLSID_CLRRuntimeHost` ou `null` sont acceptées. La deuxième valeur est apparue avec la version 2.0 de .NET car de nouvelles fonctionnalités dues à l'hébergement du CLR dans le processus de *SQL Server 2005* ont nécessité une nouvelle interface et une nouvelle classe COM.
- `pwszBuildFlavor` : L'interface ID (`IID`) de l'interface COM dont vous avez besoin. Seules les valeurs `IID_CorRuntimeHost`, `IID_CLRRuntimeHost` ou `null` sont acceptées
- `ppv` : Un pointeur vers l'interface COM retournée, de type `ICorRuntimeHost` ou `IClrRuntimeHost` selon ce qui a été demandé.

La cale ne fait que charger le CLR dans le processus. Le cycle de vie du CLR est contrôlé par la partie non gérée du code de l'hôte du moteur d'exécution grâce à l'interface COM `ICLRRuntimeHost`

renvoyée par la fonction `CorBindToRuntimeEx()`. Cette interface présente entre autres deux méthodes `Start()` et `Stop()` dont les noms illustrent leurs actions.

## *Exemple de code C++ non géré d'un hôte du moteur d'exécution propriétaire*

Dans la très grande majorité des projets, vous n'aurez pas à créer votre hôte du moteur d'exécution. Cependant, il est rassurant de savoir que ceci est possible et instructif d'en avoir déjà vu un.

Pour être bien compris, ce code nécessite une bonne connaissance de la technologie COM. N'oublions pas que les systèmes d'exploitation type *Windows 2000/XP* utilisent encore COM comme mécanisme de communication/modélisation. Le CLR présente la possibilité d'être configuré et manipulé par du code non géré, par l'intermédiaire d'interfaces COM. Parmi ces interfaces COM, l'interface `ICCLRRuntimeHost` joue un rôle prépondérant. Nous avons vu un peu plus haut qu'une telle interface peut être obtenue en appelant la fonction `CorBindToRuntimeEx()`. Voici donc le code C++ du plus basique des hôtes du moteur d'exécution :

*Exemple 4-7 :*

```
// Pour compiler ce code, il faut vous lier avec
// la bibliothèque statique mscoree.lib.
#include <mscoree.h>

// L'import doit se faire sur une seule ligne.
#import <mscorlib.tlb> raw_interfaces_only ...
   ... high_property_prefixes("_get","_put","_putref") ...
      ... rename("ReportEvent","ReportEventGéré") rename_namespace("CRL")

// Utilise l'espace de nom ComRuntimeLibrary.
using namespace CRL ;
ICLRRuntimeHost * pClrHost = NULL ;

void main (void){
   // Obtient une interface COM ICorRuntimeHost sur le CLR.
   HRESULT hr = CorBindToRuntimeEx(
               NULL, // On demande la dernière version du CLR.
               NULL, // On demande la version workstation du CLR.
               0,
               CLSID_CLRRuntimeHost,
               IID_ICLRRuntimeHost,
               (LPVOID *) &pClrHost) ;
   if (FAILED(hr)){
      printf("Echec de l'obtention du ptr ICLRRuntimeHost !") ;
      return ;
   }
   printf("Pointeur ICCLRRuntimeHost obtenu.\n") ;
   printf("Lance le CLR.\n") ;
   pClrHost->Start();
```

```
   // Ici, on peut utiliser notre pointeur COM sur le CLR.

   pClrHost->Stop();
   printf("CLR stoppé.\n") ;
   pClrHost->Release();
   printf("Ciao !\n") ;
}
```

Supposons que nous ayons un assemblage `MyManagedLib.dll` stocké dans le répertoire `C:\Test` compilé à partir de ce code :

*Exemple 4-8 :* *`MyManagedLib.cs`*

```
namespace MyProgramNamespace {
   public class MyClass {
      public static int MyMethod(string s) {
         System.Console.WriteLine(s) ;
         return 0 ;
      }
   }
}
```

Vous pouvez facilement invoquer la méthode `Main()` à partir de notre hôte comme ceci :

*Exemple 4-9 :*

```
...
   pClrHost->Start() ;
   DWORD retVal=0 ;
   hr = pClrHost->ExecuteInDefaultAppDomain(
     L"C:\\test\\MyManagedLib.dll", // Chemin + Asm.
     L"MyProgramNamespace.MyClass", // Nom entier du type.
     L"MyMethod",                   // Nom de la méthode elle doit avoir
                                    //   la signature int XXX(string).
     L"Hello from host!",           // Chaîne de caractères en argument.
     &retVal) ;                     // Valeur OUT de retour.

   pClrHost->Stop() ;
...
```

## *Modifier la configuration du CLR à partir d'un hôte du moteur d'exécution propriétaire*

Il faut savoir que lorsque le CLR est vu comme un objet COM, l'interface `ICLRRuntimeHost` n'est pas la seule interface présentée par cet objet COM. Vous pouvez en fait manipuler le CLR à travers les interfaces COM suivantes :

- `ICLRRuntimeHost` permet de créer et de décharger des domaines d'applications, de gérer le cycle de vie du CLR et de créer des preuves pour le mécanisme de sécurité CAS.

- `ICorConfiguration` permet de spécifier au CLR certaines interfaces *callback* pour pouvoir être averti de certains événements. Ces interfaces *callback* sont : `IGCThreadControl IGCHostControl IDebuggerThreadControl`. Les événements présentés par ces interfaces sont beaucoup moins fins que ceux présentés par l'API de profiling du CLR, présentée un peu plus bas.
- `ICorThreadPool` permet de manipuler le pool de threads .NET du processus et de modifier certains paramètres de configuration.
- `IGCHost` permet d'obtenir des informations sur le fonctionnement du ramasse-miettes et de modifier certains paramètres de configuration.
- `IValidator` permet de valider l'entête PE/COFF des assemblages (utilisé notamment par l'outil `peverify.exe`).
- `IMetaDataConverter` permet de convertir les métadonnées COM (i.e tlb/tlh) en méta données .NET (utilisé notamment par l'outil `tlbexp.exe`).

Pour savoir quelles sont les méthodes présentées par ces interfaces, il suffit d'observer les fichiers `mscoree.h`, `ivalidator.h` et `gchost.h`. Pour obtenir une de ces interfaces à partir de votre interface `IClrRuntimeHost`, il suffit d'utiliser la fameuse méthode `QueryInterface()` comme ceci :

```
...
   ICorThreadpool *   pThreadPool   = NULL ;
   hr = pClrHost->QueryInterface( IID_ICorThreadpool,
                                  (void**)&pThreadPool);
...
```

## *Spécificités de l'hôte du moteur d'exécution de SQL Server 2005*

Comme nous l'avons mentionné, l'hôte du moteur d'exécution de *SQL Server 2005* est très particulier du fait des contraintes de fiabilité, de sécurité et de performance hors normes imposées par un tel SGBD.

La contrainte prépondérante de ce type de serveur est la fiabilité. Les trois mécanismes *région d'exécution contrainte* (CER), *finaliseur critique* et *région critique* (CR) ont été ajoutés au CLR pour renforcer la fiabilité d'une application .NET. Ils font l'objet de la section 4 « Facilités fournies par le CLR pour rendre votre code plus fiable », page 125 du présent chapitre.

La seconde contrainte est la sécurité. Pour éviter que du code malveillant utilisateur soit chargé par inadvertance, tous les assemblages utilisateurs sont chargés par l'hôte du moteur d'exécution à partir de la base de données. Cela implique une phase de préchargement des assemblages dans la base par l'administrateur DB (le DBA). Durant cette phase, le DBA peut préciser que l'assemblage appartient à une des catégories `SAFE`, `EXTERNAL_ACCESS` et `UNSAFE` selon le niveau de confiance qu'il accorde à son code. L'ensemble des permissions qui sera accordé à l'assemblage à l'exécution par le mécanisme CAS est fonction de cette catégorie et donc, de ce niveau de confiance. Enfin, certaines fonctionnalités du *framework* .NET considérées comme sensibles, telles que certaines classes de l'espace de noms `System.Threading`, ne sont pas exploitables à partir d'un assemblage chargé dans *SQL Server 2005*.

La troisième contrainte est la performance. On la satisfait en exploitant les ressources d'une manière optimale. Dans ce contexte, les ressources les plus précieuses sont les threads et les pages mémoires chargées en mémoires vive. L'idée est de minimiser le nombre de *context switching*

entre threads et de minimiser le nombre de pages stockées en mémoire virtuelle sur le disque dur pour profiter au mieux de la mémoire vive disponible.

Les *context switching* sont normalement gérés par le mécanisme de *multitâche préemptif* du *répartiteur* de *Windows*, décrit en page 138. L'hôte du moteur d'exécution de *SQL Server 2005* implémente sont propre mécanisme de multitâche plutôt basé sur un modèle de *multitâche coopératif*. Dans ce modèle, ce sont les threads eux mêmes qui décident du moment où le processeur peut passer à un autre thread. Un avantage est que les choix de ces moments sont plus fins, car liés à la sémantique des traitements. Il en résulte une gestion globale plus efficace des threads. Un autre avantage est que ce modèle est adapté à l'utilisation du mécanisme de *fibre* de *Windows* (*fiber* en anglais).

Une fibre est un *thread logique* qualifié aussi de *thread léger*. Un même thread physique *Windows* peut enchaîner l'exécution de différentes fibres. L'avantage est que le passage d'une fibre à une autre est une opération beaucoup moins coûteuse que le *context switching*. En contrepartie, lorsque le mode fibre est utilisé par l'hôte du moteur d'exécution *de SQL Server 2005*, on perd la garantie d'une relation biunivoque entre les threads physiques *Windows* et les threads gérés .NET. Un même thread géré n'est plus forcément exécuté par le même thread physique durant son toute son existence. Il faut donc absolument s'affranchir de tout type d'affinité entre ces deux entités. Parmi les types d'affinités possibles entre un thread géré et son thread physique sous-jacent on peut citer les *Thread Local Storage*, la culture courante et les objets de synchronisation *Windows* qui dérivent de la classe `WaitHandle` style mutex, sémaphore ou événement. Sachez que vous avez la possibilité de communiquer à l'hôte du moteur d'exécution le commencement et la fin d'une région de code qui exploite ce type d'affinité avec les méthode `BeginThreadAffinity()` et `EndThreadAffinity()` de la classe `Thread`. Ce dernier saura alors désactiver temporairement le mode fibre, auquel on attribut un facteur d'optimisation de 20% des performances. (Note : *Dans la version actuelle de SQL Server 2005, le mode fibre a été retiré car les ingénieurs de Microsoft n'étaient pas certains de la fiabilité de ce mode. Ce mode sera réintroduit dans les versions ultérieures de ce produit*).

Le stockage des pages mémoires est normalement géré par le mécanisme de mémoire virtuelle de *Windows* décrit en page 134. L'hôte du moteur d'exécution de *SQL Server 2005* s'intercale entre les demandes mémoire du CLR et ce mécanisme afin de profiter au mieux de la mémoire vive disponible. Cela permet aussi d'obtenir un comportement prévisible lorsqu'une demande d'allocation mémoire du CLR échoue. Comme nous le verrons plus tard, cette particularité est essentielle pour assurer la fiabilité de serveurs tels que *SQL Server 2005*.

Toutes ces nouvelles possibilités sont accessibles grâce à une API qui permet au CLR et à son hôte de dialoguer. Une trentaine de nouvelles interfaces ont été prévues. Elles sont listées un peu plus bas. Il incombe à l'hôte de fournir un objet qui implémente l'interface `IHostControl` au moyen de la méthode `ICLRRuntimeHost.SetHostControl(IHostControl*)`. Cette interface présente la méthode `GetHostManager(IID,[out]obj)`. Le CLR appelle cette méthode pour obtenir un objet de l'hôte auquel il va déléguer une responsabilité telles que la gestion des threads ou le chargement des assemblages. Plus d'information à ce sujet sont disponibles en analysant les interfaces suivantes dans le fichier `mscoree.h`.

| Responsabilité | Interfaces implémentées par l'hôte. | Interfaces implémentées par le CLR. |
|---|---|---|
| Chargement des assemblages | IHostAssemblyManager IHostAssemblyStore | ICLRAssemblyReferenceList ICLRAssemblyIdentityManager |
| Sécurité | IHostSecurityManager IHostSecurityContext | **ICLRHostProtectionManager** |
| Gestion des échecs | IHostPolicyManager | ICLRPolicyManager |
| Gestion de la mémoire | IHostMemoryManager IHostMalloc | ICLRMemoryNotificationCallback |
| Ramasse-miettes | IHostGCManager | ICLRGCManager |
| Threading | IHostTaskManager IHostTask | ICLRTaskManager ICLRTask |
| Pool de threads | IHostThreadPoolManager | |
| Synchronisation | IHostSyncManager IHostCriticalSection IHostManualEvent IHostAutoEvent IHostSemaphore | ICLRSyncManager |
| I/O Completion | IHostIoCompletionManager | ICLRIoCompletionManager |
| Débogage | | **ICLRDebugManager** |
| Événements du CLR | IActionOnCLREvent | **ICLROnEventManager** |

# *Profiler vos applications*

Cette section à pour but de vous sensibiliser à une fonctionnalité particulièrement utile du CLR : la possibilité de profiler très finement son exécution. En d'autres termes, vous pouvez demander au CLR d'exécuter une de vos méthodes non gérées lorsqu'un événement particulier survient tel que le commencement de la compilation JIT d'une méthode ou à la fin du chargement d'un assemblage. Il est assez logique que ces callbacks soient des méthodes non gérées. En effet, ces méthodes sont censées nous permettre d'observer l'état du CLR donc il vaut mieux que ce ne soit pas ce dernier qui prenne en charge leurs exécutions.

Pour exploiter le *profiling* du CLR il faut que vous construisiez une classe COM implémentant l'interface `ICorProfilerCallback`. Cette interface est définie dans le fichier `corprof.h`qui se trouve dans le répertoire `SDK\v2.0\Include` de l'installation de *Visual Studio*. Elle contient environs 70 méthodes. Une fois l'interface `ICorProfilerCallback` implémentée, il faut que le CLR sache que c'est cette implémentation qu'il doit utiliser pour réaliser les callbacks. Pour communiquer le CLSID de cette implémentation au CLR, vous n'avez pas besoin de créer votre hôte du

moteur d'exécution. Il suffit de positionner correctement les deux variables d'environnement `Cor_Enable_Profiling` et `Cor_Profiler`. `Cor_Enable_Profiling` doit être positionnée à une valeur non nulle pour spécifier que le CLR doit réaliser les callbacks. `Cor_Profiler` doit être positionnée avec le `CLSID` ou le `ProgID` de l'implémentation de `ICorProfilerCallback`.

Nous n'allons pas détailler cette possibilité car l'article **The .NET Profiling API and the DNProfiling Tool** de *Matt Pietrek* du numéro de *Décembre 2001* de **MSDN Magazine** (disponible gratuitement en ligne) le fait déjà. Nous vous conseillons vivement de télécharger le code de cet article est de faire quelques essais sur vos propres applications .NET. Vous prendrez ainsi toute la mesure de la quantité de tâches réalisées par le CLR ! Le code fourni avec cet article a aussi l'avantage de montrer comment utiliser un masque pour n'avoir accès qu'à certaines catégories de callback. La manipulation nécessaire pour utiliser ce code est très facile :

- Enregistrez la classe COM sur votre machine ;
- ouvrez une fenêtre de commande ;
- positionnez les variables d'environnement en exécutant le fichier batch fourni ;
- lancez votre application à partir de cette fenêtre de commande.

# *Localisation et chargement des assemblages à l'exécution*

Que l'on applique la stratégie de déploiement des assemblages à titre privé, ou la stratégie de déploiement des assemblages partagés, c'est le CLR qui localise et charge les assemblages à l'exécution. Plus exactement, ces tâches incombent au sous système *chargeur d'assemblages* (*assembly loader* en anglais*)* du CLR plus communément nommé *fusion*. L'idée générale est que le processus de localisation des assemblages par le CLR est configurable et intelligent.

- Configurable dans le sens où un administrateur peut facilement déplacer des assemblages tout en permettant qu'ils soient encore localisables. Configurable aussi dans le sens où l'on peut rediriger l'utilisation de certaines versions vers d'autres versions (de deux façons différentes, avec les assemblages de stratégie d'éditeur, ou avec un fichier de configuration présenté plus loin).
- Intelligent dans le sens où lorsque le CLR ne trouve pas un assemblage dans un répertoire, il applique un algorithme qui lui permet, par exemple, d'aller chercher dans les sous répertoires qui ont le même nom que l'assemblage. Intelligent aussi dans le sens où si une application marchait mais ne marche plus à cause d'un assemblage qui n'est plus localisable, on puisse revenir très simplement en arrière.

Ces deux contraintes sont satisfaites grâce à un l'algorithme de localisation de *fusion*.

Nous avons vu page 15 qu'un assemblage pouvait être constitué de plusieurs fichiers nommés modules. Ici, nous parlons du processus de localisation d'un assemblage, c'est-à-dire du processus de localisation du module principal d'un assemblage, (celui avec le manifeste). Rappelons que tous les modules d'un même assemblage doivent se trouver impérativement dans le même répertoire.

## Quand le processus de localisation est-il démarré ?

Le processus de localisation du CLR est souvent sollicité lorsque le code d'un assemblage en cours d'exécution à besoin de charger un autre assemblage. Si la localisation échoue, elle se solde par l'envoi de l'exception `System.IO.FileNotFoundException`. Le processus de localisation est utilisé lors de :

- L'utilisation d'une des surcharges de la méthode `AppDomain.Load()` qui charge un assemblage dans le domaine d'application sur lequel est appelée la méthode.
- Le chargement implicite par le CLR d'un assemblage. Ceci est décrit dans la prochaine section lorsque nous expliquons comment le CLR résout les types.

Cet algorithme n'est pas utilisé lors de :

- L'utilisation de la méthode `AppDomain.ExecuteAssembly()` que nous avons vu au début de ce chapitre lors de la présentation des domaines d'application.
- L'utilisation de la méthode statique `Assembly.LoadFrom()` qui accepte le chemin (absolu ou relatif à partir du répertoire d'exécution de l'application) et le nom de l'assemblage.

Remarquez que ces deux méthodes ne fonctionnent pas selon la philosophie .NET puisqu'elles prennent en argument un chemin vers un fichier et non le nom d'un assemblage. Il est préférable de ne jamais utiliser `LoadFrom()` qui peut toujours être remplacée par `Load()`. En revanche l'utilisation très simple de la méthode `AppDomain.ExecuteAssembly()` peut s'avérer être un raccourci efficace.

## L'algorithme de localisation

### Recherche dans le répertoire GAC

L'algorithme va d'abord aller chercher l'assemblage dans le répertoire GAC, à condition que le nom de l'assemblage fourni soit un nom fort. Lors de la recherche dans le répertoire GAC, l'utilisateur de l'application peut choisir ou non (par l'intermédiaire du fichier de configuration de l'application) d'utiliser les assemblages de stratégies d'éditeurs relatifs à l'assemblage à localiser.

### Utilisation de l'élément CodeBase

Si le nom fort n'est pas correctement fourni ou s'il est fourni mais que l'assemblage n'est pas trouvé dans le répertoire GAC alors peut être que l'assemblage doit être chargé à partir d'une *URL* (*Unique Resource Locator*). Cette possibilité est à la base du mécanisme de déploiement *No Touch Deployment* décrit en page 84.

En effet, le fichier de configuration de l'application peut avoir un élément `<codebase>` relatif à l'assemblage à localiser. Dans ce cas, cet élément définit une URL et le CLR tentera de charger l'assemblage à partir de cette URL. Si l'assemblage n'est pas trouvé à cette URL, la localisation échouera. Si l'assemblage à localiser est défini avec un nom fort dans le fichier de configuration, alors l'URL peut être une adresse internet, une adresse intranet ou un répertoire de la machine courante. Si l'assemblage à localiser n'est pas défini avec un nom fort, l'URL ne peut être qu'un sous répertoire du répertoire du répertoire de l'application.

Voici un extrait d'un fichier de configuration d'une application avec l'élément `codebase>` :

```
...
     <dependentAssembly>
        <assemblyIdentity name="Foo3" publicKeyToken="C64B742BD612D74A"
         culture= "fr-FR"/>
        <codebase version="3.0.0.0"
         href = "http://www.smacchia.com/Foo3.dll>"/>
     </dependentAssembly>
...
```

Vous remarquez que l'URL contient le nom du module de l'assemblage contenant le manifeste (en l'occurrence `Foo3.dll`). Si l'assemblage a d'autres modules, comprenez bien que tous ces modules doivent être aussi téléchargeables à cette adresse (en l'occurrence `http://www.smacchia.com/`).

Lorsqu'un assemblage est téléchargé à partir du web grâce à l'élément `<codebase>`, il est stocké dans le cache de téléchargement. Aussi, avant de tenter de charger un assemblage à partir d'une URL, *fusion* va consulter ce cache pour vérifier s'il n'a pas déjà été téléchargé.

## *Le mécanisme de probing*

Si le nom fort n'est pas correctement fourni ou s'il est fourni mais que l'assemblage n'est pas trouvé dans le répertoire GAC et si le fichier de configuration ne contient pas d'élément `<codebase>` relatif à l'assemblage à localiser, alors l'algorithme tente de trouver l'assemblage en sondant certains répertoires. C'est le mécanisme de *probing* (qui peut se traduire par *sondage* en français) qui est exposé par l'exemple suivant :

- Supposons que l'on veuille localiser l'assemblage `Foo` (notez qu'on ne fournit pas l'extension du fichier).
- Supposons que les sous répertoires indiqués par l'élément `<probing>` du fichier de configuration de l'application pour l'assemblage `Foo` soit `"Path1"` et `"Path2\Bin"`.
- Supposons que le répertoire de base de l'application soit `"C:\AppDir\"`.
- Supposons enfin qu'aucune information de culture n'ait été fournie avec le nom d'assemblage, ou qu'il soit de culture neutre (i.e ce n'est pas un assemblage satellite).

La recherche de l'assemblage se fait dans cet ordre, dans les répertoires suivants :

```
C:\AppDir\Foo.dll
C:\AppDir\Foo\Foo.dll
C:\AppDir\Path1\Foo.dll
C:\AppDir\Path1\Foo\Foo.dll
C:\AppDir\Path2\Bin\Foo.dll
C:\AppDir\Path2\Bin\Foo\Foo.dll
C:\AppDir\Foo.exe
C:\AppDir\Foo\Foo.exe
C:\AppDir\Path1\Foo.exe
C:\AppDir\Path1\Foo\Foo.exe
C:\AppDir\Path2\Bin\Foo.exe
C:\AppDir\Path2\Bin\Foo\Foo.exe
```

Si l'assemblage est un assemblage satellite (i.e il n'a pas une culture neutre, par exemple il a la culture `"fr-FR"`) la recherche se fait dans cet ordre, dans les répertoires suivants :

```
C:\AppDir\fr-FR\Foo.dll
C:\AppDir\fr-FR\Foo\Foo.dll
C:\AppDir\Path1\fr-FR\Foo.dll
C:\AppDir\Path1\fr-FR\Foo\Foo.dll
C:\AppDir\Path2\Bin\fr-FR\Foo.dll
C:\AppDir\Path2\Bin\fr-FR\Foo\Foo.dll
C:\AppDir\fr-FR\Foo.exe
C:\AppDir\fr-FR\Foo\Foo.exe
C:\AppDir\Path1\fr-FR\Foo.exe
C:\AppDir\Path1\fr-FR\Foo\Foo.exe
C:\AppDir\Path2\Bin\fr-FR\Foo.exe
C:\AppDir\Path2\Bin\fr-FR\Foo\Foo.exe
```

### *L'événement AppDomain.AssemblyResolve*

Enfin, si l'assemblage n'a pas été trouvé après toutes ces étapes, le CLR déclenche l'événement `AssemblyResolve` de la classe `AppDomain`. Les méthodes abonnées à cet événement peuvent retournée un objet de type `Assembly`. Cela vous permet de fournir votre propre mécanisme de localisation d'assemblage. Cette possibilité est notamment exploitée par la technologie de déploiement *ClickOnce* pour télécharger des groupes de fichiers à la demande comme illustré en page 80.

## *Schéma récapitulatif de l'algorithme de localisation*

Vous pouvez vous servir de l'outil `fuslogvw.exe` pour analyser les *logs* produits par *fusion*. Cet outil est très pratique pour déterminer les causes d'un échec du chargement d'un assemblage.

## *L'élément <assemblyBinding> du fichier de configuration*

Le fichier de configuration d'un assemblage `ASM` peut contenir des paramètres utilisés par *fusion* lorsque le code de `ASM` déclenche la localisation et le chargement d'un autre assemblage. Le fichier d'installation peut contenir un élément `<probing>` qui spécifie un ou plusieurs sous répertoires que le mécanisme de probing doit sonder. Il peut aussi contenir un élément `<dependentAssembly>` pour chaque assemblage à localiser potentiellement. Chacun de ces éléments `<dependentAssembly>` peut contenir les informations suivantes sur la façon dont cet assemblage doit être localisé :

- L'élément `<publisherPolicy>` détermine si les éventuelles stratégies d'éditeur relatives à l'assemblage à charger doivent être prises en compte lors de sa localisation.
- L'élément `<codebase>`, décrit dans la section précédente, définit une URL à partir de laquelle l'assemblage à localiser doit être chargé.
- L'élément `<bindingRedirect>` permet de rediriger le numéro de version comme le ferait une stratégie d'éditeur. La stratégie d'éditeur s'applique à toutes les applications clientes d'un assemblage partagé, alors qu'ici seule l'application paramétrée par ce fichier tient compte de cet élément.

Voici à quoi peut ressembler un fichier de configuration (notez la ressemblance avec le fichier de configuration d'une stratégie d'éditeur) :

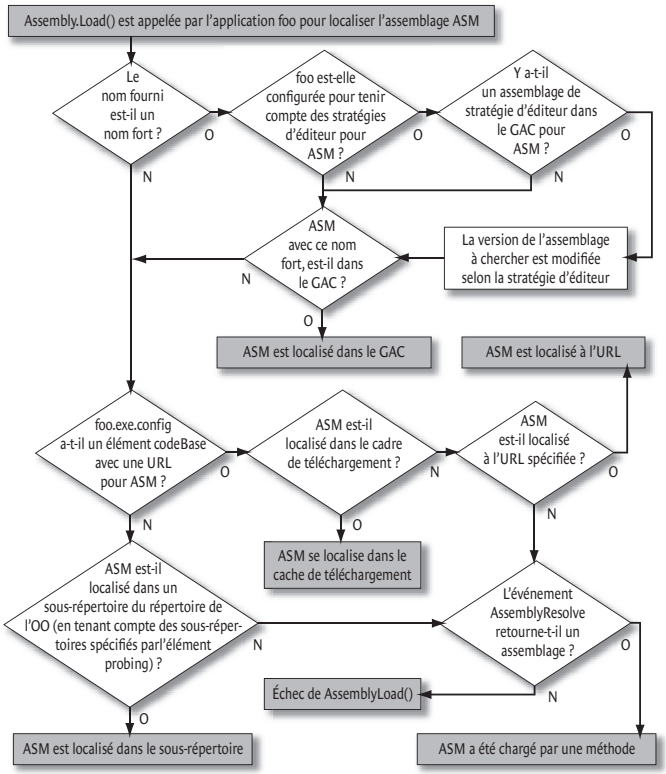


*Figure 4-2 : L'algorithme du CLR de localisation d'un assemblage*

*Exemple 4-10 :*

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
         <probing privatePath="Path1;Path2/bin" />
         <dependentAssembly>
            <assemblyIdentity name="Foo1" culture= "neutral"
                              publicKeyToken="C64B742BD612D74A" />
            <publisherPolicy apply="no" />
            <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0"/>
         </dependentAssembly>
         <dependentAssembly>
            <assemblyIdentity name="Foo2" culture= "neutral"
                              publicKeyToken="C64B742BD612D74A" />
            <codebase version="2.0.0.0"
                      href = "file:///C:/Code/Foo2.dll>"/>
         </dependentAssembly>
```

```
         <dependentAssembly>
            <assemblyIdentity name="Foo3" culture= "fr-FR"
                              publicKeyToken="C64B742BD612D74A" />
            <codebase version="3.0.0.0"
                      href = "http://www.smacchia.com/Foo3.dll>"/>
         </dependentAssembly>
      </assemblyBinding>
   </runtime>
</configuration>
```

Si vous ne souhaitez pas manipuler les documents XML, sachez que ces informations peuvent être configurées à partir de l'outil `.NET Framework Configuration` accessible par : *Menu Démarrer* ▶ *Panneau de configuration* ▶ *Outils d'administration* ▶ *Microsoft .NET Framework 2.0 Configuration* ▶ menu *configured assembly*.

## *Le mécanisme de Shadow Copy*

Lorsqu'un assemblage est chargé dans un processus, le ou les fichiers correspondants sont automatiquement verrouillés par *Windows*. Vous ne pouvez ni les mettre à jour ni les détruire. Vous pouvez seulement les renommer. Dans le cas d'un serveur type ASP.NET, où un même processus héberge potentiellement plusieurs applications, ce comportement pourrait être particulièrement gênant puisqu'il nous obligerait à redémarrer tous le processus à chaque mise à jour d'une seule application.

Heureusement, le chargeur d'assemblages du CLR présente le mécanisme dit de *shadow copy*. Il consiste à copier les fichiers d'un assemblage dans un répertoire cache dédié à l'application, avant de le charger effectivement. Ainsi, les fichiers originaux de l'assemblage peuvent être mis à jour même lorsque celui-ci est en cours d'exécution. ASP.NET exploite cette possibilité et vérifie périodiquement si un assemblage chargé avec cette technique a été mis à jour. Le cas échéant, l'application est redémarrée sans perte de requêtes.

La propriété `stringShadowCopyDirectories{get;set;}` de la classe `AppDomainSetup` vous permet de préciser les répertoires qui contiennent les assemblages qui doivent être «*shadow copiés*».

# *Résolution des types à l'exécution*

## *Notion de chargement implicite et explicite d'un assemblage*

Nous avons souvent besoin d'utiliser dans le code d'un assemblage des types déclarés dans d'autres assemblages. Il existe deux techniques pour exploiter les types définis dans un autre assemblage :

- Soit vous comptez sur votre compilateur pour créer un lien précoce avec le type et vous comptez sur le CLR pour charger l'assemblage qui le contient au bon moment. C'est cette technique nommée *chargement implicite* que nous détaillons ici.
- Soit vous *chargez explicitement* l'assemblage à l'exécution et vous créer un lien tardif avec le type que vous souhaitez exploiter.

Dans les deux cas la responsabilité du CLR incombe à une partie du CLR nommée *chargeur de classe* (*class loader*). Le chargement implicite d'un assemblage `A` est déclenché lors de la première compilation JIT d'une méthode qui utilise un type de `A`.

La notion de chargement implicite se rapproche conceptuellement du mécanisme de chargement des DLLs. De même, la notion de chargement explicite se rapproche conceptuellement du mécanisme d'utilisation d'un objet COM par un langage de script (notamment le mécanisme *Automation* et sa fameuse interface *IDispatch*).

## *Référencer un assemblage à la compilation*

Lorsque l'assemblage `A` s'exécute et lorsque le compilateur JIT compile une méthode de `A` qui a besoin de types définis dans un assemblage `B`, `B` est implicitement chargé par le CLR s'il a été référencé durant la compilation de `A`. Pour que `A` référence `B`, il faut avoir effectué une de ces deux manipulations durant la compilation :

- Soit vous compilez `A` en utilisant le compilateur `csc.exe` en ligne de commande ou dans un script de construction. Il faut alors utiliser les options de compilation `/reference /r` et `/lib`.
- Soit vous utilisez l'environnement *Visual Studio* et il faut utiliser le menu *Reference* ▶ *AddReference*. Rappelons que l'environnement *Visual Studio* .NET utilise d'une manière implicite le compilateur `csc.exe` pour produire des assemblages à partir de code source C♯. Cette manipulation ne fait donc que forcer *Visual Studio* à utiliser les options de compilation `/reference /r` et `/lib` du compilateur `csc.exe`.

Dans les deux cas il faut préciser l'assemblage à charger implicitement par son nom (fort ou non). Les types de l'assemblage `B` ainsi que leurs membres, qui sont référencés dans l'assemblage `A` sont référencés dans les tables *TypeRef* et *MemberRef* de l'assemblage `A`. L'assemblage `B` est référencé dans la table *AssemblyRef* de l'assemblage `A`. Un assemblage peut référencer plusieurs autres assemblages mais il faut absolument éviter les référencements cycliques (`A` référence `B` qui référence `C` qui référence `A`). *Visual Studio* sait détecter et interdit les référencements cycliques. Vous pouvez aussi utiliser l'outil *NDepend* (décrit en page 1034) pour détecter les référencements cycliques d'assemblages.

## *Un exemple*

Voici un exemple illustrant l'infrastructure mise en œuvre pour que le CLR puisse charger implicitement un assemblage. Le premier code définit l'assemblage référencé par l'assemblage défini par le deuxième code.

*Exemple 4-11 : Code de l'assemblage référencé : AssemblageBibliotheque.cs*

```
using System ;
namespace MesTypes{
   public class UneClasse{
      public static int Somme(int a,int b){return a+b;}
   }
}
```

*Exemple 4-12 : Code de l'assemblage référençant : AssemblageExecutable.cs*

```
using System ;
using MesTypes;
class Program{
   static void Main(){
      int i = UneClasse.Somme(3,4) ;
   }
}
```

Notez l'utilisation de l'espace de noms `MesTypes` dans le code de l'assemblage référençant. On aurait pu aussi mettre les classes `UneClasse` et `Program` dans un même espace de noms ou dans l'espace de noms anonyme. Dans ce cas, on aurait illustré le fait qu'un espace de noms peut s'étendre sur plusieurs assemblages.

Il est intéressant d'analyser le manifeste de l'assemblage référençant avec l'utilitaire `ildasm.exe`). On y voit clairement le fait que l'assemblage « `AssemblageBibliotheque` » est référencé. Cette référence est matérialisée par une entrée de la table *AssemblyRef*.

```
...
.assembly extern 'AssemblageBibliotheque'{
  .ver 0:0:0:0
}
...
```

Il est aussi intéressant d'analyser le code IL de la méthode `Main()`. On y voit clairement que la méthode `Somme()` se trouve dans un autre assemblage appelé « `AssemblageBibliotheque` » (physiquement cette information est contenue dans les tables de métadonnées *MemberRef* et *TypeRef*) :

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       9 (0x9)
  .maxstack  2
  .locals init (int32 V_0)
  IL_0000:  ldc.i4.3
  IL_0001:  ldc.i4.4
  IL_0002:  call       int32
     ['AssemblageBibliotheque']MesTypes.UneClasse::Somme(int32,int32)
  IL_0007:  stloc.0

  IL_0008:  ret
} // end of method Program::Main
```

## *Schéma récapitulatif*

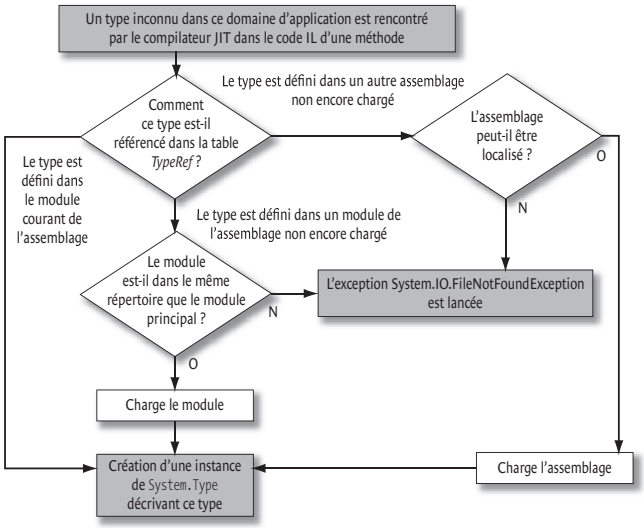


*Figure 4-3 : Résolution d'un type à l'exécution*

# *La compilation « Juste à temps » (JIT Just In Time)*

## *La portabilité au niveau binaire*

Nous rappelons que les applications .NET sont, quel que soit leur langage source, compilées en code IL. Le code IL est stocké dans une section prévue à cet effet, dans les assemblages et les modules. L'application reste codée en IL, jusqu'au moment où elle est exécutée.

Comme son nom l'indique, le langage IL (*Intermediate Langage*) est un langage objet intermédiaire entre les langages .NET de haut niveau (C#, VB.NET...) et le langage machine. L'idée est que la compilation du code IL vers le langage machine natif, se fasse durant l'exécution de l'application, quand on connaît les types de processeurs de la machine. Cela permet aux applications .NET distribuées sous forme d'assemblages contenant du code IL, d'être exécutables sur toutes machines et tous systèmes d'exploitation supportant la plateforme .NET. Le point important à souligner est que pour distribuer une application .NET sur différents systèmes d'exploitation, il n'y a pas besoin de compiler plus d'une fois le code source de haut niveau (i.e le code source C#, VB.NET...). On dit que les applications .NET sont *portables au niveau binaire*.

## *Comprendre le mécanisme de compilation « Just In Time »*

La compilation du code IL en langage machine s'effectue durant l'exécution de l'application. On pourrait imaginer qu'un des deux scénarios suivants soit appliqué :

- L'application est entièrement compilée en langage machine dès son lancement.

- Les instructions IL sont interprétées les unes après les autres. On parle de langage interprété.

Aucun de ces scénarios n'est utilisé pour la compilation du code IL en langage machine. Une solution intermédiaire et plus performante a été mise en place. Cette solution consiste à compiler le corps d'une méthode en langage IL en langage machine, juste avant le premier appel de la méthode. C'est pour cela que le mécanisme s'appelle « *Juste à temps* » (*JIT Just In Time*). La compilation se fait juste à temps pour que l'exécution de la méthode en langage machine puisse se faire.

### *Intercepter le premier appel grâce au stub*

Pour réaliser la compilation JIT le CLR utilise une astuce bien connue dans les mécanismes d'architecture distribuée type *RPC*, *DCOM .NET Remoting* ou *Corba*. À chaque fois qu'une classe est chargée par le CLR, un *stub* (ou *souche* en français) est associé à chacune des méthodes de la classe. En architecture distribuée un stub est une petite couche logicielle côté client qui intercepte les appels aux méthodes pour les convertir en appels distants (en *DCOM* le terme proxy désigne cette notion de stub). Dans le cas du JIT le stub d'une méthode est matérialisé par du code qui intercepte le premier appel à celle-ci.

Ce code contient un branchement vers une fonction du compilateur JIT. Cette fonction vérifie le code IL puis le compile en langage natif. Le code natif résultant est stocké dans l'espace mémoire du processus, puis exécuté. Bien entendu le stub est alors remplacé par un saut vers l'adresse du code natif de la méthode. Ainsi chaque méthode n'est compilée qu'une fois.

### *Vérification du code IL*

Avant de compiler une méthode en langage natif, le compilateur JIT effectue une série de vérification quant à la validité du corps de la méthode. Pour décider qu'une méthode est valide ou pas, le compilateur JIT vérifie l'enchaînement des instructions IL, évalue l'évolution du contenu de la pile et détecte les accès mémoires interdits. Une exception est envoyée si cette vérification échoue.

Votre code C# écrit en mode vérifiable est automatiquement traduit en code IL vérifiable. Cependant, en page 501 nous montrons que sous certaines conditions le compilateur C# peut produire des instructions IL spéciales qui ont la particularité d'être non vérifiables par le compilateur JIT.

### *Optimisations réalisées par le compilateur JIT*

Le compilateur JIT réalise des optimisations. En voici quelques exemples pour fixer les idées :

- Pour éviter le coût du passage des arguments entrants et sortants, le compilateur JIT a la possibilité d'insérer le corps d'une méthode appelée dans le corps de la méthode appelante. Cette optimisation est nommée *inlining*. Pour que le coût de cette optimisation ne soit pas supérieur au gain de performance, la méthode appelée doit satisfaire à un certains nombre de contraintes simples à vérifier. Son corps compilé en IL doit avoir une taille inférieure à 32 octets, elle ne doit pas rattraper d'exceptions, elle ne doit pas contenir de boucles, elle ne doit pas être virtuelle etc.
- Le compilateur JIT peut positionner à `null` une variable locale de type référence après sa dernière utilisation et avant la fin de la méthode. Ainsi, l'objet référencé aura une référence de moins vers lui. Cela augmente les chances qu'il soit collecté plus tôt par le ramasse-miettes. Cette optimisation peut être localement désactivée en utilisant la méthode `System.GC.KeepAlive()`.

- Le compilateur JIT a la possibilité de stocker les variables locales les plus fréquemment utilisées directement dans les registres du processeur plutôt que sur la pile. Cela constitue bien une optimisation car l'accès aux registres du processeur est significativement plus rapide. Cette optimisation est nommée *Enregistration*.

### *Notion de pitching*

La compilation des méthodes par le JIT consomme de la mémoire puisque le code natif des méthodes est stocké. Si le CLR détecte que la mémoire devient une ressource critique pour l'exécution de l'application, il a la possibilité de récupérer de la mémoire en libérant le code natif de certaines méthodes. Naturellement un stub est regénéré pour chacune de ces méthodes. Cette fonctionnalité est appelée *pitching*.

La mise en œuvre du pitching est beaucoup plus compliquée qu'il n'y paraît. Pour être efficace le code des méthodes libérées doit être contigu pour éviter la fragmentation de la mémoire. D'autres problèmes importants apparaissent dans les piles des threads puisqu'elles contiennent des adresses mémoire référençant le code natif de certaines méthodes. Heureusement toutes ces considérations sont totalement masquées au développeur.

Le code natif d'une méthode produit par le compilateur JIT ne survit pas au déchargement du domaine d'application qui le contient.

### *Acronymes JIT et JITA*

Pour ceux qui viennent du monde COM/DCOM/COM+ l'acronyme JIT peut leur rappeler l'acronyme *JITA* (*Just In Time Activation*, décrit page 296). Les deux mécanismes sont différents, ils ont des buts différents et sont utilisés dans des technologies différentes. Cependant l'idée sous-jacente est la même : on ne mobilise des ressources pour une entité (compilation d'une méthode IL pour JIT, activation d'un objet COM pour JITA) que lorsque l'entité est effectivement utilisée (juste avant que l'entité soit utilisée). On qualifie souvent les mécanismes qui utilisent cette idée avec l'adjectif « *paresseux* » (*lazy* en anglais).

## *Compilation avant exécution : l'outil ngen.exe*

*Microsoft* fournit l'outil `ngen.exe` capable de compiler les assemblages avant leur exécution. Cet outil fonctionne donc comme un compilateur classique et remplace le mécanisme JIT. Le vrai nom de `ngen.exe` est « *Native Image Generator* ». L'outil `ngen.exe` est à utiliser si vous constatez que le mécanisme JIT introduit une baisse de performance notable (par exemple en utilisant les compteurs de performance de la compilation JIT décrits dans la section suivante). Cependant le compilateur normal réalise un grand nombre d'optimisations inaccessibles à `ngen.exe`. L'utilisation du compilateur normal est en général préférable.

La compilation par `ngen.exe` se fait en général à l'installation de l'application. Vous n'avez pas à manipuler les nouveaux fichiers contenant le code en langage machine, appelés *images natives*. Ceux-ci sont automatiquement stockés dans un répertoire spécial de votre machine appelé le *cache des images natives* (*Native Image Cache* en anglais). Ce répertoire est accessible à partir de `ngen.exe` avec les options `/show` et `/delete`. Ce répertoire est aussi visible lorsqu'un utilisateur visualise le *cache global des assemblages* car le cache des images natives est inclus dans ce répertoire. Cette visualisation est décrite page 65).

| Option de ngen.exe | Description |
|---|---|
| /show[nom assemblage \|nom répertoire] | Permet de visualiser l'ensemble des images natives contenues dans le « Native Image Cache ».Si vous faite suivre cette option du nom d'un assemblage, vous ne verrez que les images natives ayant le même nom.Si vous faite suivre cette option du nom d'un répertoire, vous ne verrez que les images natives des assemblages contenues dans ce répertoire |
| /delete[nom assemblage \|nom répertoire] | Supprime l'ensemble des images natives contenues dans le « Native Image Cache ».Si vous faite suivre cette option du nom d'un assemblage, vous ne supprimerez que les images natives ayant le même non.Si vous faite suivre cette option du nom d'un répertoire, vous ne supprimerez que les images natives des assemblages contenues dans ce répertoire |
| /debug | Produit une image native utilisable par un débogueur. |

De nombreuses autres options existent. Elles sont décrites dans les **MSDN** à l'article **Native Image Generator (Ngen.exe)**. Notamment, des nouvelles possibilités ont été ajoutées pour supporter les assemblages exploitant la réflexion et pour automatiser la mise à jour de la version compilée d'un assemblage lorsqu'une de ses dépendances évolue. Plus d'information à ce sujet sont disponibles en ligne dans l'article **NGen Revs Up Your Performance with Powerful New Features** de *Reid Wilkes* du numéro d'*Avril 2005* de **MSDN Magazine**.

## *Compteurs de performance de la compilation JIT*

Vous avez accès aux six compteurs suivants de performance du JIT dans la catégorie ".NET CLR Jit".

| Nom du compteur, à fournir sous forme d'une chaîne de caractères. | Description |
|---|---|
| "# of IL Methods JITted" | Compte le nombre de méthodes compilées. Ce compteur n'inclut pas les méthodes déjà compilées par ngen.exe. |
| "# of IL Bytes JITted" | Compte le nombre d'octets de code IL compilé. Les méthodes « pitchées » ne sont pas soustraites de ce total. |
| "Total # of IL Bytes Jitted" | Compte le nombre d'octet de code IL compilé. Les méthodes « pitchées » sont contenues dans ce total. |
| "% Time in Jit" | Pourcentage du temps passé dans le compilateur JIT. Ce compteur est mis à jour à chaque fin de compilation JIT. |

| | |
|---|---|
| `"IL Bytes Jitted / sec"` | Nombre moyen d'octets de code intermédiaire compilé par seconde. |
| `"Standard Jit Failures"` | Nombre de fois où une méthode considérée comme invalide n'a pu être compilée par le JIT. |

Vous avez le choix de visualiser ces compteurs soit pour toutes les applications exécutées en mode géré jusqu'ici depuis le *boot* de la machine, soit pour le processus courant. Le choix de cette visualisation se fait avec l'argument de la méthode `PerformanceCounterCategory.GetCounters()`. Dans le premier cas il faut fournir la chaîne de caractères `"_Global_"` en argument à cette méthode. Dans le second cas il faut fournir la chaîne de caractères égale au nom du processus en argument à cette méthode.

Ces compteurs sont principalement utilisés pour évaluer le coût de la compilation JIT. Si ce coût vous paraît trop élevé, il faut prévoir l'utilisation de l'outil `ngen.exe` lors du déploiement de l'application. Voici un exemple d'utilisation de ces compteurs (notez que le nom de l'assemblage est `MonAssemblage.exe`) :

*Exemple 4-13 :* *`MonAssemblage.cs`*

```
using System.Diagnostics ;
class Program {
   static void DisplayJITCounters() {
      PerformanceCounterCategory perfCategory
                = new PerformanceCounterCategory(".NET CLR Jit") ;
      PerformanceCounter[] perfCounters ;
      perfCounters = perfCategory.GetCounters("MonAssemblage") ;
      foreach(PerformanceCounter perfCounter in perfCounters)
         System.Console.WriteLine("{0}:{1}",
            perfCounter.CounterName,
            perfCounter.NextValue()) ;
   }
   static void f() {
      System.Console.WriteLine("--> Appel à  f().") ;
   }
   static void Main() {
      DisplayJITCounters() ;
      f() ;
      DisplayJITCounters() ;
   }
}
```

Ce programme affiche :

```
# of Methods Jitted:2
# of IL Bytes Jitted:108
Total # of IL Bytes Jitted:108
IL Bytes Jitted / sec:0
Standard Jit Failures:0
% Time in Jit:0
```

```
Not Displayed:0
--> Appel à f().
# of Methods Jitted:3
# of IL Bytes Jitted:120
Total # of IL Bytes Jitted:120
IL Bytes Jitted / sec:0
Standard Jit Failures:0
% Time in Jit:0,02865955
Not Displayed:0
```

Précisons que les compteurs de performances sont aussi visualisables avec l'outil `perfmon.exe` (accessible avec *Menu démarrer* ▶ *Exécuter...* ▶ *perfmon.exe*).

# *Gestion du tas par le ramasse-miettes*

## *Introduction au ramasse-miettes*

Dans les langages .NET, la destruction des objets gérés n'engage aucunement la responsabilité du programmeur. Le problème de la destruction d'un objet de type valeur est très simple à résoudre. L'objet étant physiquement alloué sur la pile du thread courant, lorsque la pile se vide à cet endroit, l'objet est détruit. Le problème de la destruction d'un objet de type référence est beaucoup plus ardu. Pour simplifier la tâche du développeur, la plateforme .NET fournit un *ramasse-miettes* (*garbage collector* ou *GC* en anglais) qui prend en charge automatiquement la récupération de la mémoire des objets désalloués. On parle de *tas géré*.

Lorsqu'il n'y a plus de références vers un objet, ce dernier devient inaccessible par le programme. Le programme n'en a donc plus besoin. Le ramasse-miettes fonctionne sur ce principe : il marque les objets accessibles. Les objets non marqués sont donc inaccessibles par le programme, et doivent être détruits. Un problème est le fait que le ramasse-miettes désalloue un objet inaccessible seulement quand il le décide (en général lorsque le programme a besoin de mémoire). Le développeur a une marge de manœuvre restreinte sur la conduite du ramasse-miettes. Par rapport à d'autres langages, comme le langage C++, cela introduit un certain non déterminisme dans l'exécution des programmes. Une conséquence est que les développeurs se sentent frustrés de ne pas avoir un contrôle total. En revanche, avec ce système, les problèmes récurrents de fuite de mémoire sont beaucoup moins nombreux.

> Les fuites de mémoire subsistent malgré la présence d'un ramasse-miettes. En effet, un développeur peut, par inadvertance, concevoir un programme qui garde des références vers des objets dont il n'a plus besoin (par exemple si une collection « gonfle » indéfiniment). Cependant, la cause la plus courante des fuites de mémoire d'une application .NET est la non désallocation de ressources non gérées.

Pour fixer les idées, le ramasse-miettes est une couche logicielle comprise dans le CLR, présente en un seul exemplaire dans chaque processus.

## *Problématique des algorithmes possibles pour un ramasse-miettes*

Le but n'est pas de faire un exposé des différents algorithmes de ramasse-miettes existants, mais de sensibiliser le lecteur aux problèmes qui se posent aux concepteurs du ramasse-miettes. Ainsi

il comprendra mieux pourquoi l'algorithme décrit dans la section suivante a été choisi par *Microsoft* pour son implémentation du ramasse-miettes.

On pourrait penser qu'il suffit de libérer un objet lorsque celui-ci n'est plus référencé. Cette approche simpliste peut facilement être mise en défaut. Imaginez deux objets `A` et `B` ; `A` maintient une référence vers `B` et `B` maintient une référence vers `A` ; ces deux objets n'admettent pas d'autres références que leurs références mutuelles. Clairement le programme n'a plus besoin de ces objets et le ramasse-miettes doit les détruire, alors qu'il existe encore une référence valide sur chacun d'eux. Le ramasse-miettes utilise donc des algorithmes du type arbres de références et détection de boucles dans un graphe. L'arbre de références admet pour racines certains objets considérés comme immuables.

Un autre problème que celui de la destruction des objets incombe au ramasse-miettes. C'est la *fragmentation du tas*. Après un certain temps, à force d'avoir désalloué et alloué des zones mémoires de tailles très variables, le tas est fragmenté. Il est parsemé d'espaces mémoire non utilisés par le programme. Ce fait induit un énorme gaspillage de mémoire. Il incombe au ramasse-miettes de défragmenter le tas, afin de limiter les conséquences de ce problème. Là encore plusieurs algorithmes et approches existent.

Enfin, le bon sens et l'approche empirique nous enseigne la règle suivante : plus un objet est ancien plus son espérance de vie est longue, plus il est récent plus son espérance de vie est courte. Si cette règle est prise en compte par un algorithme, c'est-à-dire si l'on essaye plus souvent de désallouer les objets récents que les objets anciens, alors le ramasse-miettes sous-jacent gagnera nécessairement en performance.

L'algorithme du ramasse-miettes .NET prend en compte ces problématiques et cette règle temporelle.

## *Algorithme du ramasse-miettes .NET*

Une *collecte* des objets par le ramasse-miettes est déclenchée automatiquement par le CLR lorsqu'une pénurie potentielle de mémoire est pressentie. L'algorithme de cette prise de décision n'est pas documenté par *Microsoft*.

Le terme de *génération* définit le laps de temps entre deux collectes. Chaque objet appartient donc à la génération qui l'a vue naître. De plus, à chaque instant l'équation suivante est vérifiée :

{Nombre de génération qu'il y a eu dans ce processus} = 1+{Nombre de collectes qu'il y a eu dans ce processus}.

Les générations sont numérotées. Le numéro d'une génération est augmenté à chaque collecte jusqu'à ce qu'il atteigne un numéro de génération maximal (égal à 2 dans l'implémentation actuelle du CLR). Par convention la génération 0 est la génération la plus jeune. Lorsqu'un objet doit être alloué sur le tas, il fait partie de la génération 0.

Une conséquence de ce système de générations est que les objets d'une même génération sont contigus en mémoire comme l'indique la Figure 4-4 :

## *Étape 1 : définir les racines de l'arbre des références vers les objets actifs*

Les racines de l'arbre de références vers les *objets actifs* (i.e les objets à ne pas désallouer) sont principalement, les objets référencés par les champs statiques des classes, les objets référencés

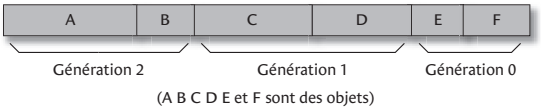


*Figure 4-4 : Générations d'objets*

dans les piles des threads du processus et les objets dont l'adresse physique (ou un *offset* basé sur celle-ci) est contenue dans un registre de l'unité centrale.

## *Étape 2 : fabriquer l'arbre et marquer les objets qui sont encore référencés*

Le ramasse-miettes construit l'arbre à partir des racines, en ajoutant les objets référencés par les objets déjà dans l'arbre, et en itérant cette opération. Chaque objet référencé par l'arbre est marqué comme actif. Lorsque l'algorithme rencontre un objet déjà marqué comme actif, il ne le prend pas en compte afin d'éviter les problèmes de cycle de référencement d'objets. Cette étape se termine lorsque le ramasse-miettes ne peut plus collecter un objet référencé, qui ne soit déjà marqué comme actif. Le ramasse-miettes se sert des métadonnées de type pour trouver les références contenues dans un objet à partir de la classe de l'objet.

Sur la Figure 4-5, on voit que les objets `A` et `B` sont des objets qui constituent les racines de l'arbre. `A` et `B` référencent `C`. `B` référence `E` et `C` référence `F`. L'objet `D` n'est donc pas marqué comme actif.

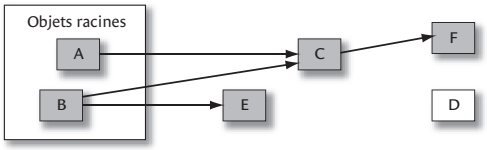


*Figure 4-5 : L'arbre des références du ramasse-miettes*

## *Étape 3 : désallouer les objets inactifs*

Le ramasse-miettes parcourt linéairement le tas et désalloue les objets non marqués comme actifs. Certains objets ont besoin que leur méthode `Finalize()` soit appelée pour pouvoir être physiquement détruit. Cette méthode nommée *finaliseur*, est définie par la classe `Object`. Elle doit être appelée avant la destruction d'un objet dont la classe redéfinit cette méthode. Les invocations des finaliseurs sont prises en charge par un thread dédié de façon à ne pas surcharger le thread qui effectue la collecte des objets. Une conséquence est que les emplacements mémoires des objets qui ont un finaliseur survivent à une collecte.

Le tas n'est pas nécessairement entièrement parcouru, soit parce que le ramasse-miettes a collecté suffisamment de mémoire, soit parce que nous avons à faire à une *collecte partielle*. L'algorithme de parcours de l'arbre des références est profondément impacté par cette possibilité de collecte partielle car il se peut que des objets anciens (de la génération 2) référencent des objets jeunes (de la générations 0 ou 1). Bien que la fréquence de déclenchement des collectes dépend complètement de votre application vous pouvez vous baser sur ces ordres de grandeur : une

collecte partielle de la génération 0 par seconde, une collecte partielle de la génération 1 pour 10 collectes partielles de la génération 0, une collecte complète pour 10 collectes partielles de la génération 1.

La Figure 4-6 montre que l'objet `D` n'est pas marqué. Il est donc inactif et va être détruit.

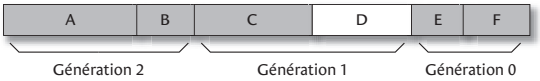


*Figure 4-6 : Destruction des objets inactifs*

## *Étape 4 : défragmentation du tas*

Le ramasse-miettes défragmente le tas, c'est-à-dire que les objets actifs sont déplacés vers le bas du tas pour « combler » les espaces mémoires vides qui contenaient les objets désalloués à l'étape précédente. L'adresse du haut du tas est recalculée et chaque génération est incrémentée. Les objets anciens sont vers le bas du tas et les objets récents vers le haut. De plus les objets actifs créés au même moment sont aussi physiquement proches. Seuls les objets jeunes sont souvent examinés. Ainsi, ce sont toujours les mêmes pages mémoire qui sont souvent sollicitées. Ce fait est très important pour obtenir de bonnes performances.

Sur la Figure 4-7 (par rapport à la Figure 4-6) le ramasse-miettes a incrémenté les numéros de génération. On suppose que la classe de l'objet `D` n'avait pas de finaliseur. Ainsi, la mémoire de l'objet `D` a été désalloué et le tas a été défragmenté (`E` et `F` ont été bougés pour combler l'espace mémoire anciennement alloué à `D`). Notez que `A` et `B` n'ont pas vu leur numéro de génération augmenter puisqu'il était déjà dans la génération 2.

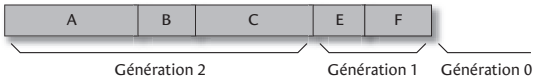


*Figure 4-7 : Défragmentation du tas*

Certains objets sont « inamovibles » (on dit aussi « épinglés ») c'est-à-dire qu'ils ne peuvent être bougés physiquement par le ramasse-miettes. Cette particularité s'appelle « *to pin an object* » en anglais (*pin* veut dire punaise/épingle en français). Un objet est inamovible lorsqu'il est pointé par du code non protégé. Plus de détails à ce sujet sont disponibles en page 505.

## *Étape 5 : recalculer les adresses contenues dans les références*

Les adresses mémoire de certains objets actifs ont pu être modifiées par l'étape précédente. Il faut donc parcourir l'arbre des objets actifs et actualiser les références avec les nouvelles adresses.

## *Bonnes pratiques*

Les bonnes pratiques suivantes découlent naturellement des propriétés de l'algorithme que nous venons d'exposer :

- Libérez vos objets dés que possible pour éviter la promotion d'objets dans les générations.

- Identifiez quels objets sont susceptibles d'avoir une vie longue, analysez les causes et essayez de réduire leurs durées de vie. Pour cela, nous vous conseillons d'avoir recours à l'outil *CLR Profiler* fournit gratuitement par *Microsoft* ou au profiler fourni avec *Visual Studio Team System* .
- Dans la mesure du possible, évitez de référencer un objet avec une vie courte à partir d'un objet avec une vie longue.
- Éviter d'implémenter un finaliseur dans vos classes pour que vos objets ne survivent pas à une collecte.
- Assigné vos références à `null` dés que possible, surtout avant un long appel.

## *Tas spécial pour les gros objets*

Tous les objets dont la taille est inférieure à une certaine taille sont traités dans le tas géré décrit dans les étapes précédentes. Cette taille est non documentée par *Microsoft.* Un ordre de grandeur est de 20 à 100 Ko. Les objets dont la taille est supérieure à ce seuil sont stockés dans un autre tas spécial, pour des raisons de performances. En effet, dans ce tas, les objets ne sont pas physiquement bougés par le ramasse-miettes. Une page mémoire *Windows* a une taille de 4 ou 8Ko en fonction du processeur sous-jacent. Chacun de ces objets est stocké sur un nombre entier de page mémoire même s'il n'a pas une taille exactement multiple de 4 ou 8Ko. Ceci induit un peu de gaspillage mais ce défaut n'affecte pas significativement le gain de performance. Cette différence de stockage est complètement transparente pour le développeur.

## *Ramasse-miettes et applications multithreads*

L'exécution d'une collecte du ramasse-miettes peut se faire avec un des threads applicatifs si elle est déclanchée manuellement ou avec un thread du CLR lorsque celui-ci décide qu'une collecte doit avoir lieu. Avant de commencer une collecte, le CLR doit stopper l'exécution des autres threads de l'application pour éviter qu'ils modifient le tas. Pour cela, différentes techniques existent. On peut citer l'insertion de *point protégés* (*safe point* en anglais) par le compilateur JIT, qui permettent aux threads applicatifs de vérifier si une collecte est en attente de commencement. Notez que la génération 0 du tas est en général découpée en portions (appelées *arènes*), une par thread, pour éviter les problèmes de synchronisation liés aux accès concurrents au tas. Rappelons enfin que les finaliseurs sont exécutés par un thread du CLR dédié à cette tâche.

## *Les références faibles (weak reference)*

### *La problématique*

Lors de l'exécution d'une application, à chaque instant, chaque objet est soit actif, c'est-à-dire que l'application a encore une référence vers lui, soit inactif. Lorsqu'un objet passe du stade actif au stade inactif, c'est-à-dire lorsque l'application vient de détruire la dernière référence vers cet objet, il n'y a plus aucun espoir d'accéder à l'objet.

En fait il existe un troisième stade intermédiaire entre actif et inactif. Lorsque l'objet est dans ce stade, l'application peut l'accéder, mais le ramasse-miettes peut le désallouer. Il y a, à priori, contradiction, car si l'objet est accessible il est référencé et s'il est référencé il ne peut être désalloué. En fait, il faut introduire la nouvelle notion de *référence faible* (*weak reference* en anglais) pour pouvoir comprendre ce stade intermédiaire. Lorsqu'un objet est référencé par une référence faible, il est à la fois accessible par l'application et désallouable par le ramasse-miettes.

## *Pourquoi utiliser des références faibles ?*

Le développeur peut utiliser une référence faible sur un objet si ce dernier satisfait toutes les conditions suivantes :

- L'objet peut être potentiellement utilisé plus tard mais ce n'est pas certain. Si nous sommes certains de l'utiliser plus tard il faut utiliser une référence forte.
- L'objet peut être intégralement reconstruit à l'identique (par exemple à partir d'une base de données). Si on a potentiellement besoin de l'objet plus tard mais qu'on ne peut le reconstruire à l'identique il ne faut pas que le ramasse-miettes détruise l'objet.
- L'objet est relativement volumineux en mémoire (plusieurs Ko). Si l'objet est léger on peut le garder en mémoire. Cependant si les deux conditions précédentes s'appliquent à un grand nombre d'objets légers il est judicieux d'utiliser une référence faible pour chacun de ces objets.

Tout ceci est bien théorique. Dans la pratique on dit qu'on utilise un cache d'objets. En effet, ces conditions sont toutes remplies par les objets contenus dans un cache (on parle du concept de cache en général et pas d'une implémentation particulière).

L'utilisation de caches peut être considérée comme une régulation naturelle et automatique du compromis entre l'espace mémoire, la puissance de calcul et la bande passante du réseau. Lorsque le cache est trop rempli, une partie des objets « cachés » sont détruits. Cependant, dans l'hypothèse, vraisemblable, où l'on doit accéder à un de ces objets plus tard, il faudra utiliser de la puissance de calcul (pour fabriquer l'objet) ou/et de la bande passante réseaux (pour obtenir les données contenues dans l'objet, à partir d'une base de données par exemple).

En résumé, si vous avez à implémenter un cache nous vous conseillons d'utiliser les références faibles.

## *Comment utiliser des références faibles ?*

L'utilisation des références faibles est particulièrement aisée grâce à la classe `System.WeakReference`. Un exemple valant mieux qu'un long discours, examinez le code C# suivant :

*Exemple 4-14 :*

```
class Program {
   public static void Main() {
      // 'obj' est une référence forte vers l'objet créé ci-dessous.
      object obj = new object() ;

      // 'wobj' est une référence faible vers notre objet.
      System.WeakReference wobj = new System.WeakReference(obj);

      obj = null ; // Destruction de la référence forte 'obj'.
      // ...
      // Ici l'objet est potentiellement détruit par le GC.
      // ...
      // Création d'une référence forte à partir de la référence faible.
      obj = wobj.Target;
      if (obj == null) {
```

```
            // Le GC a détruit l'objet, il n'est plus accessible !
        }
        else {
            // L'objet existe toujours , nous pouvons l'utiliser et il
            // ne peut plus être détruit par le GC.
        }
    }
}
```

La classe `WeakReference` présente la méthode `bool IsAlive()` qui retourne `true` si l'objet référencé faiblement n'a pas encore été détruit. En outre, il est classique et recommandé de mettre la référence forte à `null` dès qu'une référence faible est créée pour s'assurer que la référence forte est détruite.

### *Références faibles courtes et références faibles longues*

Il y a deux constructeurs pour la classe `WeakReference` :

```
WeakReference(object target) ;
WeakReference(object target, bool trackResurrection) ;
```

Dans le premier constructeur le paramètre `trackResurrection` est positionné implicitement à `false`. Si ce paramètre est positionné à `true` l'application peut encore accéder à l'objet entre l'instant où la méthode `Finalize()` a été appelée et l'instant où l'emplacement mémoire de l'objet est réellement modifié (en général par la recopie d'un autre objet dans cet emplacement mémoire lors de la défragmentation du tas). Dans ce cas on dit que c'est une *référence faible longue* (*long weak reference* en anglais). Si le paramètre `trackResurrection` est positionné à `false` l'application ne peut plus accéder à l'objet dès que la méthode `Finalize()` a été appelée. Dans ce cas on dit que c'est une *référence faible courte* (*short weak reference* en anglais).

Malgré le gain potentiel (mais minime) des références faibles longues il vaut mieux ne jamais les utiliser, car elles sont difficiles à maintenir. En effet, une évolution du corps de la méthode `Finalize()` qui ne tiendrait pas compte du fait qu'il y a des références faibles longues sur des instances de cette classe pourrait entraîner la résurrection d'objet dans un état incorrect.

## *Agir sur le comportement du ramasse-miettes avec la classe System.GC*

On peut se servir des méthodes statiques de la classe `System.GC` pour agir sur le comportement du ramasse-miettes ou seulement analyser ce comportement. L'idée est bien évidemment d'améliorer les performances de nos applications. Cependant, *Microsoft* ayant investi beaucoup de ressources à optimiser le ramasse-miettes .NET, nous vous conseillons de n'utiliser les services de la classe `System.GC` que lorsque vous êtes certain du gain de performance produit par vos modifications. Voici la propriété et les méthodes statiques de cette classe :

- `static int MaxGeneration{ get ; }`
  Cette propriété renvoie le nombre maximal de génération du tas géré -1. Par défaut, dans l'implémentation *Microsoft* courante de .NET, cette propriété vaut 2 et est garantie constante durant le cycle de vie d'une application.

- `static void WaitForPendingFinalizers()`
  Cette méthode suspend le thread qui l'appelle, jusqu'à ce que l'ensemble des finaliseurs ait été exécuté par le thread dédié à cette tâche.
- `static void Collect()`
  `static void Collect( int generation )`
  Cette méthode déclenche une collecte du ramasse-miettes. Vous avez la possibilité de déclancher une collecte partielle, auquel cas le ramasse-miettes ne s'occupera que des objets dont le numéro de génération est entre `generation` et 0. `generation` ne peut être supérieur à `MaxGeneration`. Lors de l'appel de la surcharge sans paramètre, le ramasse-miettes collecte toutes les générations.
  Cette méthode est en général invoquée à mauvais escient, parce que le développeur espère qu'elle va résoudre les problèmes de mémoire inhérents au design de son application (trop d'objets alloués, trop de références entre objets, fuite de mémoire etc).
  Il est cependant intéressant de déclancher une collecte du ramasse-miettes juste avant l'appel de traitements critiques, qui seraient gênés par une surcharge de la mémoire ou une baisse de performance subite, due au déclenchement ramasse-miettes. Dans ce cas, il est conseillé d'appeler les méthodes dans cet ordre afin d'être certains que l'on commence notre traitement avec le maximum de mémoire possible :

```
// Lance une première collecte.
GC.Collect()
// Attend que tous les finaliseurs aient été exécutés.
GC.WaitForPendingFinalizers()
// Libère la mémoire de chaque objet dont le finaliseur
// vient d'être exécuté.
GC.Collect()
```

- `static int CollectionCount( int generation )`
  Retourne le nombre de collectes déjà effectuées pour la génération indiquée. Cette méthode peut être utile pour détecter si une collecte à eu lieu durant un intervalle de temps.
- `static int GetGeneration( object obj )`
  `static int GetGeneration( WeakReference wo )`
  Retourne le numéro de génération de l'objet référencé soit par une référence forte (premier cas) soit par une référence faible (deuxième cas).
- `static void AddMemoryPressure( long pressure )`
  `static void RemoveMemoryPressure ( long pressure )`
  La problématique sous-jacente à l'utilisation de ces méthodes est une conséquence du fait que le ramasse-miettes ne tient pas compte de la mémoire non gérée dans ses algorithmes. Imaginez que 32 instances de la classe `Bitmap` qui ont chacune une taille de 32 octets maintiennent chacune une référence vers un bitmap non géré de 6Mo. Sans l'utilisation de ces méthodes, le ramasse-miettes se comporterait comme si seulement 32x32 octets étaient alloués, à savoir, il ne jugerait pas nécessaire de déclancher des collectes. Une bonne pratique est d'utiliser ces méthodes dans les constructeurs et les finaliseurs de vos classes dont les instances maintiennent des grosses zones de mémoire non gérées. Nous précisons que la classe `HandleCollector` décrite en page 277 permet de fournir le même genre de service.
- `static long GetTotalMemory( bool forceFullCollection )`

Retourne une estimation de la taille courante en octets du tas géré. Vous pouvez affiner l'estimation en mettant `forceFullCollection` à `true`. Dans ce cas la méthode devient bloquante, si une collecte est en cours d'exécution. La méthode retourne lorsque la collecte est terminée ou avant si l'attente dépasse un certain intervalle de temps. La valeur retournée sera plus précise si la méthode retourne lorsque le ramasse-miettes a fini sa tâche.

- `static void KeepAlive( object obj )`
  Garantit que `obj` ne peut être détruit par le ramasse-miettes durant l'exécution de la méthode appelant `KeepAlive()`. `KeepAlive()` doit être appelée à la fin du corps de la méthode. Vous pouvez penser que cette méthode ne sert à rien puisqu'elle nécessite une référence forte vers l'objet qui garantit par son existence ce comportement de non destruction. En fait, le compilateur JIT optimise le code natif produit en positionnant une variable locale de type référence à nulle après sa dernière utilisation et avant la fin du corps de la méthode. La méthode `KeepAlive()` ne fait que désactiver cette optimisation.
- `static void SuppressFinalize(object obj)`
  La méthode `Finalize()` ne sera pas appelée par le ramasse-miettes sur l'objet passé en paramètre. Rappelez-vous que l'appel à `Finalize()` sur tous les objets avant la fin du processus, est un comportement garanti par le ramasse-miettes.
  La méthode `Finalize()` devrait logiquement, contenir du code pour désallouer des ressources possédées par l'objet. Cependant on ne maîtrise pas le moment de l'appel à `Finalize()`. Ainsi on crée souvent une méthode spécialement dédiée à cette désallocation de ressources que l'on appelle quand on le souhaite. En général on utilise la méthode `IDisposable.Dispose()` à cet effet. C'est dans cette méthode spéciale que l'on doit appeler `SuppressFinalize()` puisque après son invocation il n'y aura plus lieu d'appeler `Finalize()`. Plus de détails à ce sujet sont disponibles en page 423.
- `static void ReRegisterForFinalize(object obj)`
  La méthode `Finalize()` de l'objet passé en paramètre sera appelée par le ramasse-miettes. On utilise en général cette méthode dans deux cas :
  - 1er cas : Lorsque la méthode `SuppressFinalize()` a déjà été appelée et que l'on change d'avis (cette pratique est néanmoins à éviter car elle signale une faiblesse de conception).
  - 2e cas : Si on appelle `ReRegisterForFinalize()` dans le code de `Finalize()` l'objet survivra aux collectes. Cela peut servir à analyser le comportement du ramasse-miettes. Cependant si on répète indéfiniment l'appel à `Finalize()` le programme ne s'arrêtera jamais, donc il faut prévoir une condition avant l'appel de `ReRegisterForFinalize()` dans le code de `Finalize()`. De plus si le code d'une telle méthode `Finalize()` référence d'autres objets il faut être très prudent car ils peuvent être détruits par le ramasse-miettes sans que l'on s'en aperçoive. En effet, cet objet « indestructible » n'est pas constamment considéré comme actif, donc ses références vers d'autres objets ne sont pas forcément prises en compte lors de la construction de l'arbre des références vers les objets actifs.

# *Facilités fournies par le CLR pour rendre votre code plus fiable*

## *Les exceptions asynchrones du CLR et la fiabilité du code géré*

Nous espérons que le contenu du présent chapitre vous a convaincu qu'exécuter du code d'une manière gérée est un progrès déterminant. Il est temps maintenant de s'intéresser à la face obscure des environnements gérés. Dans un tel environnement, des traitements coûteux peuvent être déclenchés implicitement par le CLR à pratiquement n'importe quel moment de l'exécution. Cette particularité implique qu'il est impossible de prévoir quand une pénurie de ressource peut survenir. Voici quelques exemples classiques de tels traitements coûteux déclenchés inopinément par le CLR (cette liste est loin d'être exhaustive) :

- Chargement d'un assemblage.
- Exécution d'un constructeur de classe.
- Collecte d'objets par le ramasse miettes.
- Compilation JIT d'une classe.
- Parcours de la pile CAS.
- Boxing implicite.
- Création des champs statiques d'une classe dont l'assemblage est partagé par les domaines d'application du processus.

Une pénurie de ressource se traduit en général par la levée d'une exception par le CLR de type `OutOfMemoryException`, `StackOverflowException` ou `ThreadAbortException` sur le thread qui exécute la demande de ressource responsable de la pénurie. On parle d'*exception asynchrone*. Cette notion s'oppose à la notion d'*exception applicative*. Lorsqu'une exception applicative est levée, il incombe au code courant de la rattraper et de la traiter. Typiquement lorsque vous souhaitez accéder à un fichier il faut prévoir qu'une exception de type `FileNotFoundException` peut être levée. En revanche, lorsqu'une exception asynchrone est levée par le CLR, le code en cours d'exécution ne peut en être tenu pour responsable. En conséquence, il est déconseillé de céder à la psychose en mettant des blocs `try`/`catch`/`finally` partout dans votre code pour limiter les effets de bord des exceptions asynchrones. L'entité responsable du traitement des exceptions asynchrones est l'hôte du moteur d'exécution que nous avons déjà eu l'occasion de présenter dans ce chapitre.

Dans le cas d'une application console ou fenêtrée classique, la levée d'une exception asynchrone est un événement rare qui traduit en général un problème d'algorithme (fuite de mémoire, appels récursifs abusifs ou infinis etc). En conséquence l'hôte du moteur d'exécution de ces applications fait en sorte de terminer le processus tout entier lorsqu'une exception asynchrone est non rattrapée par le code applicatif.

Il en est de même dans le cas des applications ASP.NET. En effet, on constate que les différents mécanismes de détection d'un comportement anormal recyclent le processus en général avant même que des exceptions asynchrones soient lancées. Ces mécanismes sont exposés en page 890.

Ainsi, jusqu'à l'intégration du CLR dans le produit *SQL Server 2005*, les exceptions asynchrones n'étaient pas si problématiques. Pour ne pas régresser, la version *2005* de *SQL Server* doit fournir un taux de fiabilité de cinq neufs. En d'autres termes le service de persistance des données doit

être disponible 99.999% du temps soit, au pire, 5 minutes et 15 secondes d'indisponibilité par an. En outre, pour être efficace, le processus de *SQL Server 2005* doit charger un maximum de données en mémoire et limiter les chargements de pages mémoire à partir du disque dur. Il est donc amené à flirter régulièrement avec la limite de 2 ou 3GB du processus si la mémoire vive disponible le permet (ce qui est maintenant le cas sur la plupart des serveurs). Enfin, les mécanismes de *time out* sur les exécutions des requêtes fonctionnent à base de levée de l'exception `ThreadAbortException`. En résumé, lorsque vous avez à faire à ce type de serveur qui pousse le système dans ses retranchements non seulement les exceptions asynchrones deviennent des événements banals mais en plus il faut éviter absolument qu'elles ne provoquent le crash du processus.

Face à de tels impératifs les concepteurs du CLR ont du imaginer de nouvelles techniques. Elles constituent le sujet de la présente section.

Garder bien à l'esprit que ces techniques doivent être utilisées avec une grande sagesse, seulement lorsque vous développez un serveur de grande envergure qui nécessite son propre moteur d'exécution et qui est susceptible de faire face à des exceptions asynchrones.

## *Les régions d'exécution contraintes (CER)*

Pour éviter de faire tomber le processus tout entier, il faut pouvoir décharger un domaine d'application lorsqu'une exception asynchrone y survient. On parle de *recyclage de domaines d'application*. Toute la problématique d'un tel recyclage est de le réaliser proprement, sans fuite de mémoire et sans corrompre l'état général du processus. Il faut donc un mécanisme permettant de se protéger ponctuellement des exceptions asynchrones. Sans un tel mécanisme, il est impossible de garantir que les ressources non gérées détenues au moment où une exception asynchrone survient soient désallouées correctement.

Le *framework* .NET 2.0 vous permet d'indiquer au CLR les portions de code où la levée inopinée d'une exception asynchrone serait catastrophique. Si une exception asynchrone doit survenir, l'idée est de forcer le CLR à la lever soit avant l'exécution de la portion de code soit après mais pas pendant. On nomme de telles portions de code des *régions d'exécution contraintes* (*Constrained Execution Region*s en anglais ou plus simplement *CER*).

Pour éviter de lever une exception asynchrone durant l'exécution d'une CER, le CLR doit effectuer un certains nombre de préparation juste avant de commencer à l'exécuter. L'idée est de tenter de déclencher la pénurie de ressource si celle-ci doit avoir lieu avant même que la CER soit exécutée. Typiquement, le CLR compile en code natif toutes les méthodes susceptibles d'être exécutées. Pour connaître ces méthodes, il parcourt statiquement le graphe d'appel ayant pour racine la CER. En outre, le CLR sait retenir une exception de type `ThreadAbortException` qui survient pendant l'exécution d'une CER jusqu'à la fin de l'exécution.

Le développeur doit veiller à ne pas allouer de mémoire au sein d'une CER. Cette contrainte est particulièrement forte puisque de multiples conditions implicites peuvent déclencher une allocation mémoire. On peut citer les instructions de *boxing*, les accès à un tableau multidimensionnel ou les manipulations des objets de synchronisation.

## *Définir vos propres CER*

Une CER se définit par l'appel à la méthode statique `void PrepareConstrainedRegion()` de la classe `System.Runtime.CompilerServices.RuntimeHelpers`juste avant la déclaration d'un bloc

try/catch/finally. Tout le code atteignable à partir des blocs catch et finally représente alors une CER.

La méthode statique ProbeForSufficientStack() de cette classe est éventuellement appelée lors de l'appel à PrepareConstrainedRegion(). Cela dépend du fait que l'implémentation des CER par votre hôte du moteur d'exécution doit gérer les cas de dépassement de la taille maximale de la pile d'appels du thread courant (*stack overflow*). Sur un processeur x86, cette méthode tentera de réserver 48Ko.

Malgré cette quantité de mémoire réservée, il se peut qu'une situation de *stack overflow* survienne. Aussi vous pouvez vous faire suivre votre appel à PrepareConstrainedRegion() par un appel à la méthode statique ExecuteCodeWithGuaranteedCleanup() afin d'indiquer une méthode contenant du code de nettoyage à invoquer le cas échéant. Une telle méthode doit être marquée avec l'attribut PrePrepareMethodAttribute pour préciser à l'outil ngen.exe sa fonction particulière.

La classe RuntimeHelpers présente des méthodes permettant aux développeurs d'assister le CLR dans la préparation du terrain avant l'exécution de la CER. Vous pouvez par exemple les appeler dans des constructeurs de classes.

- `static void PrepareMethod(RuntimeMethodHandle m)`
  Le CLR parcours statiquement le graphe des méthodes appelées dans une CER afin de les compiler en code natif. Un tel parcours statique ne peut détecter qu'elle version d'une méthode virtuelle est appelée. Aussi, il incombe au développeur de forcer la compilation de la version appelée avant l'exécution de la CER.
- `static void PrepareDelegate(Delegate d)`
  Les délégués qui vont être invoqués durant l'exécution d'une CER doivent être préparés au préalable en ayant recours à cette méthode.
- `static void RunClassConstructor(type t)`
  Vous pouvez forcer l'exécution d'un constructeur de classe avec cette méthode. Naturellement, l'exécution n'a lieu que si le constructeur de classe concerné n'a pas déjà été invoqué.

## *Les portails de mémoire (memory gates)*

Dans le même esprit que la méthode statique ProbeForSufficientStack() vous pouvez avoir recours à la classe System.Runtime.MemoryFailPoint qui s'utilise comme suit :

```
// On est sur le point d'effectuer une opération qui a besoin
// d'au plus 15 Mb de mémoire.
using( new MemoryFailPoint(15) ) {
    // Effectue l'opération ...
}
```

Contrairement à la méthode ProbeForSufficientStack() la quantité de mémoire indiquée n'est pas réservée. Le constructeur de cette clase évalue seulement si lors de son exécution cette quantité de mémoire pourrait être demandée au système d'exploitation par le CLR sans qu'une exception de type OutOfMemoryException soit levée. Notez qu'entre le moment où s'effectue cette demande et le déroulement effectif de l'opération, les conditions peuvent avoir évoluées, notamment parce qu'un autre thread a demandé beaucoup de mémoire. Malgré cette faiblesse l'utilisation de cette technique est généralement efficace.

S'il s'avère que la quantité de mémoire indiquée est indisponible, le constructeur de la classe `MemoryFailPoint` lève une exception de type `System.InsufficientMemoryException`. Pour cette raison, on parle parfois de *portail de mémoire* (*memory gates* en anglais) pour désigner cette fonctionnalité.

## *Les contrats de fiabilité*

Le *framework* .NET 2.0 présente l'attribut `System.Runtime.ConstrainedExecution.ReliabilityContractAttribute` qui ne s'applique qu'aux méthodes. Cet attribut permet de documenter le niveau de gravité maximal auquel on peut s'attendre si une exception asynchrone survient lors de l'exécution d'une méthode marquée. Ces niveaux de gravité sont définis par les valeurs de l'énumération `System.Runtime.ConstrainedExecution.Consistency` :

| Consistency | Description |
|---|---|
| MayCorruptProcess | La méthode marquée peut au pire corrompre l'état du processus tout entier et ainsi, provoquer un crash. |
| MayCorruptAppDomain | La méthode marquée peut au pire corrompre l'état du domaine d'application courant et ainsi, provoquer son déchargement. |
| MayCorruptInstance | La méthode marquée peut au pire corrompre l'état de l'instance sur laquelle elle est appelée et ainsi, provoquer sa destruction. |
| WillNotCorruptState | La méthode marquée ne peut corrompre aucun état. |

Une deuxième valeur de type `System.Runtime.ConstrainedExecution.Consistency` s'applique à chaque attribut `ReliabilityContractAttribute`. Elle permet de documenter si la méthode peut mettre en défaut les garanties d'une éventuelle CER qui l'appelle. Bien entendu, si la méthode peut corrompre un des états processus ou domaine d'application, elle peut mettre en défaut les garanties et ainsi il faut prévoir la valeur `Cer.None`.

Les contrats de fiabilité constituent un moyen de documenter votre code. Sachez cependant qu'ils sont aussi exploités par le CLR lorsque celui-ci parcours le graphe d'appels statique lors de la préparation de l'exécution d'une CER. Si une méthode sans contrat de fiabilité suffisant est rencontrée, ce parcours s'arrête (puisque de toutes façons le code est non fiable) mais la CER sera quand même exécutée. Ce comportement dangereux a été décidé par les ingénieurs de *Microsoft* car trop peu de méthodes du *framework* sont pour l'instant annotées avec un contrat de fiabilité suffisant. Notez que seuls les trois contrats de fiabilité suivants sont considérés comme suffisant pour une CER :

```
[ReliabilityContract(Consistency.MayCorruptInstance, Cer.MayFail)]
[ReliabilityContract(Consistency.WillNotCorruptState, Cer.MayFail)]
[ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
```

## Les finaliseurs critiques

Le CLR considère que le code d'un finaliseur d'une classe qui dérive de la classe `System.Runtime.ConstrainedExecution.CriticalFinalizerObject` est une CER. On parle alors de *finaliseur critique*. En plus d'être des CER, les finaliseurs critiques seront, au sein d'une même collecte, tous exécutés après l'exécution de tous les finaliseurs normaux. Cela garantit que les ressources les plus critiques, celles sur lesquelles les objets gérés dépendent, seront libérées en dernier.

Le mécanisme de finaliseur critique est notamment exploité par les classes du *framework* responsables du cycle de vie d'un handle win32 (à savoir `System.Runtime.InteropServices.CriticalHandle` et `System.Runtime.InteropServices.SafeHandle` décrites en page 278).

Les autres classes du *framework* qui dérivent de la classe `CriticalFinalizerObject` sont `System.Security.SecureString` (voir page 227) et `System.Threading.ReaderWriterLock` (voir page 158).

## Les régions critiques

Un second mécanisme de renforcement de la fiabilité est prévu en plus des CERs. L'idée est de fournir une information au CLR pour qu'il sache quand une ressource partagée entre plusieurs threads est mise à jour. Une portion de code responsable de la mise à jour d'une telle ressource est nommée *région critique* (*Critical Region* en anglais ou *CR*). Pour définir le commencement et la fin d'une région critique il suffit d'appeler les méthode `BeginCriticalRegion()` et `EndCriticalRegion()` de la classe `Thread`.

Si une exception asynchrone survient dans une région critique l'état de la ressource partagée entre plusieurs threads est potentiellement corrompu. Le thread courant va être détruit mais cela ne suffit pas à garantir que l'application va pouvoir continuer normalement son exécution. En effet, d'autres threads peuvent avoir accès aux données corrompues et ainsi, avoir un comportement imprévisible. La seule solution envisageable est de décharger le domaine d'application courant et c'est effectivement ce qu'il se passe. Ce comportement de propager un problème local à un thread au domaine d'application tout entier est nommé *politique de l'escalade* (*escalation policy* en anglais). Un second effet inhérent à la notion de région critique est de forcer le CLR à décharger le domaine d'application courant si une demande d'allocation mémoire échoue.

Si vous avez recours aux classes de verrous du *framework* (la classe `Monitor` et mot clé `lock`, la classe `ReaderWriterLock` etc) pour synchroniser les accès à vos ressources partagées, vous n'aurez pas besoin de définir explicitement les régions critiques de votre code. En effet, les méthodes d'acquisition et de libération du verrou de ses classes appèlent implicitement les méthodes `BeginCriticalRegion()` et `EndCriticalRegion()`.

En conséquence, les régions critiques sont principalement à utiliser si vous développez votre propre mécanisme de synchronisation, ce qui, vous en conviendrez, n'est pas une tâche courante.

# CLI et CLS

Sous ces deux acronymes se cache la magie qui permet à .NET de supporter plusieurs langages. Le *CLI* (*Common Langage Infrastructure*) est une spécification décrivant les contraintes respec-

tées par le CLR et les assemblages. Une couche logicielle qui supportent les contraintes du CLI est à même de gérer l'exécution des applications .NET. Cette spécification est produite par l'ECMA. Elle est disponible sur le site de l'ECMA à l'URL : `http://www.ecma-international.org/publications/standards/ECMA-335.HTM`

## *Introduction aux contraintes imposées aux langages .NET*

Pour que les assemblages, compilés à partir d'un langage, puissent être gérés par le CLR (ou par une couche logicielle supportant le CLI) et utiliser toutes les classes et outils du *framework* .NET, il faut que le langage et son compilateur respectent un ensemble de contraintes appelées *CLS* (*Common Langage Specification*). Parmi ces contraintes on peut inclure le support des types du CTS mais ce n'est pas la seule contrainte.

Les contraintes imposées aux langages et à leurs compilateurs par le CLS sont nombreuses. En voici quelques-unes parmi les plus couramment citées :

- Le langage doit prévoir une syntaxe pour résoudre le cas d'une classe qui implémente deux interfaces qui ont un conflit de définitions de méthodes. Il y a conflit de définitions de méthodes lorsque deux méthodes, une dans chaque interface implémentée par la classe, ont le même nom et la même signature. Le CLS impose que la classe doit implémenter deux méthodes distinctes.
- Seulement certains types primitifs sont compatibles avec le CLS. Par exemple le type `ushort` de C# n'est pas compatible avec le CLS.
- Les types des paramètres des méthodes publiques doivent être *CLS compliant*. Cette notion de *CLS compliant* est présentée quelques lignes plus loin.
- Un objet lancé dans une exception doit être une instance de la classe `System.Exception`, ou d'une classe dérivée de celle-ci.

La liste exhaustive de ces contraintes est disponible dans les **MSDN** à l'article « **Common Language Specification** ».

La compatibilité d'un langage avec le CLS n'est pas forcément totale et on observe deux niveaux de compatibilité :

- Un langage supporte la « *compatibilité consommateur* » s'il peut instancier et utiliser les membres publics des classes publiques contenues dans des assemblages compatibles avec le CLS.
- Un langage supporte la « *compatibilité extenseur* » s'il peut produire des classes dérivant de classes publiques contenues dans des assemblages compatibles avec le CLS. Cette compatibilité induit la compatibilité consommateur.

Les langages C#, VB.NET et C++/CLI supportent ces deux niveaux de compatibilité.

## *Le point de vue du développeur*

Un assemblage est compatible avec le CLS (on dit *CLS compliant* en anglais) si les éléments suivants sont compatibles avec le CLS :

- La définition des types publics.

- La définition des membres publics et des membres protégés des types publics.
- Les paramètres des méthodes publiques et des méthodes protégées.

Ce qui veut dire que le code des classes et méthodes privées n'a pas à être compatible avec la CLS.

Le développeur a tout intérêt à développer des librairies compatibles avec le CLS. Il lui sera ainsi plus aisé de réutiliser ces classes dans le futur. Heureusement, le développeur n'a pas à être spécialiste des contraintes imposées par le CLS pour vérifier si ses classes sont compatibles avec le CLS. Vous pouvez, grâce à l'attribut `System.CLSCompliantAttribute`, faire vérifier par le compilateur si les éléments de vos applications (assemblages, classes, méthodes...) sont compatibles avec le CLS. Par exemple :

*Exemple 4-15 :*

```
using System ;
[assembly : CLSCompliantAttribute(true)]
namespace CLSCompliantTest {
   public class Program {
      public static void Fct(ushort i ) { ushort j = i; }
      static void Main(){}
   }
}
```

Le compilateur génère un avertissement car le type `ushort`, qui n'est pas compatible avec le CLS, est utilisé comme type d'un paramètre d'une méthode publique d'une classe publique. En revanche, l'utilisation du type `ushort` à l'intérieur du corps de la méthode `Fct()` ne provoque pas d'erreur ni d'avertissement de compilation.

Avec l'attribut `CLSCompliantAttribute`, on peut aussi indiquer au compilateur de ne pas tester la compatibilité avec le CLS pour la méthode `Fct()` tout en gardant la vérification de la compatibilité avec le CLS dans le reste du programme comme ceci :

*Exemple 4-16 :*

```
using System ;
[assembly : CLSCompliantAttribute(true)]
namespace CLSCompliantTest {
   public class Program {
      [CLSCompliantAttribute(false)]
      public static void Fct(ushort i ) { ushort j = i ; }
      static void Main(){}
   }
}
```

Comprenez bien que la méthode `Fct()` ne peut pas être appelée à partir du code d'un langage ne connaissant pas le type `ushort`.

# 5

# *Processus, threads et gestion de la synchronisation*

Nous exposons ici les notions fondamentales que sont les processus et les threads, dans l'architecture des systèmes d'exploitation de type *Windows NT/2000/XP*. Il faut avoir à l'esprit que le CLR (ou moteur d'exécution) décrit dans le chapitre précédent est une couche logicielle chargée dans un processus par un hôte du moteur d'exécution lorsqu'un assemblage .NET est lancé.

## *Introduction*

Un *processus* (*process* en anglais) est concrètement une zone mémoire contenant des ressources. Les processus permettent au système d'exploitation de répartir son travail en plusieurs *unités fonctionnelles*.

Un processus possède une ou plusieurs *unités d'exécution* appelée(s) *threads*. Un processus possède aussi un espace d'adressage virtuel privé accessible en lecture et écriture, seulement par ses propres threads.

Dans le cas des programmes .NET, un processus contient aussi dans son espace mémoire la couche logicielle appelée CLR ou moteur d'exécution. La description du CLR fait l'objet du chapitre précédent. Cette couche logicielle est chargée dès la création du processus par l'hôte du moteur d'exécution (ceci est décrit page 94).

Un thread ne peut appartenir qu'à un processus et ne peut utiliser que les ressources de ce processus. Quand un processus est créé par le système d'exploitation, ce dernier lui alloue automatiquement un thread appelé *thread principal* (*main thread* ou *primary thread* en anglais). C'est ce thread qui exécute l'hôte du moteur d'exécution, le chargeur du CLR.

Une *application* est constituée d'un ou plusieurs processus coopérants. Par exemple l'environnement de développement *Visual Studio* est une application, qui peut utiliser un processus pour éditer les fichiers sources et un processus pour la compilation.

Sous les systèmes d'exploitation *Windows NT/2000/XP*, on peut visualiser à un instant donné toutes les applications et tous les processus en lançant le *gestionnaire des tâches* (*task manager* en anglais). Il est courant d'avoir une trentaine de processus en même temps, même si vous avez ouvert un petit nombre d'applications. En fait le système exécute un grand nombre de processus, un pour la gestion de la session courante, un pour la barre des tâches, et bien d'autres encore.

# *Les processus*

## *Introduction*

Dans un système d'exploitation *Windows* 32 bits, tournant sur un processeur 32 bits, un processus peut être vu comme un espace linéaire mémoire de 4Go (2^32 octets), de l'adresse 0x00000000 à 0xFFFFFFFF. Cet espace de mémoire est dit privé, car inaccessible par les autres processus. Cet espace se partage en 2Go pour le système et 2Go pour l'utilisateur. *Windows* et certains processeurs s'occupent de faire l'opération de translation entre cet espace d'adressage virtuel et l'espace d'adressage réel.

Si N processus tournent sur une machine il n'est (heureusement) pas nécessaire d'avoir Nx4Go de RAM.

- *Windows* alloue seulement la mémoire nécessaire à chaque processus, 4Go étant la limite supérieure dans un environnement 32 bits.
- Un mécanisme de *mémoire virtuelle* du système sauve sur le disque dur et charge en RAM des « morceaux » de processus appelés pages mémoire. Chaque page a une taille de 4Ko. Là encore tout ceci est transparent pour le développeur et l'utilisateur.

## *La classe System.Diagnostics.Process*

Une instance de la classe `System.Diagnostics.Process` référence un processus. Les processus qui peuvent être référencés sont :

- Le processus courant dans lequel l'instance est utilisée.
- Un processus sur la même machine autre que le processus courant.
- Un processus sur une machine distante.

Les méthodes et champs de cette classe permettent de créer, détruire, manipuler ou obtenir des informations sur ces processus. Nous exposons ici quelques techniques courantes d'utilisation de cette classe.

## *Créer et détruire un processus fils*

Le petit programme suivant crée un nouveau processus, appelé *processus fils*. Dans ce cas le processus initial est appelé *processus parent*. Ce processus fils exécute le bloc note. Le thread du

processus parent attend une seconde avant de tuer le processus fils. Le programme a donc pour effet d'ouvrir et de fermer le bloc note :

*Exemple 5-1 :*

```
using System.Diagnostics ;
using System.Threading ;
class Program {
   static void Main() {
      // Crée un processus fils qui lance notepad.exe
      // avec le fichier texte hello.txt.
      Process process = Process.Start("notepad.exe", "hello.txt") ;
      // Endors le thread 1 seconde.
      Thread.Sleep(1000) ;
      // Tue le processus fils.
      process.Kill();
   }
}
```

La méthode statique `Start()` peut utiliser les associations qui existent sur un système d'exploitation, entre un programme et une extension de fichier. Concrètement ce programme a le même comportement si l'on écrit :

```
Process process = Process.Start("hello.txt") ;
```

Par défaut un processus fils hérite du contexte de sécurité de son processus parent. Cependant une version surchargée de la méthode `Process.Start()` permet de lancer le processus fils dans le contexte de sécurité de n'importe quel utilisateur pourvu que vous ayez pu fournir le couple login/mot de passe dans une instance de la classe `System.Diagnostics.ProcessStartInfo`.

## *Empêcher de lancer deux fois le même programme sur la même machine*

Cette fonctionnalité est requise par de nombreuses applications. En effet, il est courant qu'il n'y ait pas de sens à lancer simultanément plusieurs fois une même application sur la même machine. Par exemple il n'y a pas de sens à lancer plusieurs fois *WindowMessenger* sur la même machine.

Jusqu'ici, pour satisfaire cette contrainte sous *Windows*, les développeurs utilisaient le plus souvent la technique dite du « *mutex nommé* » décrite page 154. L'utilisation de cette technique pour satisfaire cette contrainte souffre des défauts suivants :

- Il y a le risque faible, mais potentiel, que le nom du mutex soit utilisé par une autre application, auquel cas cette technique ne marche absolument plus et peut provoquer des bugs difficiles à détecter.
- Cette technique ne peut résoudre le cas général où l'on n'autorise que N instances de l'application.

Grâce aux méthodes statiques `GetCurrentProcess()` (qui retourne le processus courant) et `GetProcesses()` (qui retourne tous les processus lancés sur la machine) de la classe `System.`

Diagnostics.Process, ce problème trouve une solution élégante et très facile à implémenter exposée par le programme suivant :

*Exemple 5-2 :*

```
using System.Diagnostics ;
class Program {
   static void Main() {
      if ( TestSiDejaLance() ) {
         System.Console.WriteLine("Ce programme est déjà  lancé.") ;
      }
      else{
         // ici le code de l'application
      }
   }
   static bool TestSiDejaLance() {
      Process processCurrent = Process.GetCurrentProcess() ;
      Process[] processes = Process.GetProcesses() ;
      foreach ( Process process in processes )
         if ( processCurrent.Id != process.Id )
            if ( processCurrent.ProcessName == process.ProcessName )
               return true ;
      return false ;
   }
}
```

La méthode GetProcesses() peut aussi retourner tous les processus sur une machine distante en communiquant comme argument le nom de la machine distante.

## *Terminer le processus courant*

Vous pouvez décider de terminer le processus courant en appelant une des méthodes statiques Exit(int exitCode) ou FailFast(string message) de la classe System.Environment.

La méthode Exit() représente l'alternative de choix. Le processus se terminera proprement et retournera au système d'exploitation le code de sorti spécifié. La terminaison est propre dans le sens où tous les finaliseurs des objets courants sont correctement appelés et les éventuelles exécutions de blocs finally en attente seront effectués par les différents threads. En contrepartie, la terminaison du processus prendra un certain temps.

Comme son nom l'indique, l'alternative FailFast() termine rapidement le processus. Les précautions citées pour la méthode Exit() ne sont pas prises. Une erreur fatale contenant le message précisé est loguée par le système d'exploitation. Cette méthode est à utiliser si lors de la détection d'un problème non récupérable vous jugez que la continuation du programme peut potentiellement engendrer une corruption de donnée.

# *Les threads*

## *Introduction*

Un thread comprend :

- Un compteur d'instructions, qui pointe vers l'instruction en cours d'exécution.
- Une pile.
- Un ensemble de valeurs pour les registres, définissant une partie de l'état du processeur exécutant le thread.
- Une zone privée de données.

Tous ces éléments sont rassemblés sous le nom de *contexte d'exécution du thread*. L'espace d'adressage et par conséquent toutes les ressources qui y sont stockées, sont communs à tous les threads d'un même processus.

Nous ne parlerons pas de l'exécution des threads en *mode noyau* et en *mode utilisateur*. Ces deux modes, utilisés par *Windows* depuis bien avant .NET, existent toujours puisqu'ils se situent dans une couche en dessous du CLR. Néanmoins ces modes ne sont absolument pas visibles du *framework* .NET.

L'utilisation en parallèle de plusieurs threads constitue souvent une réponse naturelle à l'implémentation des algorithmes. En effet, les algorithmes utilisés dans les logiciels sont souvent constitués de tâches dont les exécutions peuvent se faire en parallèle. Attention, utiliser une grande quantité de threads génère beaucoup de *context switching*, et finalement nuit aux performances.

En outre, nous constatons depuis quelques années que la loi de *Moore* qui prédisait un doublement de la rapidité d'exécution des processeurs n'est plus vérifiée. Leur fréquence semble stagner autour de 3/4GHz. Cela est du à des limites physiques qui prendront quelques temps à être surmontées. Aussi, pour continuer la course aux performances, les grands fabricants de processeurs tels que *AMD* et *Intel* s'orientent vers des solutions types multi processeurs dans un seul chip. En conséquence, on peut s'attendre à voir proliférer ce type d'architecture dans les prochaines années. La seule solution pour améliorer les performances des applications sera alors d'avoir un recours massif au multithreading, d'où l'importance des notions présentées dans le présent chapitre.

## *Notion de thread géré*

Il faut bien comprendre que les threads qui exécutent les applications .NET sont bien ceux de *Windows*. Cependant on dit qu'un thread est géré quand le CLR connaît ce dernier. Concrètement, un thread est géré s'il est créé par du code géré. Si le thread est créé par du code non géré, alors il n'est pas géré. Cependant un tel thread devient géré dès qu'il exécute du code géré.

Un thread géré se distingue d'un thread non géré par le fait que le CLR crée une instance de la classe `System.Threading.Thread` pour le représenter et le manipuler. En interne, le CLR garde une liste des threads gérés nommée *ThreadStore*.

Le CLR fait en sorte que chaque thread géré soit exécuté au sein d'un domaine d'application, à un instant donné. Cependant un thread n'est absolument pas cantonné à un domaine d'application, et il peut en changer au cours du temps. La notion de domaine d'application est présentée page 87.

Dans le domaine de la sécurité, l'utilisateur principal d'un thread géré est indépendant de l'utilisateur principal du thread non géré sous-jacent.

## *Le multitâche préemptif*

On peut se poser la question suivante : mon ordinateur a un processeur (voire deux) et pourtant le gestionnaire des tâches indique qu'une centaine de threads s'exécutent simultanément sur ma machine ! Comment cela est-il possible ?

Cela est possible grâce au *multitâche préemptif* qui gère *l'ordonnancement des threads*. Une partie du noyau de *Windows*, appelée *répartiteur* (*scheduler* en anglais), segmente le temps en portions appelées *quantum* (appelées aussi *time slices*). Ces intervalles de temps sont de l'ordre de quelques millisecondes et ne sont pas de durée constante. Pour chaque processeur, chaque quantum est alloué à un seul thread. La succession très rapide des threads donne l'illusion à l'utilisateur que les threads s'exécutent simultanément. On appelle « *context switching* » l'intervalle entre deux quantums consécutifs. Un avantage de cette méthode est que les threads en attente d'une ressource n'ont pas d'intervalle de temps alloué jusqu'à la disponibilité de la ressource.

L'adjectif « préemptif » utilisé pour qualifier une telle gestion du multitâche vient du fait que les threads sont interrompus d'une manière autoritaire par le système. Pour les curieux sachez que durant le context switching, le système d'exploitation place une instruction de saut vers le prochain context switching dans le code qui va être exécuté par le prochain thread. Cette instruction est de type *interruption software*. Si le thread doit s'arrêter avant de rencontrer cette instruction (par exemple parce qu'il est en attente d'une ressource) cette instruction est automatiquement enlevée et le context switching a lieu prématurément.

L'inconvénient majeur du multitâche préemptif est la nécessité de protéger les ressources d'un accès anarchique avec des mécanismes de synchronisation. Il existe théoriquement un autre modèle de la gestion du multitâche, dit *multitâche coopératif*, où la responsabilité de décider quand donner la main incombe au threads eux-mêmes, mais ce modèle est dangereux car les risques de ne jamais rendre la main sont trop grands. Comme nous l'expliquons en page 100, ce mécanisme est cependant utilisé en interne pour optimiser les performances de certains serveurs tels que *SQL Server2005*. En revanche, les systèmes d'exploitation *Windows* n'implémentent plus que le multitâche préemptif.

## *Les niveaux de priorité d'exécution*

Certaines tâches sont plus prioritaires que d'autres. Concrètement elles méritent que le système d'exploitation leur alloue plus de temps processeur. Par exemple, certains pilotes de périphériques pris en charge par le processeur principal, ne doivent pas être interrompus. Une autre catégorie de tâches prioritaires sont les interfaces graphiques utilisateurs. En effet, les utilisateurs n'aiment pas attendre que l'interface se rafraîchisse.

Ceux qui viennent du monde win32, savent bien que le système d'exploitation *Windows*, sous-jacent au CLR, assigne un numéro de priorité à chaque thread entre 0 et 31. Cependant, il n'est pas dans la philosophie de la gestion des threads sous .NET de travailler directement avec cette valeur, parce que :

- Elle est peu explicite.
- Cette valeur est susceptible de changer au cours du temps.

### *Niveau de priorité d'un processus*

Vous pouvez assigner une priorité à vos processus avec la propriété `PriorityClass` de la classe `System.Diagnostics.Process`. Cette propriété est de type l'énumération `System.Diagnostics.ProcessPriorityClass` qui a les valeurs suivantes :

| Valeur de `ProcessPriorityClass` | Niveau de priorité correspondant |
|---|---|
| `Low` | 4 |
| `BelowNormal` | 6 |
| `Normal` | 8 |
| `AboveNormal` | 10 |
| `High` | 13 |
| `RealTime` | 24 |

Le processus propriétaire de la fenêtre en premier plan voit sa priorité incrémentée d'une unité si la propriété `PriorityBoostEnabled` de la classe `System.Diagnostics.Process` est positionnée à `true`. Cette propriété est par défaut positionnée à `true`. Cette propriété n'est accessible sur une instance de la classe `Process`, que si celle-ci référence un processus sur la même machine.

Vous avez la possibilité de changer la priorité d'un processus en utilisant le *gestionnaire des tâches* avec la manipulation : *Click droit sur le processus choisi* ▶ *Définir la priorité* ▶ Choisir parmi les six valeurs proposées à savoir *Temps réel, Haute ; Supérieure à la normale ; Normale ; Inférieure à la normale ; Basse*.

Les systèmes d'exploitation *Windows* ont un *processus d'inactivité* (*idle* en anglais) qui a la priorité 0. Cette priorité n'est accessible à aucun autre processus. Par définition l'activité des processeurs, notée en pourcentage, est :

100% moins le pourcentage de temps passé dans le thread du processus d'inactivité.

## *Niveau de priorité d'un thread*

Chaque thread peut définir sa propre priorité par rapport à celle de son processus, avec la propriété `Priority` de la classe `System.Threading.Thread`. Cette propriété est de type l'énumération `System.Threading.ThreadPriority` qui présente les valeurs suivantes :

| Valeur de `ThreadPriority` | Effet sur la priorité du thread |
|---|---|
| `Lowest` | -2 unités par rapport à la priorité du processus |
| `BelowNormal` | -1 unité par rapport à la priorité du processus |
| `Normal` | même priorité que la priorité du processus |
| `AboveNormal` | +1 unité par rapport à la priorité du processus |
| `Highest` | +2 unités par rapport à la priorité du processus |

Dans la plupart de vos applications, vous n'aurez pas à modifier la priorité de vos processus et threads, qui par défaut, est assignée à `Normal`.

## *La classe System.Threading.Thread*

Le CLR associe automatiquement une instance de la classe `System.Threading.Thread` à chaque thread géré. Vous pouvez utiliser cet objet pour manipuler le thread à partir d'un autre thread ou à partir du thread lui-même. Vous pouvez obtenir cet objet associé au thread courant avec la propriété statique `CurrentThread` de la classe `System.Threading.Thread` :

```
using System.Threading ;
...
Thread threadCurrent = Thread.CurrentThread ;
```

Une fonctionnalité de la classe `Thread`, bien pratique pour déboguer une application multi-thread (multitâches), est la possibilité de pouvoir nommer ses threads avec une chaîne de caractères :

```
threadCurrent.Name = "thread Foo" ;
```

## *Créer et joindre un thread*

Pour créer un nouveau thread dans le processus courant, il suffit de créer une nouvelle instance de la classe `Thread`. Les différents constructeurs de cette classe prennent en argument un délégué de type `System.Threading.ThreadStart` ou de type `System.Threading.ParametrizedThreadStart` qui référence la méthode qui va être exécutée par le thread créé. L'utilisation d'un délégué de type `ParametrizedThreadStart` permet de passer un objet à la méthode qui va être exécutée par un nouveau thread. Des constructeurs de la classe thread acceptent aussi un paramètre entier permettant de fixer la taille maximale en octet de la pile du thread créé. Cette taille doit être au moins égale à 128Ko (i.e 131072 octets). Après qu'une instance de type `Thread` est créée, il faut appeler la méthode `Thread.Start()` pour effectivement démarrer le thread :

*Exemple 5-3 :*

```
using System.Threading ;
class Program {
   static void f1() { System.Console.WriteLine("f1") ; }
   void f2() { System.Console.WriteLine("f2") ; }
   static void f3(object obj) { System.Console.WriteLine(
                                "f3 obj = {0}",obj) ; }
   static void Main() {

      // Spécification explicite de la délégation ThreadStart.
      Thread t1 = new Thread(new ThreadStart(f1));

      Program program = new Program();
      // Utilisation de la possibilité C#2 d'inférer
      // le type d'un délégué.
      Thread t2 = new Thread( program.f2 );
```

```
        // Inférence d'un délégué de type ParametrizedThreadStart
        // puisque f3() a un unique paramétre de type object.
        Thread t3 = new Thread( f3 );

        t1.Start() ; t2.Start() ; t3.Start("hello") ;
        t1.Join() ;  t2.Join() ;  t3.Join() ;
    }
}
```

Ce programme affiche :

```
f1
f2
f3 obj = hello
```

Dans cet exemple, nous utilisons la méthode `Join()`, qui suspend l'exécution du thread courant jusqu'à ce que le thread sur lequel s'applique cette méthode ait terminé. Cette méthode existe aussi en une version surchargée qui prend en paramètre un entier qui définie le nombre maximal de millisecondes à attendre la fin du thread (i.e un *time out*). Cette version de `Join()` retourne un booléen positionné à `true` si le thread s'est effectivement terminé.

## *Suspendre l'activité d'un thread*

Vous avez la possibilité de suspendre l'activité d'un thread pour une durée déterminée en utilisant la méthode `Sleep()` de la classe `Thread`. Vous pouvez spécifier la durée au moyen d'un entier qui désigne un nombre de millisecondes ou avec une instance de la structure `System.TimeSpan`. Bien qu'une telle instance puisse spécifier une durée avec la précision du dixième de milliseconde (100 nano secondes) la granularité temporelle de la méthode `Sleep()` n'est qu'à la milliseconde.

```
// Le thread courant est suspendu pour une seconde.
Thread.Sleep(1000) ;
```

On peut aussi suspendre l'activité d'un thread en appelant la méthode `Suspend()` de la classe `Thread`, à partir d'un autre thread ou à partir du thread à suspendre. Dans les deux cas le thread se bloque jusqu'à ce qu'un autre thread appelle la méthode `Resume()` de la classe `Thread`. Contrairement à la méthode `Sleep()`, un appel à `Suspend()` ne suspend pas immédiatement le thread, mais le CLR suspendra ce thread au prochain point protégé rencontré. La notion de point protégé est présentée en page 120.

## *Terminer un thread*

Un thread géré peut se terminer selon plusieurs scénarios :

- Il sort de la méthode sur laquelle il avait commencé sa course (la méthode `Main()` pour le thread principal, la méthode référencée par le délégué `ThreadStart` pour les autres threads).
- Il s'auto interrompt (il se suicide).
- Il est interrompu par un autre thread.

Le premier cas étant trivial, nous ne nous intéressons qu'aux deux autres cas. Dans ces deux cas, la méthode `Abort()` peut être utilisée (par le thread courant ou par un thread extérieur). Elle provoque l'envoi d'une exception de type `ThreadAbortException`. Cette exception a la particularité d'être relancée automatiquement lorsqu'elle est rattrapée par un gestionnaire d'exception car le thread est dans un état spécial nommé `AbortRequested`. Seul l'appel à la méthode statique `ResetAbort()` (si on dispose de la permission nécessaire) dans le gestionnaire d'exception empêche cette propagation.

*Exemple 5-4 : Suicide du thread principal*

```
using System ;
using System.Threading ;
namespace ThreadTest{
   class Program {
      static void Main() {
         Thread t = Thread.CurrentThread ;
         try{
            t.Abort() ;
         }
         catch( ThreadAbortException ) {
            Thread.ResetAbort() ;
         }
      }
   }
}
```

Lorsqu'un thread `A` appelle la méthode `Abort()` sur un autre thread B, il est conseillé que `A` attende que `B` soit effectivement terminé en appelant la méthode `Join()` sur B.

Il existe aussi la méthode `Interrupt()` qui permet de terminer un thread lorsqu'il est dans un état d'attente (i.e bloqué sur une des méthodes `Wait()`, `Sleep()` ou `Join()`). Cette méthode a un comportement différent selon que le thread à terminer est dans un état d'attente ou non.

- Si le thread à terminer est dans un état d'attente lorsque `Interrupt()` est appelée par un autre thread, l'exception `ThreadInterruptedException` est lancée.
- Si le thread à terminer n'est pas dans un état d'attente lorsque `Interrupt()` est appelée, la même exception sera lancée dès que ce thread rentrera dans un état d'attente. Le comportement est le même si le thread à terminer appelle `Interrupt()` sur lui-même.

## *Notion de threads foreground et background*

La classe `Thread` présente la propriété booléenne `IsBackground`. Un *thread foreground* est un thread qui empêche la terminaison du processus tant qu'il n'est pas terminé. À l'opposé un *thread background* est un thread qui est terminé automatiquement par le CLR (par l'appel à la méthode `Abort()`) lorsqu'il n'y a plus de thread *foreground* dans le processus concerné. `IsBackground` est positionnée à `false` par défaut, ce qui fait que les threads sont *foreground* par défaut.

On pourrait traduire thread *foreground par thread de premier plan, et thread background par thread de fond.*

## *Diagramme d'états d'un thread*

La classe `Thread` a le champ `ThreadState` de type l'énumération `System.Threading.ThreadState`. Les valeurs de cette énumération sont :

```
Aborted             AbortRequested          Background
Running             Stopped                 StopRequested
Suspended           SuspendRequested        Unstarted
WaitSleepJoin
```

La description de chacun de ces états se trouve dans l'article **ThreadState Enumeration** des **MSDN**. Cette énumération est un *indicateur binaire*, c'est-à-dire que ses instances peuvent prendre plusieurs valeurs à la fois. Par exemple un thread peut être à la fois dans l'état `Running` `AbortRequested` et `Background`. La notion d'indicateur binaire est présentée page 369.

D'après ce qu'on a vu dans la section précédente, on peut définir le diagramme d'état simplifié suivant :

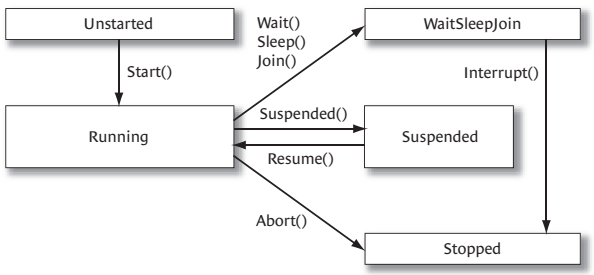


*Figure 5-1 : Diagramme d'état d'un thread, simplifié*

# *Introduction à la synchronisation des accès aux ressources*

En informatique, le mot *synchronisation* ne peut être utilisé que dans le cas des applications multithreads (mono ou multi-processus). En effet, la particularité de ces applications est d'avoir plusieurs unités d'exécution, d'où possibilité de conflits d'accès aux ressources. Les objets de synchronisation sont des objets partageables entre threads exécutés sur la même machine. Le propre d'un objet de synchronisation est de pouvoir bloquer un des threads utilisateur jusqu'à la réalisation d'une condition par un autre thread.

Comme nous allons le voir, il existe de nombreuses classes et mécanismes de synchronisation. Chacun répond à un ou plusieurs besoins spécifiques et il est nécessaire d'avoir assimilé tout ce chapitre avant de concevoir une application professionnelle multithreads utilisant la synchronisation. Nous nous sommes efforcé de souligner les différences, surtout les plus subtiles, qui existent entre les différents mécanismes. Quand vous aurez compris les différences, vous serez capable d'utiliser ces mécanismes.

Synchroniser correctement un programme est une des tâches du développement logiciel les plus subtiles. Le sujet remplit de nombreux ouvrages. Avant de vous plonger dans des spécifications

compliquées, soyez certains que l'utilisation de la synchronisation est incontournable. Souvent l'utilisation de quelques règles simples suffit à éviter d'avoir à gérer la synchronisation. Parmi ces règles, citons la règle d'affinité entre threads et ressources que nous décrivons un peu plus loin dans ce chapitre.

Soyez conscient que la difficulté de synchroniser les accès aux ressources d'un programme vient du dilemme entre une granularité des verrous trop fine et une granularité trop grossière. Si vous synchronisez trop grossièrement les accès à vos ressources, vous simplifierez votre code mais vous vous exposez à des problèmes de contention type goulot d'étranglement. Si vous les synchronisez trop finement, votre complexifierez votre code et à terme vous ne pourrez plus le maintenir. Vous êtes alors exposé aux problèmes de *deadlock* et de *race condition* décris ci après.

Avant d'aborder les mécanismes de synchronisation, il est nécessaire d'avoir une idée précise des notions de *race conditions* (*situations de compétition* en français) et de *deadlocks* (*interblocages* en français).

## *Race conditions*

Il s'agit d'une situation où des actions effectuées par des unités d'exécution différentes s'enchaînent dans un ordre illogique, entraînant des états non prévus.

Par exemple un thread `T` modifie une ressource `R`, rend les droits d'accès d'écriture à `R`, reprend les droits d'accès en lecture sur `R` et utilise `R` comme si son état était celui dans lequel il l'avait laissé. Pendant l'intervalle de temps entre la libération des droits d'accès en écriture et l'acquisition des droits d'accès en lecture, il se peut qu'un autre thread ait modifié l'état de `R`.

Un autre exemple classique de situation de compétition est le modèle producteur consommateur. Le producteur utilise souvent le même espace physique pour stocker les informations produites. En général on n'oublie pas de protéger cet espace physique des accès concurrents entre producteurs et consommateurs. On oublie plus souvent que le producteur doit s'assurer qu'un consommateur a effectivement lu une ancienne information avant de produire une nouvelle information. Si l'on ne prend pas cette précaution, on s'expose au risque de produire des informations qui ne seront jamais consommées.

Les conséquences de situations de compétition mal gérées peuvent être des failles dans un système de sécurité. Une autre application peut forcer un enchaînement d'actions non prévues par les développeurs. Typiquement il faut absolument protéger l'accès en écriture à un booléen qui confirme ou infirme une authentification. Sinon il se peut que son état soit modifié entre l'instant ou ce booléen est positionné par le mécanisme d'authentification et l'instant ou ce booléen est lu pour protéger des accès à des ressources. De célèbres cas de failles de sécurité dues à une mauvaise gestion des situations de compétition ont existé. Une de celles-ci concernait notamment le noyau *Unix*.

## *Deadlocks*

Il s'agit d'une situation de blocage à cause de deux ou plusieurs unités d'exécution qui s'attendent mutuellement. Par exemple :

Un thread `T1` acquiert les droits d'accès sur la ressource `R1`.

Un thread `T2` acquiert les droits d'accès sur la ressource `R2`.

`T1` demande les droits d'accès sur `R2` et attend, car c'est `T2` qui les possède.

`T2` demande les droits d'accès sur `R1` et attend, car c'est `T1` qui les possède.

`T1` et `T2` attendront donc indéfiniment, la situation est bloquée ! Il existe trois grandes approches pour éviter ce problème qui est plus subtil que la plupart des bugs que l'on rencontre.

- N'autoriser aucun thread à avoir des droits d'accès sur plusieurs ressources simultanément.
- Définir une relation d'ordre dans l'acquisition des droits d'accès aux ressources. C'est-à-dire qu'un thread ne peut acquérir les droits d'accès sur `R2` s'il n'a pas déjà acquis les droits d'accès sur `R1`. Naturellement la libération des droits d'accès se fait dans l'ordre inverse de l'acquisition.
- Systématiquement définir un temps maximum d'attente (*timeout*) pour toutes les demandes d'accès aux ressources et traiter les cas d'échec. Pratiquement tous les mécanismes de synchronisation .NET offrent cette possibilité.

Les deux premières techniques sont plus efficaces mais aussi plus difficile à implémenter. En effet, elles nécessitent chacune une contrainte très forte et difficile à maintenir durant l'évolution de l'application. En revanche les situations d'échecs sont inexistantes.

Les gros projets utilisent systématiquement la troisième technique. En effet, si le projet est gros, le nombre de ressources est en général très grand. Dans ces projets, les conflits d'accès simultanés à une ressource sont donc des situations marginales. La conséquence est que les situations d'échec sont, elles aussi, marginales. On dit d'une telle approche qu'elle est optimiste. Dans le même esprit, on décrit en page 727 le modèle de gestion optimiste des accès concurrents à une base de données.

# *Synchronisation avec les champs volatiles et la classe Interlocked*

## *Les champs volatiles*

Un champ d'un type peut être accédé par plusieurs threads. Supposons que ces accès, en lecture ou en écriture, ne soient pas synchronisés. Dans ce cas, les nombreux mécanismes internes du CLR pour gérer le code font qu'il n'y a pas de garantie que chaque accès en lecture au champ charge la valeur la plus récente. Un champ déclaré volatile vous donne cette garantie. En langage C#, un champ est déclaré volatile si le mot-clé `volatile` est écrit devant sa déclaration.

Tous les champs ne peuvent pas être volatiles. Il y a une restriction sur le type du champ. Pour qu'un champ puisse être volatile, il faut que son type soit dans cette liste :

- Un type référence.
- Un pointeur (dans une zone de code non protégée).
- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `char`, `float`, `bool` (`double`, `long` et `ulong`, à la condition de travailler avec une machine 64 bits).
- Une énumération dont le type sous-jacent est parmi : `byte`, `sbyte`, `short`, `ushort`, `int`, `uint` (`double`, `long` et `ulong` à condition de travailler avec une machine 64 bits).

Comme vous l'aurez remarquez, seuls les types dont la valeur ou la référence fait au plus le nombre d'octets d'un entier natif (quatre ou huit selon le processeur sous-jacent) peuvent être

volatiles. Cela implique que les opérations concurrentes sur une valeur de plus de ce nombre d'octets (une grosse structure par exemple) doivent être protégées en utilisant les mécanismes de synchronisation présentés ci-après.

## *La classe System.Threading.Interlocked*

L'expérience a montré que les ressources à protéger dans un contexte multithreads sont souvent des variables entières. Les opérations les plus courantes réalisées par les threads sur ces variables entières partagées sont l'incrémentation et la décrémentation d'une unité et l'addition de deux variables. Le *framework* .NET prévoit donc un mécanisme spécial avec la classe `System.Threading.Interlocked` pour ces opérations très spécifiques, mais aussi très courantes. Cette classe à les méthodes statiques `Increment()`, `Decrement()` et `Add()` qui respectivement incrémente, décrémente et additionne des entiers de type `int` ou `long` passés par référence. On dit que l'utilisation de la classe `Interlocked` rend ces opérations *atomiques* (c'est-à-dire indivisibles, comme ce que l'on pensait il y a quelques décennies pour les atomes de la matière).

Le programme suivant présente l'accès concurrent de deux threads à la variable entière `compteur`. Un thread l'incrémente cinq fois tandis que l'autre la décrémente cinq fois.

*Exemple 5-5 :*

```
using System.Threading ;
class Program {
   static long compteur = 1 ;
   static void Main() {
      Thread t1 = new Thread(f1) ;
      Thread t2 = new Thread(f2) ;
      t1.Start() ; t2.Start() ; t1.Join() ; t2.Join() ;
   }
   static void f1() {
      for (int i = 0 ; i < 5 ; i++){
         Interlocked.Increment(ref compteur);
         System.Console.WriteLine("compteur++ {0}", compteur) ;
         Thread.Sleep(10) ;
      }
   }
   static void f2() {
      for (int i = 0 ; i < 5 ; i++){
         Interlocked.Decrement(ref compteur);
         System.Console.WriteLine("compteur-- {0}", compteur) ;
         Thread.Sleep(10) ;
      }
   }
}
```

Ce programme affiche ceci (d'une manière non déterministe, c'est-à-dire que l'affichage pourrait varier d'une exécution à une autre) :

```
compteur++ 2
compteur-- 1
compteur++ 2
```

```
compteur-- 1
compteur++ 2
compteur-- 1
compteur++ 2
compteur-- 1
compteur-- 0
compteur++ 1
```

Si on n'endormait pas les threads 10 millièmes de seconde à chaque modification, les threads auraient le temps de réaliser leurs tâches en un quantum et il n'y aurait pas l'entrelacement des exécutions, donc pas d'accès concurrent.

### *Autre possibilité d'utilisation de la classe Interlocked*

La classe `Interlocked` permet de rendre atomique une autre opération usuelle qui est la recopie de l'état d'un objet source vers un objet destination au moyen de la méthode statique surchargée `Exchange()`. Elle permet aussi de rendre atomique l'opération de comparaison des états de deux objets, et dans le cas d'égalité, la recopie de cet état vers un troisième objet au moyen de la méthode statique surchargée `CompareExchange()`.

## *Synchronisation avec la classe System.Threading.Monitor et le mot-clé lock*

Le fait de rendre des opérations simples atomiques (des opérations comme l'incrémentation, la décrémentation ou la recopie d'un état), est indéniablement important mais est loin de couvrir tous les cas où la synchronisation est nécessaire. La classe `System.Threading.Monitor` permet de rendre n'importe quelle portion de code exécutable par un seul thread à la fois. On appelle une telle portion de code une *section critique*.

### *Les méthodes Enter() et Exit()*

La classe `Monitor` présente les méthodes statiques `Enter(object)` et `Exit(object)`. Ces méthodes prennent un objet en paramètre. Cet objet constitue un moyen simple d'identifier de manière unique la ressource à protéger d'un accès concurrent. Lorsqu'un thread appelle la méthode `Enter()`, il attend d'avoir le droit exclusif de posséder l'objet référencé (il n'attend que si un thread a déjà ce droit). Une fois ce droit acquis et consommé, le thread libère ce droit en appelant `Exit()` sur ce même objet.

Un thread peut appeler `Enter()` plusieurs fois sur le même objet à la condition qu'il appelle `Exit()` autant de fois sur le même objet pour se libérer des droits exclusifs.
Un thread peut posséder des droits exclusifs sur plusieurs objets à la fois, mais cela peut mener à une situation de *deadlock*.
Il ne faut jamais appeler les méthodes `Enter()` et `Exit()` sur un objet de type valeur, comme un entier !
Il faut toujours appeler la méthode `Exit()` dans un bloc `finally` afin d'être certain de libérer les droits d'accès exclusifs quoi qu'il arrive.

Si dans l'Exemple 5-5, un thread doit élever la variable `compteur` au carré tandis que l'autre thread doit la multiplier par deux, il faudrait remplacer l'utilisation de la classe `Interlocked` par l'utilisation de la classe `Monitor`. Le code de `f1()` et `f2()` serait alors :

*Exemple 5-6 :*

```
using System.Threading ;
class Program {
   static long compteur = 1 ;
   static void Main() {
      Thread t1 = new Thread(f1) ;
      Thread t2 = new Thread(f2) ;
      t1.Start() ; t2.Start() ; t1.Join() ; t2.Join() ;
   }
   static void f1() {
      for (int i = 0 ; i < 5 ; i++){
         try{
            Monitor.Enter( typeof(Program) );
            compteur *= compteur ;
         }
         finally{ Monitor.Exit( typeof(Program) ) ; }
         System.Console.WriteLine("compteur^2 {0}", compteur) ;
         Thread.Sleep(10) ;
      }
   }
   static void f2() {
      for (int i = 0 ; i < 5 ; i++){
         try{
            Monitor.Enter( typeof(Program) );
            compteur *= 2 ;
         }
         finally{ Monitor.Exit( typeof(Program) ) ; }
         System.Console.WriteLine("compteur*2 {0}", compteur) ;
         Thread.Sleep(10) ;
      }
   }
}
```

Il est tentant d'écrire `compteur` à la place de `typeof(Program)` mais `compteur` est un membre statique de type valeur. Remarquez que les opérations « élévation au carré » et « multiplication par deux » n'étant pas commutatives, la valeur finale de `compteur` est ici non déterminée.

## *Le mot clé lock de C#*

Le langage C# présente le mot-clé `lock` qui remplace élégamment l'utilisation des méthode `Enter()` et `Exit()`. La méthode `f1()` pourrait donc s'écrire :

*Exemple 5-7 :*

```
using System.Threading ;
class Program {
```

```
   static long compteur = 1 ;
   static void Main() {
      Thread t1 = new Thread(f1) ;
      Thread t2 = new Thread(f2) ;
      t1.Start() ; t2.Start() ; t1.Join() ; t2.Join() ;
   }
   static void f1() {
      for (int i = 0 ; i < 5 ; i++){
         lock( typeof(Program) ) { compteur *= compteur ; }
         System.Console.WriteLine("compteur^2 {0}", compteur) ;
         Thread.Sleep(10) ;
      }
   }
   static void f2() {
      for (int i = 0 ; i < 5 ; i++){
         lock( typeof(Program) ) { compteur *= 2 ; }
         System.Console.WriteLine("compteur*2 {0}", compteur) ;
         Thread.Sleep(10) ;
      }
   }
}
```

À l'instar des blocs `for` et `if`, les blocs définis par le mot-clé `lock` ne sont pas tenus d'avoir des accolades s'ils ne contiennent qu'une instruction. On aurait donc pu écrire :

```
...
lock( typeof(Program) )
   compteur *= compteur;
...
```

L'usage du mot-clé `lock` provoque bien la création par le compilateur C# d'un bloc `try/finally` qui permet d'anticiper les levées d'exceptions. Vous pouvez le vérifier avec un des outils *Reflector* ou `ildasm.exe`.

## *Le pattern SyncRoot*

À l'instar des exemples précédents, on utilise en général la classe `Monitor` avec une instance de la clase `Type` du type courant à l'intérieur d'une méthode statique. De même, on se synchronise souvent sur le mot clé `this` à l'intérieur d'une méthode non statique. Dans les deux cas, on se synchronise sur un objet visible hors de la classe. Cela peut poser des problèmes si d'autres parties du code se synchronisent sur ces objets. Pour éviter ces problèmes potentiels, nous vous conseillons d'utiliser un membre privé `SyncRoot` de type `object`, statique ou non selon vos besoins :

*Exemple 5-8 :*

```
class Foo {
   private static object staticSyncRoot = new object();
   private object instanceSyncRoot = new object();
   public static void StaticFct() {
      lock (staticSyncRoot) { /*...*/ }
```

```
    }
    public void InstanceFct() {
       lock (instanceSyncRoot) { /*...*/ }
    }
}
```

L'interface `System.Collections.ICollection` présente la propriété `object SyncRoot{get;}`. La plupart des classes de collections (génériques ou non) implémentent cette interface. Aussi, vous pouvez vous servir de cette propriété pour synchroniser les accès aux éléments d'une collection. Ici, le pattern *SyncRoot* n'est pas vraiment appliqué puisque l'objet sur lequel on synchronise les accès n'est pas privé :

*Exemple 5-9 :*

```
using System.Collections.Generic ;
using System.Collections ;
public class Program {
   public static void Main() {
      List<int> list = new List<int>() ;
      // ...
      lock (((ICollection)list).SyncRoot) {
         foreach (int i in list) {
            // faire un traitement...
         }
      }
   }
}
```

## *Notion de classe thread-safe*

Une classe *thread-safe* est une classe dont chaque instance ne peut être accédée par plusieurs threads à la fois. Pour fabriquer une classe thread-safe, il suffit d'appliquer le *pattern SyncRoot* que l'on vient de voir à toutes ses méthodes d'instances. Un moyen efficace pour ne pas encombrer le code d'une classe que l'on souhaite être *thread-safe* est fournir une classe dérivée *wrapper thread-safe* comme ceci :

*Exemple 5-10 :*

```
class Foo {
   private class FooSynchronized : Foo {
      private object syncRoot = new object() ;
      private Foo m_Foo ;
      public FooSynchronized(Foo foo) { m_Foo = foo ; }
      public override bool IsSynchronized { get { return true ; } }
      public override void Fct1(){ lock(syncRoot) { m_Foo.Fct1() ; } }
      public override void Fct2(){ lock(syncRoot) { m_Foo.Fct1() ; } }
   }
   public virtual bool IsSynchronized { get { return false ; } }
   public static Foo Synchronized(Foo foo){
      if( ! foo.IsSynchronized )
```

```
            return new FooSynchronized( foo );
         return foo ;
      }
      public virtual void Fct1() { /*...*/ }
      public virtual void Fct2() { /*...*/ }
   }
```

Un autre moyen est d'avoir recours à l'attribut `System.Runtime.Remoting.Contexts.Synchronization` présenté un peu plus loin dans ce chapitre.

## *La méthode Monitor.TryEnter()*

```
public static bool TryEnter(object [,int] )
```

Cette méthode est similaire à `Enter()` mais elle n'est pas bloquante. Si les droits d'accès exclusifs sont déjà possédés par un autre thread, cette méthode retourne immédiatement et sa valeur de retour est `false`. On peut aussi rendre un appel à `TryEnter()` bloquant pour une durée limitée spécifiée en millisecondes. Puisque l'issue de cette méthode est incertaine, et que dans le cas où l'on acquerrait les droits d'accès exclusifs il faudrait les libérer dans un bloc `finally`, il est conseillé de sortir immédiatement de la méthode courante dans le cas où l'appel `TryEnter()` échouerait :

*Exemple 5-11 :*

```
using System.Threading ;
class Program {
   static void Main() {
      // À commenter pour tester le cas où TryEnter() retourne true.
      Monitor.Enter(typeof(Program)) ;
      Thread t1 = new Thread(f1) ;
      t1.Start() ; t1.Join() ;
   }
   static void f1() {
      bool bOwner = false ;
      try {
         if( ! Monitor.TryEnter( typeof(Program) ) )
            return;
         bOwner = true;
         // ...
      }
      finally {
         // Ne surtout pas appeler Monitor.Exit() si on a pas l'accès.
         // N'oubliez pas que l'on passe nécessairement par le bloc
         // finally, y compris si l'appel à TryEnter() retourne false.
         if( bOwner )
            Monitor.Exit( typeof(Program) );
      }
   }
}
```

## *Les méthodes Wait(), Pulse() et PulseAll() de la classe Monitor*

```
public static bool Wait(object [,int])
public static void Pulse(object)
public static void PulseAll(object)
```

Les trois méthodes `Wait()`, `Pulse()` et `PulseAll()` doivent être utilisées ensembles et ne peuvent être correctement comprises sans un petit scénario. L'idée est qu'un thread ayant les droits d'accès exclusifs à un objet décide d'attendre (en appelant `Wait()`) que l'état de l'objet change. Pour cela, ce thread doit accepter de perdre momentanément les droits d'accès exclusifs à l'objet afin de permettre à un autre thread de changer l'état de l'objet. Ce dernier doit signaler le changement avec la méthode `Pulse()`. Voici un petit scénario expliquant ceci dans les détails :

- Le thread `T1` possédant l'accès exclusif à l'objet `OBJ`, appelle la méthode `Wait(OBJ)` afin de s'enregistrer dans une liste d'attente passive de `OBJ`.
- Par cet appel `T1` perd l'accès exclusif à `OBJ`. Ainsi un autre thread `T2` prend l'accès exclusif à `OBJ` en appelant la méthode `Enter(OBJ)`.
- `T2` modifie éventuellement l'état de `OBJ` puis appelle `Pulse(OBJ)` pour signaler cette modification. Cet appel provoque le passage du premier thread de la liste d'attente passive de `OBJ` (en l'occurrence `T1`) en haut de la liste d'attente active de `OBJ`. Le premier thread de la liste active de `OBJ` a la garantie qu'il sera le prochain à avoir les droits d'accès exclusifs à `OBJ` dès qu'ils seront libérés. Il pourra ainsi sortir de son attente dans la méthode `Wait(OBJ)`.
- Dans notre scénario `T2` libère les droits d'accès exclusifs sur `OBJ` en appelant `Exit(OBJ)` et `T1` les récupère et sort de la méthode `Wait(OBJ)`.
- La méthode `PulseAll()` fait en sorte que les threads de la liste d'attente passive, passent tous dans la liste d'attente active. L'important est que les threads soient débloqués dans le même ordre qu'ils ont appelés `Wait()`.

Si `Wait(OBJ)` est appelée par un thread ayant appelé plusieurs fois `Enter(OBJ)`, ce thread devra appeler `Exit(OBJ)` le même nombre de fois pour libérer les droits d'accès à `OBJ`. Même dans ce cas, un seul appel à `Pulse(OBJ)` par un autre thread suffit à débloquer le premier thread.

Le programme suivant illustre cette fonctionnalité au moyen de deux threads `ping` et `pong` qui se obtiennent à tour de rôle les droits d'accès à un objet `balle` :

*Exemple 5-12 :*

```
using System.Threading ;
public class Program {
   static object balle = new object();
   public static void Main() {
      Thread threadPing = new Thread( ThreadPingProc ) ;
      Thread threadPong = new Thread( ThreadPongProc ) ;
      threadPing.Start() ; threadPong.Start() ;
      threadPing.Join() ;  threadPong.Join() ;
   }
   static void ThreadPongProc() {
      System.Console.WriteLine("ThreadPong: Hello!") ;
      lock (balle)
         for (int i = 0 ; i < 5 ; i++){
```

```
            System.Console.WriteLine("ThreadPong: Pong ") ;
            Monitor.Pulse(balle);
            Monitor.Wait(balle);
         }
      System.Console.WriteLine("ThreadPong: Bye!") ;
   }
   static void ThreadPingProc() {
      System.Console.WriteLine("ThreadPing: Hello!") ;
      lock (balle)
         for(int i=0 ; i< 5 ; i++){
            System.Console.WriteLine("ThreadPing: Ping ") ;
            Monitor.Pulse(balle);
            Monitor.Wait(balle);
         }
      System.Console.WriteLine("ThreadPing: Bye!") ;
   }
}
```

Ce programme affiche d'une manière non déterministe :

```
ThreadPing: Hello!
ThreadPing: Ping
ThreadPong: Hello!
ThreadPong: Pong
ThreadPing: Ping
ThreadPong: Pong
ThreadPing: Ping
ThreadPong: Pong
ThreadPing: Ping
ThreadPong: Pong
ThreadPing: Ping
ThreadPong: Pong
ThreadPing: Bye!
```

Le thread pong ne se termine pas et reste bloqué sur la méthode `Wait()`. Ceci résulte du fait que le thread pong a obtenu le droit d'accès exclusif sur l'objet `balle` en deuxième.

# *Synchronisation avec des mutex, des événements et des sémaphores*

La classe de base abstraite `System.Threading.WaitHandle` admet trois classes dérivées dont l'utilisation est bien connue de ceux qui ont déjà utilisé la synchronisation sous win32 :

- La classe `Mutex` (le mot mutex est la concaténation de mutuelle exclusion. En français on parle parfois de *mutant*)).
- La classe `AutoResetEvent` qui définit un événement à repositionnement automatique. La classe `ManualResetEvent` qui définit un événement à repositionnement manuelle. Ces deux classes dérivent de la classe `EventWaitHandle` qui représente un événement au sens large.

- La classe `Semaphore`.

La classe `WaitHandle` et ses classes dérivées, ont la particularité d'implémenter la méthode non statique `WaitOne()` et les méthodes statiques `WaitAll()`, `WaitAny()` et `SignalAndWait()`. Elles permettent respectivement d'attendre qu'un objet soit signalé, que tous les objets dans un tableau soient signalés, qu'au moins un objet dans un tableau soit signalé, de signaler un objet et d'attendre sur un autre. Contrairement à la classe `Monitor` et `Interlocked`, ces classes doivent être instanciées pour être utilisées. Il faut donc raisonner ici en terme d'objets de synchronisation et non d'objets synchronisés. Ceci implique que les objets passés en paramètre des méthodes statiques `WaitAll()`, `WaitAny() et SignalAndWait()` sont soit des mutex, soit des événements soit des sémaphores.

## *Partage d'objets de synchronisation*

Il est important de noter une autre grosse distinction entre le modèle de synchronisation proposé par les classes dérivées de `WaitHandle` et celui de la classe `Monitor`. L'utilisation de la classe `Monitor` se cantonne à un seul processus. L'utilisation des classes dérivées de `WaitHandle` fait appel à des objets win32 non gérés qui peuvent être connus de plusieurs processus de la même machine. Pour cela, certains constructeurs des classes dérivées de la classe `WaitHandle` présentent un argument de type `string` qui permet de nommer l'objet de synchronisation. Ces classes présentent aussi la méthode `OpenExisting(string)` qui permet d'obtenir une référence vers un objet de synchronisation existant.

Vous pouvez en fait allez plus loin puisque la classe `WaitHandle` dérive de la classe `MarshalByRefObject`. Ainsi un objet de synchronisation peut être partagé entre plusieurs processus de machines différentes qui communiquent avec la technologie .NET Remoting.

En page 211 nous expliquons comment exploiter les types `MutexSecurity`, `EventWaitHandleSecurity` et `SemaphoreSecurity` de l'espace de noms `System.Security.AccessControl` pour manipuler les droits d'accès *Windows* accordés aux objets de synchronisation.

## *Les Mutex*

En terme de fonctionnalités les mutex sont proches de l'utilisation de la classe `Monitor` à ces différences près :

- On peut utiliser un même mutex dans plusieurs processus d'une même machine ou s'exécutant sur des machines différentes.
- L'utilisation de `Monitor` ne permet pas de se mettre en attente sur plusieurs objets.
- Les mutex n'ont pas la fonctionnalité des méthodes `Wait()`, `Pulse()` et `PulseAll()` de la classe `Monitor`.

Sachez que lorsque l'utilisation d'un mutex se cantonne à un processus, il vaut mieux synchroniser vos accès avec la classe `Monitor` qui est plus performante.

Le programme suivant montre l'utilisation d'un *mutex nommé* pour protéger l'accès à une ressource partagée par plusieurs processus sur la même machine. La ressource partagée est un fichier dans lequel chaque instance du programme écrit 10 lignes.

*Exemple 5-13 :*

```
using System.Threading ;
using System.IO ;
class Program {
   static void Main() {
      // Le mutex est nommé 'MutexTest'.
      Mutex mutexFile = new Mutex(false, "MutexTest");
      for (int i = 0 ; i < 10 ; i++){
         mutexFile.WaitOne();
         // Ouvre le fichier, écrit Hello i, ferme le fichier.
         FileInfo fi = new FileInfo("tmp.txt") ;
         StreamWriter sw = fi.AppendText() ;
         sw.WriteLine("Hello {0}", i) ;
         sw.Flush() ;
         sw.Close() ;
         // Attend 1 seconde pour rendre évidente l'action du mutex.
         System.Console.WriteLine("Hello {0}", i) ;
         Thread.Sleep(1000) ;
         mutexFile.ReleaseMutex();
      }
      mutexFile.Close();
   }
}
```

Remarquez l'utilisation de la méthode `WaitOne()` qui bloque le thread courant jusqu'à l'obtention du mutex et l'utilisation de la méthode `ReleaseMutex()` qui libère le mutex.

Dans ce programme, `new Mutex` ne signifie pas forcément la création du mutex mais la création d'une référence vers le mutex nommé `"MutexTest"`. Le mutex est effectivement créé par le système d'exploitation seulement s'il n'existe pas déjà. De même la méthode `Close()` ne détruit pas forcément le mutex si ce dernier est encore référencé par d'autres processus.

Pour ceux qui avaient l'habitude d'utiliser la technique du mutex nommé en win32 pour éviter de lancer deux instances du même programme sur la même machine, sachez qu'il existe une façon plus élégante de procéder sous .NET, décrite dans la section page 135.

## *Les événements*

Contrairement aux mécanismes de synchronisation vus jusqu'ici, les événements ne définissent pas explicitement de notion d'appartenance d'une ressource à un thread à un instant donné. Les événements servent à passer une notification d'un thread à un autre, cette notification étant « un événement s'est passé ». L'événement concerné est associé à une instance d'une des deux classes d'événement, `System.Threading.AutoResetEvent` et `System.Threading.ManualResetEvent`. Ces deux classes dérivent de la classe `System.Threading.EventWaitHandle`. Une instance de `EventWaitHandle` peut être initialisée avec une des deux valeurs `AutoReset` ou `ManualReset` de l'énumération `System.Threading.EventResetMode` ce qui fait que vous pouvez éviter de vous servir de ses deux classes filles.

Concrètement, un thread attend qu'un événement soit signalé en appelant la méthode bloquante `WaitOne()` sur l'objet événement associé. Puis un autre thread signale l'événement en

appelant la méthode `Set()` sur l'objet événement associé et permet ainsi au premier thread de reprendre son exécution.

La différence entre les *événement à repositionnement automatique* et les *événement à repositionnement manuel* est que l'on a besoin d'appeler la méthode `Reset()` pour repositionner l'événement en position non active, sur un événement de type « à repositionnement manuel » après l'avoir signalé.

> La différence entre le repositionnement manuel et automatique est plus importante que l'on pourrait croire. Si plusieurs threads attendent sur un même événement à repositionnement automatique, il faut signaler l'événement une fois pour chaque thread. Dans le cas d'un événement à repositionnement manuel il suffit de signaler une fois l'événement pour débloquer tous les threads.

Le programme suivant crée deux threads, `t0` et `t1`, qui incrémentent chacun leur propre compteur à des vitesses différentes. `t0` signale l'événement `events[0]` lorsqu'il est arrivé à 2 et `t1` signale l'événement `events[1]` lorsqu'il est arrivé à 3. Le thread principal attend que les deux événements soient signalés pour afficher un message.

*Exemple 5-14 :*

```
using System ;
using System.Threading ;
class Program {
   static EventWaitHandle[] events ;
   static void Main() {
      events = new EventWaitHandle[2] ;
      // Position initiale de l'événement : false.
      events[0] = new EventWaitHandle(false,EventResetMode.AutoReset);
      events[1] = new EventWaitHandle(false,EventResetMode.AutoReset);
      Thread t0 = new Thread(ThreadProc0) ;
      Thread t1 = new Thread(ThreadProc1) ;
      t0.Start() ; t1.Start() ;
      AutoResetEvent.WaitAll(events);
      Console.WriteLine("MainThread: Thread0 est arrivé à 2"   +
                        " et Thread1 est arrivé à 3."  ) ;
      t0.Join();t1.Join() ;
   }
   static void ThreadProc0() {
      for (int i = 0 ; i < 5 ; i++){
         Console.WriteLine("Thread0: {0}", i) ;
         if (i == 2) events[0].Set();
         Thread.Sleep(100) ;
      }
   }
   static void ThreadProc1() {
      for (int i = 0 ; i < 5 ; i++){
         Console.WriteLine("Thread1: {0}", i) ;
         if (i == 3) events[1].Set();
```

```
            Thread.Sleep(60) ;
        }
    }
}
```

Ce programme affiche :

```
Thread0: 0
Thread1: 0
Thread1: 1
Thread0: 1
Thread1: 2
Thread1: 3
Thread0: 2
MainThread: Thread0 est arrivé à 2 et Thread1 est arrivé à 3
Thread1: 4
Thread0: 3
Thread0: 4
```

## *Les sémaphores*

Une instance de la classe `System.Threading.Semaphore` permet de contraindre le nombre d'accès simultanés à une ressource. Vous pouvez vous l'imaginez comme la barrière d'entrée d'un parking automobile qui contient un nombre fixé de places. Elle ne s'ouvre plus lorsque le parking est complet. De même, une tentative d'entrée dans un sémaphore au moyen de la méthode `WaitOne()` devient bloquante lorsque le nombre d'entrées courantes a atteint le nombre d'entrée maximal. Ce nombre d'entrée maximal est fixé par le deuxième argument des constructeurs de la classe `Semaphore`. Le premier argument définit le nombre d'entrées libres initiales. Si le premier argument a une valeur inférieure à celle du deuxième, le thread qui appelle le constructeur détient automatiquement un nombre d'entrées égal à la différence des deux valeurs. Cette dernière remarque montre qu'un même thread peut détenir simultanément plusieurs entrées d'un même sémaphore.

L'exemple suivant illustre tout ceci en lançant 3 threads qui tentent régulièrement d'entrer dans un sémaphore dont le nombre d'entrées simultanées maximal est fixé à cinq. Le thread principal détient durant toute la durée de l'exécution trois de ces entrées, obligeant les 3 threads fils à se partager 2 entrées.

*Exemple 5-15 :*

```
using System ;
using System.Threading ;
class Program {
   static Semaphore semaphore;
   static void Main() {
      // Nombre d'entrées libres initiales    : 2.
      // Nombre maximal d'entrées simultanées : 5.
      // Nombre d'entrées détenues par le thread principal : 3 (5-2).
      semaphore = new Semaphore(2, 5);
      for (int i = 0 ; i < 3 ; ++i){
         Thread t = new Thread(WorkerProc) ;
```

```
            t.Name = "Thread" + i ;
            t.Start() ;
            Thread.Sleep(30) ;
         }
      }
      static void WorkerProc() {
         for (int j = 0 ; j < 3 ; j++){
            semaphore.WaitOne();
            Console.WriteLine(Thread.CurrentThread.Name + ": Begin") ;
            Thread.Sleep(200) ;  // Simule un travail de 200 ms.
            Console.WriteLine(Thread.CurrentThread.Name + ": End") ;
            semaphore.Release();
         }
      }
}
```

Voici l'affichage de ce programme. On voit bien qu'il n'y a jamais plus de deux threads fils qui « travaillent » simultanément :

```
Thread0: Begin
Thread1: Begin
Thread0: End
Thread2: Begin
Thread1: End
Thread1: Begin
Thread2: End
Thread2: Begin
Thread1: End
Thread1: Begin
Thread2: End
Thread2: Begin
Thread1: End
Thread0: Begin
Thread2: End
Thread0: End
Thread0: Begin
Thread0: End
```

## *Synchronisation avec la classe System.Threading.ReaderWriterLock*

La classe `System.Threading.ReaderWriterLock` implémente le mécanisme de synchronisation « accès lecture multiple/accès écriture unique ». Contrairement au modèle de synchronisation « accès exclusif » offert par la classe `Monitor` ou `Mutex`, ce mécanisme tient compte du fait qu'un thread demande les droits d'accès en lecture ou en écriture. Un accès en lecture peut être obtenu lorsqu'il n'y a pas d'accès en écriture courant. Un accès en écriture peut être obtenu lorsqu'il n'y a aucun accès courant, ni en lecture ni en écriture. De plus lorsqu'un accès en écriture a

été demandé mais pas encore obtenu, les nouvelles demandes d'accès en lecture sont mises en attente.

Lorsque ce modèle de synchronisation peut être appliqué, il faut toujours le privilégier par rapport au modèle proposé par les classes `Monitor` ou `Mutex`. En effet, le modèle « accès exclusif » ne permet en aucun cas des accès simultanés et est donc moins performant. De plus, empiriquement, on se rend compte que la plupart des applications accèdent beaucoup plus souvent aux données en lecture qu'en écriture.

À l'instar des mutex et des événements et au contraire de la classe `Monitor`, la classe `ReaderWriterLock` doit être instanciée pour être utilisée. Il faut donc ici aussi raisonner en terme d'objets de synchronisation et non d'objets synchronisés.

Voici un exemple de code qui montre l'utilisation de cette classe, mais qui n'en exploite pas toutes les possibilités. En effet les méthodes `DowngradeFromWriterLock()` et `UpgradeToWriterLock()` permettent de demander un changement de droit d'accès sans avoir à libérer son droit d'accès courant.

*Exemple 5-16 :*

```
using System ;
using System.Threading ;
class Program {
   static int laRessource = 0 ;
   static ReaderWriterLock rwl = new ReaderWriterLock();
   static void Main() {
      Thread tr0 = new Thread(ThreadReader) ;
      Thread tr1 = new Thread(ThreadReader) ;
      Thread tw = new Thread(ThreadWriter) ;
      tr0.Start() ; tr1.Start() ; tw.Start() ;
      tr0.Join() ;  tr1.Join() ;  tw.Join() ;
   }
   static void ThreadReader() {
      for (int i = 0 ; i < 3 ; i++)
         try{
            // AcquireReaderLock() lance une
            // ApplicationException si timeout.
            rwl.AcquireReaderLock(1000);
            Console.WriteLine(
               "Début Lecture  laRessource = {0}", laRessource) ;
            Thread.Sleep(10) ;
            Console.WriteLine(
               "Fin   Lecture  laRessource = {0}", laRessource) ;
            rwl.ReleaseReaderLock() ;
         }
         catch ( ApplicationException ) { /* à traiter */ }
   }
   static void ThreadWriter() {
      for (int i = 0 ; i < 3 ; i++)
```

```
         try{
            // AcquireWriterLock() lance une
            // ApplicationException si timeout.
            rwl.AcquireWriterLock(1000);
            Console.WriteLine(
               "Début Ecriture laRessource = {0}", laRessource) ;
            Thread.Sleep(100) ;
            laRessource++ ;
            Console.WriteLine(
               "Fin   Ecriture laRessource = {0}", laRessource) ;
            rwl.ReleaseWriterLock() ;
         }
         catch ( ApplicationException ) { /* à traiter */ }
   }
}
```

Ce programme affiche :

```
Début lecture  laRessource = 0
Début lecture  laRessource = 0
Fin   lecture  laRessource = 0
Fin   lecture  laRessource = 0
Début écriture laRessource = 0
Fin   écriture laRessource = 1
Début lecture  laRessource = 1
Début lecture  laRessource = 1
Fin   lecture  laRessource = 1
Fin   lecture  laRessource = 1
Début écriture laRessource = 1
Fin   écriture laRessource = 2
Début lecture  laRessource = 2
Début lecture  laRessource = 2
Fin   lecture  laRessource = 2
Fin   lecture  laRessource = 2
Début écriture laRessource = 2
Fin   écriture laRessource = 3
```

## *Synchronisation avec l'attribut System.Runtime.Remoting.Contexts.SynchronizationAttribute*

Lorsque l'attribut `System.Runtime.Remoting.Contexts.Synchronization` est appliqué à une classe, une instance de cette classe ne peut pas être accédée par plus d'un thread à la fois. On dit alors que la classe est *thread-safe*.

Pour que ce comportement s'applique correctement il faut que la classe sur laquelle s'applique l'attribut soit *context-bound*, c'est-à-dire qu'elle doit dériver de la classe `System.ContextBoundObject`. La signification du terme context-bound est expliquée page 842.

Voici un exemple illustrant comment appliquer ce comportement :

*Exemple 5-17 :*

```
using System.Runtime.Remoting.Contexts ;
using System.Threading ;
[Synchronization(SynchronizationAttribute.REQUIRED)]
public class Foo : System.ContextBoundObject {
   public void AfficheThreadId() {
      System.Console.WriteLine("Début : ManagedThreadId = " +
                        Thread.CurrentThread.ManagedThreadId ) ;
      Thread.Sleep(1000) ;
      System.Console.WriteLine("Fin :   ManagedThreadId = " +
                        Thread.CurrentThread.ManagedThreadId ) ;
   }
}
public class Program {
   static Foo m_Objet = new Foo() ;
   static void Main() {
      Thread t0 = new Thread( ThreadProc ) ;
      Thread t1 = new Thread( ThreadProc ) ;
      t0.Start() ; t1.Start() ;
      t0.Join() ;  t1.Join() ;
   }
   static void ThreadProc() {
      for (int i = 0 ; i < 2 ; i++)
         m_Objet.AfficheThreadId() ;
   }
}
```

Cet exemple affiche :

```
Début : ManagedThreadId = 27
Fin :   ManagedThreadId = 27
Début : ManagedThreadId = 28
Fin :   ManagedThreadId = 28
Début : ManagedThreadId = 27
Fin :   ManagedThreadId = 27
Début : ManagedThreadId = 28
Fin :   ManagedThreadId = 28
```

## *La notion de domaine de synchronisation*

Pour une bonne compréhension de ce qui suit, il faut avoir une connaissance des notions suivantes :

- domaine d'application (décrit page 87),
- contexte .NET et classe context-bound/context-agile (décrit page 842),
- intercepteur de messages (décrit page 822).

Un domaine de synchronisation est une entité entièrement prise en charge par le CLR. Un tel domaine contient un ou plusieurs contextes .NET et donc, contient les objets de ces contextes. Un contexte .NET ne peut appartenir qu'à au plus un seul domaine de synchronisation. En outre, la notion de domaine de synchronisation est plus fine que la notion de domaine d'application. La figure suivante illustre les relations d'inclusions entre processus, domaines d'application, domaines de synchronisation, contextes .NET et objets .NET :

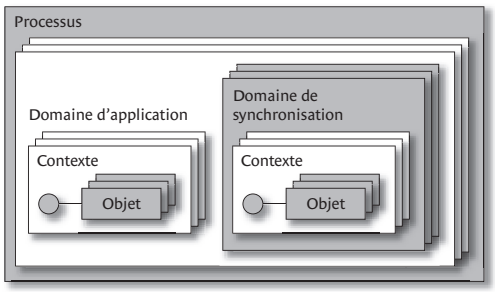


*Figure 5-2 : Processus, domaine d'application, domaine de synchronisation, contexte.NET et objet .NET*

Puisque nous parlons de la synchronisation, et donc des applications *multi-threaded*, il est utile de rappeler que les domaines d'applications et les threads d'un processus sont deux notions orthogonales. En effet, un thread peut traverser librement la frontière entre deux domaines d'application, par exemple en appelant un objet `A` situé dans le domaine d'application `DA`, à partir d'un objet `B` situé dans le domaine d'application `DB`. De plus, le code d'un domaine d'application peut être exécuté simultanément par zéro, un ou plusieurs threads.

En revanche, tout l'intérêt d'un domaine de synchronisation réside dans le fait qu'il ne peut être partagé simultanément par plusieurs threads. Autrement dit, les méthodes des objets contenus dans un domaine de synchronisation ne peuvent être simultanément exécutées par plusieurs threads. Ceci implique qu'à un instant donné, au plus un thread se trouve dans domaine de synchronisation. On parle alors de *droits d'accès exclusifs* au domaine de synchronisation. Encore une fois, la gestion de ces droits d'accès exclusifs est entièrement assurée par le CLR.

## *L'attribut System.Runtime.Remoting.Contexts.Synchronization et les domaines de synchronisation*

Vous avez peut-être déjà deviné que l'attribut `System.Runtime.Remoting.Contexts.Synchronization` sert en fait à indiquer au CLR quand créer un domaine de synchronisation et comment en délimiter sa frontière. Ces réglages se font en utilisant dans la déclaration d'un attribut `System.Runtime.Remoting.Contexts.Synchronization` une des quatre valeurs suivantes `NOTSUPPORTED`, `SUPPORTED`, `REQUIRED` ou `REQUIRES_NEW`. Notez que la valeur `REQUIRED` est choisie par défaut.

L'appartenance à un domaine d'application se communique de proche en proche lorsqu'un objet en crée un autre. On parle ainsi d'*objet créateur*, mais prenez en compte qu'un objet peut être aussi créé au sein d'une méthode statique. Or, il se peut que l'exécution de la méthode statique soit le fruit d'un appel d'un objet situé dans un domaine de synchronisation. Il faut alors savoir

que dans ce cas, la méthode statique propage l'appartenance au domaine d'application et joue ainsi le rôle d'un objet créateur. Voici l'explication des quatre comportements possibles :

| | |
|---|---|
| NOT_SUPPORTED | Ce paramètre assure qu'une instance de la classe sur laquelle s'applique l'attribut `Synchronization` **n'appartiendra jamais** à un domaine de synchronisation (que son objet créateur appartienne à un domaine de synchronisation ou non). |
| SUPPORTED | Ce paramètre indique qu'une instance de la classe sur laquelle s'applique l'attribut `Synchronization` **n'a pas besoin d'appartenir** à un domaine de synchronisation. Néanmoins, une telle instance appartiendra au domaine de synchronisation de son objet créateur si ce dernier en a un. Ce comportement est peu utile, car le développeur doit quand même prévoir un autre mécanisme de synchronisation au sein des méthodes de la classe. Cependant, il peut éventuellement permettre de propager l'appartenance à un domaine de synchronisation à partir d'un objet qui n'a pas besoin d'être synchroniser (ce type d'objet est rare, puisqu'il ne doit pas avoir d'état). |
| REQUIRED | Ce paramètre assure qu'une instance de la classe sur laquelle s'applique l'attribut `Synchronization` **sera de toutes façons dans** un domaine de synchronisation. Si l'objet créateur appartient déjà à un domaine de synchronisation, alors on se satisfera de celui là. Sinon, un nouveau domaine de synchronisation est créé pour accueillir ce nouvel objet. |
| REQUIRES_NEW | Ce paramètre assure qu'une instance de la classe sur laquelle s'applique l'attribut `Synchronization` **sera dans un nouveau** domaine de synchronisation (que son objet créateur appartienne à un domaine de synchronisation ou non). |

Ces comportements peuvent se résumer ainsi :

| **La valeur appliquée:** | **Est-ce que l'objet créateur est dans un domaine de synchronisation ?** | **L'objet créé résidera...** |
|---|---|---|
| NOT_SUPPORTED | Non | ...à l'extérieur de tout domaine de synchronisation. |
| | Oui | |
| SUPPORTED | Non | ...à l'extérieur de tout domaine de synchronisation. |
| | Oui | ...dans le domaine de synchronisation de l'objet créateur. |

| | | |
|---|---|---|
| REQUIRED | Non | ...dans un nouveau domaine de synchronisation. |
| | Oui | ...dans le domaine de synchronisation de l'objet créateur. |
| REQUIRES_NEW | Non | ...dans un nouveau domaine de synchronisation. |
| | Oui | |

## *La notion de réentrance dans les domaines de synchronisation*

Dans un domaine de synchronisation `D`, lorsque le thread `T1` qui a les droits d'accès exclusifs effectue un appel sur un objet situé hors de `D`, deux comportements peuvent alors être appliqués par le CLR :

- Soit le CLR autorise un autre thread `T2` à acquérir les droits d'accès exclusifs de `D`. Dans ce cas, il se peut que T1 doivent attendre au retour de son appel à l'extérieur de `D` que `T2` libère les droits d'accès exclusifs de `D`. On dit qu'il y a *réentrance* puisque `T1` demande à réentrer dans `D`.
- Soit les droits d'accès exclusifs de `D` restent alloués à `T1`. Dans ce cas, un autre thread `T2` souhaitant invoquer une méthode d'un objet de `D` devra attendre que `T1` libère les droits d'accès exclusifs de `D`.

Un appel à une méthode statique n'est pas considéré comme un appel à l'extérieur d'un domaine de synchronisation.

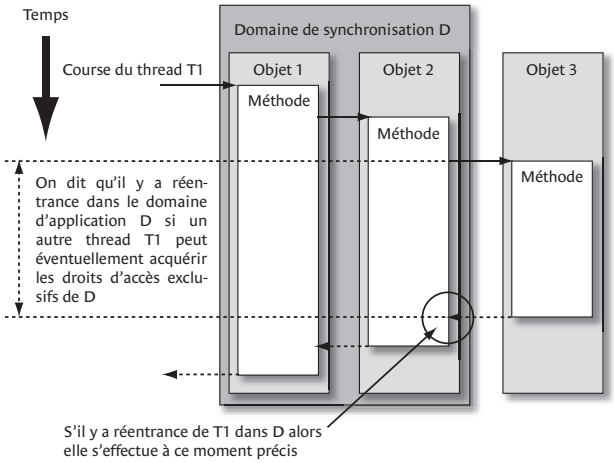


*Figure 5-3 : La réentrance dans un domaine de synchronisation*

Certains constructeurs de la classe `System.Runtime.Remoting.Contexts.Synchronization` acceptent un booléen en paramètre. Ce booléen définit s'il y a réentrance lors d'un appel hors du domaine de synchronisation courant. Lorsque ce booléen est positionné à `true`, il y a réentrance.

Notez qu'il suffit que l'attribut de synchronisation de la classe du premier objet rencontré par un thread dans un domaine de synchronisation ait positionné la réentrance à `true` pour qu'il y ait effectivement réentrance.

L'exemple suivant est l'illustration par le code de la figure ci-dessus :

*Exemple 5-18 :*

```
using System ;
using System.Runtime.Remoting.Contexts ;
using System.Threading ;
[Synchronization(SynchronizationAttribute.REQUIRES_NEW, true)]
public class Foo1 : ContextBoundObject {
   public void AfficheThreadId() {
      Console.WriteLine("Foo1 Début : ManagedThreadId = " +
                        Thread.CurrentThread.ManagedThreadId) ;
      Thread.Sleep(1000) ;
      Foo2 obj2 = new Foo2() ;
      obj2.AfficheThreadId() ;
      Console.WriteLine("Foo1 Fin :   ManagedThreadId = " +
                        Thread.CurrentThread.ManagedThreadId) ;
   }
}
[Synchronization(SynchronizationAttribute.REQUIRED)]
public class Foo2 : ContextBoundObject {
   public void AfficheThreadId() {
      Console.WriteLine("Foo2 Début : ManagedThreadId = " +
                        Thread.CurrentThread.ManagedThreadId) ;
      Thread.Sleep(1000) ;
      Foo3 obj3 = new Foo3() ;
      obj3.AfficheThreadId() ;
      Console.WriteLine("Foo2 Fin :   ManagedThreadId = " +
                        Thread.CurrentThread.ManagedThreadId) ;
   }
}
// On est certains que les instances de cette classe ne
// seront dans aucun domaine de synchronisation.
[Synchronization(SynchronizationAttribute.NOT_SUPPORTED)]
public class Foo3 : ContextBoundObject {
   public void AfficheThreadId() {
      Console.WriteLine("Foo3 Début : ManagedThreadId = " +
                         Thread.CurrentThread.ManagedThreadId) ;
      Thread.Sleep(1000) ;
      Console.WriteLine("Foo3 Fin :   ManagedThreadId = " +
                         Thread.CurrentThread.ManagedThreadId) ;
   }
}
public class Program {
   static Foo1 m_Objet = new Foo1() ;
   static void Main() {
```

```
      Thread t1 = new Thread( ThreadProc ) ;
      Thread t2 = new Thread( ThreadProc ) ;
      t1.Start() ; t2.Start() ;
      t1.Join() ; t2.Join() ;
   }
   static void ThreadProc() {
      m_Objet.AfficheThreadId() ;
   }
}
```

Ce programme affiche ceci :

```
Foo1 Début : ManagedThreadId = 3
Foo2 Début : ManagedThreadId = 3
Foo1 Début : ManagedThreadId = 4
Foo2 Début : ManagedThreadId = 4
Foo3 Début : ManagedThreadId = 4
Foo3 Début : ManagedThreadId = 3
Foo3 Fin :   ManagedThreadId = 4
Foo2 Fin :   ManagedThreadId = 4
Foo1 Fin :   ManagedThreadId = 4
Foo3 Fin :   ManagedThreadId = 3
Foo2 Fin :   ManagedThreadId = 3
Foo1 Fin :   ManagedThreadId = 3
```

Il est clair que si l'on avait désactivé la réentrance dans `Foo1`, l'affichage de ce programme aurait été le suivant :

```
Foo1 Début : ManagedThreadId = 3
Foo2 Début : ManagedThreadId = 3
Foo3 Début : ManagedThreadId = 3
Foo3 Fin :   ManagedThreadId = 3
Foo2 Fin :   ManagedThreadId = 3
Foo1 Fin :   ManagedThreadId = 3
Foo1 Début : ManagedThreadId = 4
Foo2 Début : ManagedThreadId = 4
Foo3 Début : ManagedThreadId = 4
Foo3 Fin :   ManagedThreadId = 4
Foo2 Fin :   ManagedThreadId = 4
Foo1 Fin :   ManagedThreadId = 4
```

La réentrance est utilisée pour optimiser la gestion des ressources car elle permet de réduire globalement les durées d'accès exclusifs des threads sur les domaines de synchronisation. Cependant par défaut, la réentrance est désactivée. En effet, lorsqu'il y a réentrance, notre compréhension de la notion de domaine de synchronisation est fortement perturbée. Bien pire encore, activer à tort et à travers la réentrance peut rapidement amener à des situations de *deadlock*. Pour ces raisons, il est fortement déconseillé d'activer la réentrance.

## *Un autre attribut nommé Synchronization*

Le *framework* .NET présente un autre attribut ayant ce nom mais faisant partie d'une autre espace de noms. Cet attribut `System.EntrepriseServices.Synchronization` a la même fina-

lité, mais il utilise le service d'entreprise COM+ de synchronisation. L'utilisation de l'attribut `System.Runtime.Remoting.Contexts.Synchronization` est préférable pour deux raisons :

- Son utilisation est plus performante.
- Ce mécanisme supporte les appels asynchrones, contrairement à la version COM+.

La notion de service d'entreprise COM+ est présentée page 295.

# *Le pool de threads du CLR*

## *Introduction*

Le concept de *pool de threads* n'est pas nouveau. Cependant le *framework* .NET vous permet d'utiliser un pool de threads beaucoup plus simplement que n'importe quelle autre technologie, grâce à la classe `System.Threading.ThreadPool`.

Dans une application multithreads, la plupart des threads passent leur temps à attendre des événements. Concrètement vos threads sont globalement sous exploités. De plus, le fait de devoir tenir compte de la gestion des threads lors du design de votre application est une difficulté dont on se passerait volontiers.

L'utilisation d'un pool de threads résout de façon élégante et performante ces problèmes. Vous postez des tâches au pool qui se charge de les distribuer à ses threads. Le pool est entièrement responsable de :

- la création et de la destruction de ses threads ;
- la distribution des tâches ;
- l'utilisation optimale de ses threads.

Le développeur est donc déchargé de ces responsabilités. Malgré tous ces avantages, il est souhaitable de ne pas utiliser de pool de threads lorsque :

- Vos tâches doivent être gérées avec un système de priorité.
- Vos tâches sont longues à s'exécuter (plusieurs secondes).
- Vos tâches doivent être traitées dans des *STA* (*Single Apartment Thread*). En effet les threads d'un pool sont de type *MTA* (*Multiple Apartement Thread*). Cette notion de *thread apartment*, inhérente à la technologie COM, est expliquée page 285.

## *Utilisation d'un pool de threads*

En .NET il n'y a qu'un pool de threads par processus. Ainsi toutes les méthodes de la classe `ThreadPool` sont statiques puisqu'elles s'appliquent à l'unique pool. Le *framework* .NET utilise ce pool pour les appels asynchrones, décrits un peu plus loin dans ce chapitre, les mécanismes d'*entré/sortie* asynchrones ou les timers décrits dans la prochaine section.

Le nombre maximal de threads du pool est de 25 threads par processeur pour traiter les opérations asynchrones (*completion port thread* ou *thread I/O* en anglais) et de 25 *threads ouvrier* (*worker thread* en anglais) par processeur. Ces deux limites par défaut sont modifiables à tout moment en appelant la méthode statique `ThreadPool.SetMaxThreads()`. Si le nombre maximal de threads

est atteint dans le pool, ce dernier ne crée plus de nouveaux threads et les tâches dans la file du pool ne seront traitées que lorsqu'un thread du pool se libèrera. En revanche, le thread responsable de la création de la tâche, n'a pas à attendre qu'elle soit traitée. Vous pouvez utiliser le pool de threads de deux façons différentes :

- En postant vos propres tâches et leurs méthodes de traitement avec la méthode statique `ThreadPool.QueueUserWorkItem()`. Une fois qu'une tâche a été postée au pool, elle ne peut plus être annulée.
- En créant un timer qui poste périodiquement une tâche prédéfinie et sa méthode de traitement au pool. Pour cela, il faut utiliser la méthode statique `ThreadPool.RegisterWaitForSingleObject()`.

Chacune de ces deux méthodes existe dans une version non protégée (`UnsafeQueueUserWorkItem()` et `UnsafeRegisterWaitForSingleObject()`) . Ces versions non protégées permettent aux threads ouvriers du pool de ne pas être dans le même *contexte* de sécurité que le thread qui a déposé la tâche. L'utilisation de ces méthodes améliore donc les performances puisque la pile du thread qui a déposé la tâche n'est pas vérifiée lors de la gestion des contextes de sécurité. Les méthodes de traitement sont référencées par des *délégués*.

Voici un exemple qui montre l'utilisation de tâches utilisateurs (paramétrées par un numéro et traitées par la méthode `ThreadTache()`) et de tâches postées périodiquement (sans paramètre et traitées par la méthode `ThreadTacheWait()`). Les tâches utilisateurs sont volontairement longues pour forcer le pool à créer de nouveaux threads.

*Exemple 5-19 :*

```
using System.Threading ;
public class Program {
   public static void Main() {
      // Positionnement initial de l'événement : false.
      ThreadPool.RegisterWaitForSingleObject(
         new AutoResetEvent(false),
         // Méthode de traitement de la tâche périodique.
         new WaitOrTimerCallback( ThreadTacheWait ),
         null,   // La tâche périodique n'a pas de paramètres.
         2000,   // La période est de 2 secondes.
         false) ; // La tâche périodique est indéfiniment déclenchée.

      // Poste 3 tâches utilisateur de paramètres 0,1,2.
      for (int count = 0 ; count < 3 ; ++count)
         ThreadPool.QueueUserWorkItem(
            new WaitCallback(ThreadTache), count) ;

      // Attente de 12 secondes avant de finir le processus
      // car les threads du pool sont des threads background.
      Thread.Sleep(12000) ;
   }
   static void ThreadTache(object obj) {
      System.Console.WriteLine("Thread#{0}  Tâche#{1}  Début",
            Thread.CurrentThread.ManagedThreadId , obj.ToString()) ;
```

```
      Thread.Sleep(5000) ;
      System.Console.WriteLine("Thread#{0}  Tâche#{1}  Fin",
            Thread.CurrentThread.ManagedThreadId , obj.ToString()) ;
   }
   static void ThreadTacheWait(object obj, bool signaled) {
      System.Console.WriteLine("Thread#{0}  TâcheWait",
            Thread.CurrentThread.ManagedThreadId ) ;
   }
}
```

Ce programme affiche ceci (d'une manière non déterministe):

```
Thread#4  Tâche#0  Début
Thread#5  Tâche#1  Début
Thread#6  Tâche#2  Début
Thread#7  TâcheWait
Thread#7  TâcheWait
Thread#4  Tâche#0  Fin
Thread#5  Tâche#1  Fin
Thread#6  Tâche#2  Fin
Thread#7  TâcheWait
Thread#7  TâcheWait
```

# *Timers*

La plupart des applications contiennent des tâches qui doivent être exécutées périodiquement. Par exemple vous pouvez imaginez de vérifier périodiquement la disponibilité d'un serveur. La première solution qui vient à l'esprit pour implémenter ce paradigme est de créer un thread dédié à une tâche qui appelle la méthode `Thread.Sleep()` entre deux exécutions. Cette implémentation présente l'inconvénient de consommer inutilement un thread entre deux exécutions. Nous allons présenter plusieurs implémentations plus performantes fournies par le *framework* .NET.

Plus précisément, le *framework* présente les trois classes `System.Timers.Timer`, `System.Threading.Timer` et `System.Windows.Forms.Timer`. La classe de l'espace de noms `Forms` doit être utilisée pour exécuter des tâches périodiques graphiques, telles que le rafraîchissement de données sur l'écran ou l'affichage consécutif des images d'une animation. Le choix entre les classes des espaces de noms `Timers` et `Threading` est plus délicat et dépend de vos besoins.

## *La classe System.Timers.Timer*

L'implémentation `System.Timers.Timer` utilise les threads du pool de threads pour exécuter une tâche. Aussi, vous devez synchroniser les accès aux ressources consommées par une telle tâche.

Cette classe présente la propriété `double Interval{get;set;}` qui permet d'accéder à la période exprimée en millisecondes. Les méthodes `Start()` et `Stop()` vous permettent d'activer ou de désactiver le timer. Vous pouvez aussi changer l'activation du timer en utilisant la propriété `bool Enabled{get;set;}`.

Un délégué de type `ElapsedEventHandler` référence la méthode qui représente la tâche à exécuter. Cette référence est représentée par l'événement `ElapsedEventHandler Elapsed`. La signature proposée par la délégation `ElapsedEventHandler` est celle-ci :

```
void ElapsedEventHandler(object sender, ElapsedEventArgs e)
```

Le premier argument référence le timer qui a déclenché la tâche. Ainsi, plusieurs timers peuvent déclencher une même méthode et vous pouvez les différencier au moyen de cet argument. De plus, puisqu'un délégué peut référencer plusieurs méthodes, un même timer peut appeler consécutivement plusieurs méthodes à chaque déclenchement. Le second argument contient la date de déclenchement du timer que vous pouvez récupérer avec la propriété `DateTime ElapsedEvenArgs.SignalTime{get;}`. Le programme suivant illustre l'utilisation de la classe `System.Timers.Timer` :

*Exemple 5-20 :*

```
using System.Timers ;
class Program {
   static Timer Timer1 = new Timer() ;
   static Timer Timer2 = new Timer() ;
   static void Main() {
      Timer1.Interval = 1000 ;  // Période = 1 seconde.
      Timer1.Elapsed +=
            new ElapsedEventHandler(PeriodicTaskHandler) ;
      Timer2.Interval = 2000 ;  // Période = 2 secondes.
      Timer2.Elapsed +=
            new ElapsedEventHandler(PeriodicTaskHandler) ;
      Timer2.Elapsed +=
            new ElapsedEventHandler(PeriodicTaskHandler) ;
      Timer1.Start() ; Timer2.Start() ;
      System.Threading.Thread.Sleep(5000) ; // Dors 5 secondes.
      Timer1.Stop() ; Timer2.Stop() ;
   }
   static void PeriodicTaskHandler(object sender,
                                   ElapsedEventArgs e) {
      string str = (sender == Timer1) ? "Timer1 " : "Timer2 " ;
      str += e.SignalTime.ToLongTimeString() ;
      System.Console.WriteLine(str) ;
   }
}
```

Ce programme affiche ceci :

```
Timer1 19:42:49
Timer1 19:42:50
Timer2 19:42:50
Timer2 19:42:50
Timer1 19:42:51
Timer1 19:42:52
Timer2 19:42:52
Timer2 19:42:52
Timer1 19:42:53
```

Enfin, sachez que la propriété `ISynchronizeInvoke Timer.SynchronizingObject` vous permet de préciser le thread qui doit exécuter la tâche. L'interface `ISynchronizeInvoke` est présentée un peu plus loin dans ce chapitre.

### *La classe System.Threading.Timer*

La classe `System.Threading.Timer` est assez similaire à la classe `System.Timers.Timer`. Cette classe utilise aussi les threads du pool de threads pour exécuter une tâche mais à la différence de la classe `System.Timers.Timer`, elle ne vous permet de préciser exactement un thread.

Une autre différence est que cette classe vous permet de fournir une *échéance de démarrage* (*due time* en anglais). L'échéance de démarrage définie l'instant où le timer va démarrer en précisant une durée. Vous pouvez modifier l'échéance de démarrage à n'importe quel moment en appelant la méthode `Change()`. En précisant une échéance de démarrage nulle vous pouvez démarrer le timer immédiatement. En précisant la constante `System.Threading.Timer.Infinite` vous pouvez stopper le timer. Vous pouvez aussi stopper le timer en appelant la méthode `Dispose()` mais alors vous ne pourrez plus le redémarrer.

### *La classe System.Windows.Forms.Timer class*

La philosophie d'utilisation de la classe `System.Windows.Forms.Timer` est plus proche de celle de la classe `System.Timers.Timer` que celle de la classe `System.Threading.Timer`. La particularité de cette classe est qu'elle démarre ses tâches toujours avec le thread dédié à la fenêtre concernée. Ceci vient du fait que cette implémentation utilise en interne le message *Windows* `WM_TIMER`. Vous ne devez pas utiliser cette classe pour démarrer des tâches dont l'exécution prend plus qu'une fraction de seconde sous peine de geler votre interface graphique. Un exemple d'utilisation de ce type de timer peut être trouvé dans le chapitre consacré à *Windows Forms* en page 702.

## *Appel asynchrone d'une méthode*

> Pour aborder cette section, il est nécessaire d'avoir compris la notion de délégué, expliquée en page 379.

On dit d'un appel de méthode qu'il est synchrone lorsque le thread du coté qui réalise l'appel, attend que la méthode soit exécutée avant de continuer. Ce comportement consomme des ressources puisque le thread est bloqué pendant ce temps. Lors d'un appel sur un objet distant, cette durée est potentiellement immense, puisque le coût d'un appel réseau représente des milliers, voire des millions, de cycles processeurs. Cependant cette attente n'est obligatoire que dans le cas où les informations retournées par l'appel de la méthode sont immédiatement consommées après l'appel. Dans la programmation en général et dans les architectures distribuées en particulier, il arrive souvent qu'un appel de méthode effectue une action et ne retourne que l'information décrivant si l'action s'est bien passée ou non. Dans ce cas, le programme n'a pas forcément besoin de savoir immédiatement si l'action s'est bien passée. On peut décider d'essayer de recommencer cette action plus tard si on apprend qu'elle a échoué.

Pour gérer ce type de situation on peut utiliser un *appel asynchrone*. L'idée est que le thread qui réalise un appel de méthode sur un objet, retourne immédiatement, sans attendre la fin

de l'appel. L'appel est automatiquement pris en charge par un thread du pool de threads du processus. Le programme peut ultérieurement récupérer les informations retournées par l'appel asynchrone. La technique d'appel asynchrone est entièrement gérée par le CLR.

Le mécanisme que nous décrivons peut être utilisé dans votre propre architecture. Il est aussi utilisé par les classes du *framework* .NET, notamment pour gérer les flots de données d'une manière asynchrone ou pour gérer des appels asynchrones sur des objets distants, c'est-à-dire situés dans un autre domaine d'application.

## *Délégation asynchrone*

Lors d'un appel asynchrone vous n'avez pas à créer ni à vous occupez du thread qui exécute le corps de la méthode. Ce thread est géré par le pool de threads décrit un peu plus haut.

Avant d'utiliser effectivement un *délégué asynchrone*, il est judicieux de remarquer que toutes les délégations présentent automatiquement les deux méthodes `BeginInvoke()` et `EndInvoke()`. La signature de ces deux méthodes est calquée sur la signature de la délégation qui les présente. Par exemple la délégation suivante...

```
delegate int Deleg(int a,int b) ;
```

...expose les deux méthodes suivantes :

```
IAsyncResult BeginInvoke(int a,int b,AsyncCallback callback,object o) ;
int EndInvoke(IAsyncResult result) ;
```

Ces deux méthodes sont produites par le compilateur de C#. Cependant, le mécanisme d'intellisense de *Visual Studio* est capable de les inférer à partir d'une délégation de façon à vous les présenter.

Pour appeler une méthode d'une manière asynchrone, il faut d'abord la référencer avec un délégué ayant la même signature. Il suffit ensuite d'appeler la méthode `BeginInvoke()` sur ce délégué. Comme vous l'avez remarqué, le compilateur a fait en sorte que les premiers arguments de `BeginInvoke()` soient les arguments de la méthode à appeler. Les deux derniers arguments de cette méthode de type `AsyncCallback` et `object` sont expliqués un peu plus loin.

La valeur de retour de l'appel asynchrone d'une méthode, peut être récupérée en appelant la méthode `EndInvoke()`. Là aussi, le compilateur a fait en sorte que le type de la valeur de retour de `EndInvoke()` soit le même que le type de la valeur de retour de la délégation (ce type est `int` dans notre exemple). L'appel à `EndInvoke()` est bloquant. C'est-à-dire que l'appel ne retourne que lorsque l'exécution asynchrone est effectivement terminée.

Le programme suivant illustre l'appel asynchrone d'une méthode `WriteSomme()`. Notez que pour bien différencier le thread exécutant la méthode `Main()` et le thread exécutant la méthode `WriteSomme()`, nous affichons la valeur de hachage du thread courant (qui est différente pour chaque thread).

*Exemple 5-21 :*

```
using System.Threading ;
class Program {
   public delegate int Deleg(int a, int b) ;
   static int WriteSomme(int a, int b) {
      int somme = a + b ;
```

```
      System.Console.WriteLine("Thread#{0} : WriteSomme() somme = {1}",
               Thread.CurrentThread.ManagedThreadId , somme) ;
      return somme ;
   }
   static void Main() {
      Deleg proc = WriteSomme ;
      System.IAsyncResult async= proc.BeginInvoke(10, 10, null, null);
      // Possibilité de faire quelquechose ici...
      int somme = proc.EndInvoke(async);
      System.Console.WriteLine("Thread#{0} : Main()       somme = {1}",
               Thread.CurrentThread.ManagedThreadId , somme) ;
   }
}
```

Ce programme affiche :

```
Thread 15 : WriteSomme() Somme = 20
Thread 18 : Main()       Somme = 20
```

Un appel asynchrone est matérialisé par un objet dont la classe implémente l'interface `System.IAsyncResult`. Dans cet exemple la classe sous-jacente est `System.Runtime.Remoting.Messaging.AsyncResult`. L'objet `AsyncResult` est retourné par la méthode `BeginInvoke()`. Il est passé en argument de la méthode `EndInvoke()` pour identifier l'appel asynchrone.

Si une exception est lancée lors d'un appel asynchrone, elle est automatiquement interceptée et stockée par le CLR. Le CLR relancera l'exception lors de l'appel à `EndInvoke()`.

## *Procédure de finalisation*

Vous avez la possibilité de spécifier une méthode qui sera automatiquement appelée lorsque l'appel asynchrone sera terminé. Cette méthode est appelée *procédure de finalisation*. Une procédure de finalisation est appelée par le même thread que celui qui a exécuté l'appel asynchrone.

Pour utiliser une procédure de finalisation, il vous suffit de spécifier la méthode dans une délégation de type `System.AsyncCallback` comme avant-dernier paramètre de la méthode `BeginInvoke()`. Cette méthode doit être conforme à cette délégation, c'est-à-dire qu'elle doit retourner le type `void` et prendre pour seul argument une interface `IAsyncResult`. Comme le montre l'exemple suivant, cette méthode doit appeler `EndInvoke()`.

Un problème se pose, car les threads du pool utilisés pour traiter les appels asynchrones sont des threads background. À l'instar de l'exemple ci-dessous, il faut que vous implémentiez un mécanisme de gestion d'événements pour vous assurer que l'application ne se termine pas sans avoir terminé l'exécution asynchrone. L'interface `IAsyncResult` présente un objet de synchronisation d'une classe dérivée de `WaitHandle`, mais cet objet est signalé dès que le traitement asynchrone est fini et avant que la procédure de finalisation soit appelée. Cet objet ne peut donc pas permettre d'attendre la fin de l'exécution de la procédure de finalisation.

*Exemple 5-22 :*

```
using System ;
using System.Threading ;
using System.Runtime.Remoting.Messaging ;
class Program {
```

```
    public delegate int Deleg(int a, int b) ;
    // Position initiale de l'événement : false.
    static AutoResetEvent ev = new AutoResetEvent(false);
    static int WriteSomme(int a, int b) {
       Console.WriteLine(
         "{0} : Somme = {1}",Thread.CurrentThread.ManagedThreadId ,a+b) ;
       return a + b ;
    }
    static void SommeFinie(IAsyncResult async) {
       // Attend une seconde pour simuler un traitement long.
       Thread.Sleep(1000) ;
       Deleg proc =
               ((AsyncResult)async).AsyncDelegate as Deleg ;
       int somme = proc.EndInvoke(async) ;
       Console.WriteLine("{0} : Procédure de finalisation somme = {1}",
                         Thread.CurrentThread.ManagedThreadId , somme) ;
       ev.Set();
    }
    static void Main() {
       Deleg proc = WriteSomme ;
       IAsyncResult async = proc.BeginInvoke(10, 10,
                                new AsyncCallback(SommeFinie), null) ;
       Console.WriteLine(
      "{0} : BeginInvoke() appelée ! Attend l'éxécution de SommeFinie()",
         Thread.CurrentThread.ManagedThreadId ) ;
       ev.WaitOne();
       Console.WriteLine(
                 "{0} : Bye... ", Thread.CurrentThread.ManagedThreadId ) ;
    }
}
```

Cet exemple affiche :

```
12 : BeginInvoke() appelée ! Attend l'éxécution de SommeFinie()
14 : Somme = 20
14 : Procédure de finalisation Somme = 20
12 : Bye...
```

Si vous enlevez le mécanisme d'événement, cet exemple affiche ceci :

```
12 : BeginInvoke() appelée ! Attend l'éxécution de SommeFinie()
12 : Bye...
```

L'application n'attend pas la fin de l'exécution du traitement asynchrone et de sa procédure de finalisation.

## *Passage d'un état à la procédure de finalisation*

Si vous ne le positionnez pas à `null`, le dernier paramètre de la méthode `BeginInvoke()` représente une référence vers un objet utilisable à la fois dans le thread qui déclenche l'appel

asynchrone et dans la procédure de finalisation. Une autre référence vers cet objet est la propriété `AsyncState` de l'interface `IAsyncResult`. Vous pouvez vous en servir pour représenter un état positionné dans la procédure de finalisation. Par exemple, l'événement de l'exemple de la section précédente peut être vu comme un état. Voici l'exemple précédent réécrit pour utiliser cette fonctionnalité :

*Exemple 5-23 :*

```
using System ;
using System.Threading ;
using System.Runtime.Remoting.Messaging ;
class Program {
   public delegate int Deleg(int a, int b) ;
   static int WriteSomme(int a, int b) {
      Console.WriteLine(
        "{0} : Somme = {1}", Thread.CurrentThread.ManagedThreadId , a+b) ;
      return a + b ;
   }
   static void SommeFinie(IAsyncResult async) {
      // Attend une seconde pour simuler un traitement long.
      Thread.Sleep(1000) ;
      Deleg proc =
              ((AsyncResult)async).AsyncDelegate as Deleg ;
      int somme = proc.EndInvoke(async) ;
      Console.WriteLine("{0} : Procédure de finalisation somme = {1}",
                        Thread.CurrentThread.ManagedThreadId , somme) ;
      ((AutoResetEvent)async.AsyncState).Set();
   }
   static void Main() {
      Deleg proc = WriteSomme ;
      AutoResetEvent ev = new AutoResetEvent(false) ;
      IAsyncResult async = proc.BeginInvoke(10, 10,
                               new AsyncCallback(SommeFinie), ev) ;
      Console.WriteLine(
   "{0} : BeginInvoke() appelée ! Attend l'éxécution de SommeFinie()",
         Thread.CurrentThread.ManagedThreadId ) ;
      ev.WaitOne();
      Console.WriteLine(
            "{0} : Bye... ", Thread.CurrentThread.ManagedThreadId ) ;
   }
}
```

## *Appels sans retour (One-Way)*

Vous avez la possibilité d'appliquer l'attribut `System.Runtime.Remoting.Messaging.OneWay` à n'importe quelle méthode, statique ou non. Cet attribut indique au CLR que cette méthode ne retourne aucune information. Même si une méthode qui retourne une valeur de retour ou des arguments de retour (i.e définis avec le mot-clé `out`) est marquée avec cet attribut, elle ne retourne rien.

Une méthode marquée avec l'attribut `OneWay` peut être appelée d'une manière synchrone ou asynchrone. Si une exception est lancée et non rattrapée durant l'exécution d'une méthode marquée avec l'attribut `OneWay`, elle est propagée si l'appel est synchrone. Dans le cas d'un appel asynchrone sans retour l'exception n'est pas propagée. Dans la plupart des cas, les méthodes marquées sans retour sont appelées de manière asynchrone.

Les appels asynchrones sans retour effectuent en général des tâches annexes dont la réussite ou l'échec n'ont pas d'incidence sur le bon déroulement de l'application. La plupart du temps, on les utilise pour communiquer des informations sur le déroulement de l'application.

# *Affinité entre threads et ressources*

Vous pouvez grandement simplifier la synchronisation des accès à vos ressources en utilisant la notion d'affinité entre threads et ressources. L'idée est d'accéder à une ressource toujours avec le même thread. Ainsi, vous supprimez le besoin vous protéger des accès concurrents puisque la ressource n'est jamais partagée. Le *framework* .NET présente plusieurs mécanismes pour implémenter ce concept d'affinité.

## *L'attribut System.ThreadStatic*

Par défaut, un champ statique est partagé par tous les threads d'un processus. Ce comportement oblige le développeur à synchroniser les accès à un tel champ. En appliquant l'attribut `System.ThreadStaticAttribute` sur un champ statique, vous pouvez contraindre le CLR à créer une instance de ce champ pour chaque thread du processus. Ainsi, l'utilisation de ce mécanisme est bien un moyen d'implémenter la notion d'affinité entre threads et ressources.

Il vaut mieux éviter d'initialiser directement lors de sa déclaration un champ statique qui est marqué avec cet attribut. En effet, dans ce cas seul le thread qui charge la classe effectuera l'initialisation sur sa propre version du champ. Ce comportement est illustré par le programme suivant :

*Exemple 5-24 :*

```
using System.Threading ;
class Program {
   [System.ThreadStatic]
   static string str = "Valeur initiale ";
   static void DisplayStr() {
      System.Console.WriteLine("Thread#{0} Str={1}",
         Thread.CurrentThread.ManagedThreadId , str) ;
   }
   static void ThreadProc() {
      DisplayStr() ;
      str = "Valeur ThreadProc" ;
      DisplayStr() ;
   }
   static void Main() {
      DisplayStr() ;
      Thread thread = new Thread(ThreadProc) ;
      thread.Start() ;
```

```
      thread.Join() ;
      DisplayStr() ;
   }
}
```

Ce programme affiche ceci :

```
Thread#1 Str=Valeur initiale
Thread#2 Str=
Thread#2 Str=Valeur ThreadProc
Thread#1 Str=Valeur initiale
```

## *Thread local storage*

La notion d'affinité entre threads et ressources peut être implémentée à l'aide du concept de *thread local storage* (souvent nommé *TLS*). Ce concept n'est pas nouveau et existe au niveau de *win32*. D'ailleurs le *framework* .NET se base sur cette implémentation.

Le concept de TLS utilise la notion de *slot de données*. Un slot de données est une instance de la classe `System.LocalDataStoreSlot`. Un slot de données peut être vu comme un tableau d'objets. La taille de ce tableau est en permanence égale au nombre de threads dans le processus courant. Ainsi, chaque thread a son propre objet dans le slot de données. Cet objet est invisible des autres threads. Pour chaque slot de données, le CLR s'occupe d'établir la correspondance entre les threads et leurs objets. La classe `Thread` fournit les deux méthodes suivantes pour accéder en lecture et en écriture à un objet stocké dans un slot de données :

```
static public object GetData(LocalDataStoreSlot slot) ;
static public void SetData(LocalDataStoreSlot slot, object obj) ;
```

### *Les slots de données nommés*

Vous avez la possibilité de nommer un slot de données pour l'identifier. La classe `Thread` fournit les méthodes suivantes pour créer, obtenir ou détruire un slot de données nommé :

```
static public LocalDataStoreSlot AllocateNamedDataSlot(string slotName) ;
static public LocalDataStoreSlot GetNamedDataSlot(string slotName) ;
static public void FreeNamedDataSlot(string slotName) ;
```

Le ramasse-miettes ne détruit pas les slot de données nommés. Cette responsabilité incombe donc au développeur.

Le programme suivant utilise un slot de données nommé pour fournir un compteur à chaque thread du processus. Ce compte est incrémenté à chaque appel de la méthode `fServer()`. Ce programme bénéficie des TLS dans la mesure où la méthode `fServer()` ne prend pas de référence vers un compteur en argument. Un autre avantage est que le développeur n'a pas à maintenir lui-même un compteur pour chaque thread.

*Exemple 5-25 :*

```
using System ;
using System.Threading ;
class Program {
   static readonly int NTHREAD = 3 ;    // 3 threads à créer.
```

```
    // 2 appels à  fServer() pour chaque thread créé.
    static readonly int MAXCALL = 2 ;
    static readonly int PERIOD = 1000 ; // 1 seconde entre chaque appel.
    static bool fServer() {
       LocalDataStoreSlot dSlot = Thread.GetNamedDataSlot("Compteur");
       int compteur = (int)Thread.GetData(dSlot);
       compteur++ ;
       Thread.SetData(dSlot, compteur);
       return !(compteur == MAXCALL) ;
    }
    static void ThreadProc() {
       LocalDataStoreSlot dSlot = Thread.GetNamedDataSlot("Compteur");
       Thread.SetData(dSlot, (int) 0);
       do{
          Thread.Sleep(PERIOD) ;
          Console.WriteLine(
             "Thread#{0} J'ai appelé fServer(), Compteur = {1}",
             Thread.CurrentThread.ManagedThreadId ,
             (int)Thread.GetData(dSlot)) ;
       } while (fServer()) ;
       Console.WriteLine("Thread#{0} bye",
             Thread.CurrentThread.ManagedThreadId ) ;
    }
    static void Main() {
       Console.WriteLine(
          "Thread#{0} Je suis le thread principal, hello world",
          Thread.CurrentThread.ManagedThreadId ) ;
       Thread.AllocateNamedDataSlot("Compteur");
       Thread thread ;
       for (int i = 0 ; i < NTHREAD ; i++) {
          thread = new Thread(ThreadProc) ;
          thread.Start() ;
       }
       // Nous n'utilisons pas un mécanisme pour attendre la
       // terminaison des threads fils, aussi faut-il attendre
       // assez longtemps pour les laisser finir leur travail.
       Thread.Sleep( PERIOD * (MAXCALL + 1) ) ;
       Thread.FreeNamedDataSlot("Compteur");
       Console.WriteLine("Thread#{0} Je suis le thread principal,bye.",
          Thread.CurrentThread.ManagedThreadId ) ;
    }
}
```

Ce programme affiche ceci :

```
Thread#1 Je suis le thread principal, hello world
Thread#3 J'ai appelé fServer(), Compteur = 0
Thread#4 J'ai appelé fServer(), Compteur = 0
Thread#5 J'ai appelé fServer(), Compteur = 0
Thread#3 J'ai appelé fServer(), Compteur = 1
```

```
Thread#3 bye
Thread#4 J'ai appelé fServer(), Compteur = 1
Thread#4 bye
Thread#5 J'ai appelé fServer(), Compteur = 1
Thread#5 bye
Thread#1 Je suis le thread principal, bye.
```

## *Les slots de données anonymes*

Vous pouvez appeler la méthode statique `AllocateDataSlot()` de la classe `Thread` pour créer un slot de données anonyme. Vous n'êtes pas responsable de la destruction d'un slot de données anonyme. En revanche, vous devez faire en sorte qu'une instance de la classe `LocalDataStoreSlot` soit visible de tous les threads. Réécrivons le programme précédent avec la notion de slot de données anonyme :

*Exemple 5-26 :*

```
using System ;
using System.Threading ;
class Program {
   static readonly int NTHREAD = 3 ;    // 3 threads à créer.
   // 2 appels à  fServer() pour chaque thread créé.
   static readonly int MAXCALL = 2 ;
   static readonly int PERIOD = 1000 ; // 1 seconde entre chaque appel.
   static LocalDataStoreSlot dSlot;
   static bool fServer() {
      int Counter = (int)Thread.GetData(dSlot);
      Counter++ ;
      Thread.SetData(dSlot, Counter);
      return !(Counter == MAXCALL) ;
   }
   static void ThreadProc() {
      Thread.SetData(dSlot, (int) 0);
      do{
         Thread.Sleep(PERIOD) ;
         Console.WriteLine(
            "Thread#{0} J'ai appelé fServer(), Compteur = {1}",
            Thread.CurrentThread.ManagedThreadId ,
            (int)Thread.GetData(dSlot)) ;
      } while (fServer()) ;
      Console.WriteLine("Thread#{0} bye",
         Thread.CurrentThread.ManagedThreadId ) ;
   }
   static void Main() {
      Console.WriteLine(
         "Thread#{0} Je suis le thread principal, hello world",
         Thread.CurrentThread.ManagedThreadId ) ;
      dSlot = Thread.AllocateDataSlot();
      for (int i = 0 ; i < NTHREAD ; i++){
         Thread thread = new Thread(ThreadProc) ;
```

```
            thread.Start() ;
        }
        Thread.Sleep(PERIOD * (MAXCALL + 1)) ;
        Console.WriteLine("Thread#{0} Je suis le thread principal, bye",
            Thread.CurrentThread.ManagedThreadId ) ;
    }
}
```

## *L'interface System.ComponentModel.ISynchronizeInvoke*

L'interface `System.ComponentModel.ISynchronizeInvoke` est définie comme ceci :

```
public object System.ComponentModel.ISynchronizeInvoke{
public object       Invoke(Delegate method,object[] args) ;
public IAsyncResult BeginInvoke(Delegate method,object[] args) ;
public object       EndInvoke(IAsyncResult result) ;
public bool         InvokeRequired{get;}
}
```

Une implémentation de cette interface peut faire en sorte que certaines méthodes soient toujours exécutées par le même thread, d'une manière synchrone ou asynchrone :

- Dans le scénario synchrone, un thread `T1` appelle une méthode `M()` sur un objet `OBJ`. En fait, `T1` appelle la méthode `ISynchronizeInvoke.Invoke()` en spécifiant un délégué qui référence `OBJ.M()` et un tableau contenant les arguments. Un autre thread `T2` exécute la méthode `OBJ.M()`. `T1` attend la fin de l'exécution puis récupère les informations de retour de l'appel.
- Le scénario asynchrone diffère du scénario synchrone par le fait que `T1` appelle la méthode `ISynchronizeInvoke.BeginInvoke()`. `T1` ne reste pas bloqué pendant que `T2` exécute la méthode `OBJ.M()`. Lorsque `T1` a besoin des informations de retour de l'appel il appelle la méthode `ISynchronizeInvoke.EndInvoke()` qui les lui fournira si `T2` à terminé l'exécution de `OBJ.M()`.

L'interface `ISynchronizeInvoke` est notamment utilisée par le *framework* pour forcer la technologie *Windows Form* à exécuter les méthodes d'un formulaire avec un même thread. Cette contrainte vient du fait que la technologie *Windows Form* est construite autour de la notion de messages *Windows*. Le même genre de problématique est aussi adressée par la classe `System.ComponentModel.BackgroundWorker` décrite en page .

Vous pouvez développer vos propres implémentations de l'interface `ISynchronizeInvoke` en vous inspirant de l'exemple **Implementing ISynchronizeInvoke** fourni par *Juval Lowy* à l'URL `http://docs.msdnaa.net/ark_new3.0/cd3/content/Tech_System%20Programming.htm`.

# *Contexte d'exécution*

Le *framework* .NET 2.0 présente des nouvelles classes qui permettent de capturer et de propager le contexte d'exécution du thread courant à un autre thread :

- `System.Security.SecurityContext`

Une instance de cette classe contient l'identité de l'utilisateur *Windows* sous-jacent sous la forme d'une instance de la classe `System.Security.Principal.WindowsIdentity` ainsi que l'état de la pile du thread sous la forme d'une instance de la classe `System.Threading.CompressedStack`. L'état de la pile est notamment exploité par le mécanisme CAS lors du parcours de la pile.

- `System.Threading.SynchronizationContext`
  Permet de s'affranchir des contraintes de compatibilité entre différents modèles de synchronisation.
- `System.Threading.HostExecutionContext`
  Permet à un hôte du moteur d'exécution d'être pris en compte dans le contexte d'exécution du thread courant.
- `System.Runtime.Remoting.Messaging.LogicalCallContext`
  .NET Remoting permet de propager des informations au travers de contexte .NET Remoting au moyen d'instances de cette classe. Plus d'informations à ce sujet sont disponibles en page 856.
- `System.Threading.ExecutionContext`
  Une instance de cette classe contient la réunion des contextes cités.

Reprenons l'exemple en page 209 qui modifie le contexte de sécurité en impersonifiant l'utilisateur `invité` sur le thread courant :

*Exemple 5-27 :*

```
...
   static void Main() {
      System.IntPtr pJeton ;
      if (LogonUser(
            "invité" ,    // login
            string.Empty, // domaine Windows
            "invitépwd" , // mot de passe
            2,            // LOGON32_LOGON_INTERACTIVE
            0,            // LOGON32_PROVIDER_DEFAUT
            out pJeton)) {
         WindowsIdentity.Impersonate(pJeton) ;
         DisplayContext("Main");
         ThreadPool.QueueUserWorkItem(Callback,null) ;
         CloseHandle(pJeton) ;
      }
   }
   static void Callback(object o) {
      DisplayContext("Callback");
   }
   static void DisplayContext(string s) {
      System.Console.WriteLine(s+" Thread#{0} Current user is {1}",
         Thread.CurrentThread.ManagedThreadId,
         WindowsIdentity.GetCurrent().Name) ;
   }
...
```

Cet exemple affiche ceci :

```
Main Thread#1 Current user is PSMACCHIA\invité
Callback Thread#3 Current user is PSMACCHIA\invité
```

En .NET 1.1 cet exemple afficherait ceci :

```
Main Thread#1 Current user is PSMACCHIA\invité
Callback Thread#3 Current user is PSMACCHIA\pat
```

En effet, en .NET 2.0 le pool de thread propage par défaut le contexte du thread postant une tâche au thread exécutant la tâche. Ce n'est pas le cas en .NET 1.1. L'utilisation de la méthode `ExecutionContext.SuppressFlow()` permet de retrouver le comportement de .NET 1.1 en .NET 2.0 :

*Exemple 5-28 :*

```
...
         DisplayContext("Main") ;
         ExecutionContext.SuppressFlow();
         ThreadPool.QueueUserWorkItem( Callback, null ) ;
...
```

Cet exemple affiche ceci :

```
Main Thread#1 Current user is PSMACCHIA\invité
Callback Thread#3 Current user is PSMACCHIA\pat
```

L'exemple suivant montre comment propager soit même le contexte d'exécution. Tout d'abord il faut le capturer avec la méthode `ExecutionContext.Capture()`. Ensuite, nous en créons une copie que l'on passe au thread du pool solicité. Ce dernier propage le contexte qu'on lui fournit en appelant la méthode `ExceutionContext.Run()`. Cette méthode prend en paramètre un contexte d'exécution et un délégué. Elle invoque sur le thread courant la méthode référencée par le délégué, en ayant au préalable positionné le contexte du thread courant :

*Exemple 5-29 :*

```
...
   static void Main() {
      ...
         WindowsIdentity.Impersonate(pJeton) ;
         DisplayContext("Main") ;
         ExecutionContext ctx = ExecutionContext.Capture();
         ExecutionContext.SuppressFlow();
         ThreadPool.QueueUserWorkItem(
            SetContextAndThenCallback, ctx.CreateCopy() ) ;
         CloseHandle(pJeton) ;
      }
   }
   static void SetContextAndThenCallback(object o) {
      ExecutionContext ctx = o as ExecutionContext;
      ExecutionContext.Run(ctx, Callback, null);
   }
```

```
   static void Callback(object o) {
      DisplayContext("Callback") ;
   }
...
```

Sans surprise, cet exemple affiche :

```
Main Thread#1 Current user is PSMACCHIA\invité
Callback Thread#3 Current user is PSMACCHIA\invité
```

# 6

# *La gestion de la sécurité*

Ce chapitre présente les différentes facettes de la sécurité sous .NET :

- Nous commencerons par exposer la technologie Code Access Security (CAS). La technologie CAS permet de mesurer le degré de confiance que l'on peut avoir en un assemblage en vérifiant sa provenance et en s'assurant sa non falsification.
- Nous verrons ensuite comment mesurer le degré de confiance que l'on peut avoir en un utilisateur. La notion d'utilisateur est implémentée à plusieurs niveaux (*Windows*, ASP.NET, COM+ etc).
- Enfin nous exposerons les différents mécanismes de cryptographie que le *framework* met à notre disposition.

D'autres informations relatives à la sécurité sont disponibles dans cet ouvrage. Notamment en page 660 nous présentons différentes techniques pour établir une communication sécurisée entre deux machines et en page 957 nous présentons la sécurisation d'une application web ASP.NET.

## *Introduction à Code Access Security (CAS)*

### *Notion de code mobile*

Le modèle de déploiement des logiciels a considérablement évolué avec la puissance accrue des réseaux. Nous téléchargeons de plus en plus nos logiciels à partir d'Internet et nous utilisons de moins en moins de supports physiques tel que le CD pour le déploiement. On utilise la métaphore de *code mobile* pour désigner le code de ce type d'application que l'on distribue par l'intermédiaire de réseaux.

Les avantages de l'utilisation d'un réseau pour distribuer un logiciel sont nombreux : disponibilité immédiate, mise à jour en temps réel etc. Cependant, le code mobile pose de gros problèmes

de sécurité. En effet, un individu mal intentionné peut exploiter les faiblesses des différents réseaux et les failles des systèmes d'exploitations pour substituer son propre code à du code mobile ou pour faire parvenir du code mobile sur votre machine. En outre, la simplicité d'obtention du code mobile nous pousse à télécharger des logiciels que l'on n'aurait pas pris la peine de commander. On est donc moins regardant quant à l'éditeur qui publie le logiciel. Il est donc nécessaire de limiter l'ensemble des permissions accordées à du code mobile (peut il détruire des fichiers sur mon disque dur ? peut il avoir accès au réseau ? etc).

La technologie COM adresse ce problème d'une manière grossière. Avant d'exécuter un composant COM qui vient d'être téléchargé, l'utilisateur est averti par une fenêtre popup qui lui laisse le choix d'exécuter ou non le composant. Un certain niveau de garantie quant à la provenance du composant peut être fourni grâce à un mécanisme de certificat mais le problème majeur subsiste : une fois que l'utilisateur a choisi d'exécuter le composant, le code de celui-ci a les mêmes droits que l'utilisateur.

Le fait d'avoir une machine virtuelle telle que le CLR permet à la plateforme .NET d'adresser cette problématique avec un mécanisme beaucoup plus fin que celui de COM. En effet, le CLR est à même d'intercepter et peut empêcher une action malicieuse telle que la destruction d'un fichier avant que celle-ci ne se produise. Ce mécanisme est nommé *CAS* (*Code Access Security*). Il fait l'objet de la présente section.

Le déploiement de code mobile développé avec .NET 2.0 se fait de préférence avec la technologie *ClickOnce* qui fait l'objet d'une section page 76. Il est absolument nécessaire de bien comprendre CAS pour tirer partie de *ClickOnce*.

## *Schéma général de CAS*

.NET définit une vingtaine de permissions (paramétrables) qui peuvent être accordées ou non à l'exécution du code d'un assemblage. Chacune de ces permissions définit les règles qui régissent l'accès à une ressource critique, comme la base des registres ou les fichiers et les répertoires. Accorder de la confiance à un assemblage signifie qu'on lui accorde certaines permissions dont il a besoin pour s'exécuter correctement.

Le mécanisme CAS est utilisé par le CLR lors de deux situations :

- Lors du chargement d'un assemblage le CLR lui attribue des permissions.
- Lorsque du code demande d'effectuer une opération critique, le CLR doit vérifier au préalable que les assemblages contenant ce code en ont tous la permission.

## *Attribution des permissions lors du chargement d'un assemblage*

À l'instar des relations humaines, en .NET la confiance se mérite. Le CLR accorde de la confiance à un assemblage seulement s'il peut en extraire un certain nombre *preuves*. Ces preuves sont relatives à la provenance et à la non corruption des données contenues dans l'assemblage.

L'étape d'accord de permissions à l'assemblage en fonction des preuves qu'il a fourni, est entièrement configurable. Le paramétrage de cette étape est stocké dans des entités nommées *stratégies de sécurité*. Les informations contenues dans une stratégie de sécurité ressemblent à : « Si les informations contenues dans un assemblage prouvent qu'il a été produit par l'entreprise XXX alors on peut lui accorder cet ensemble de permissions ». Nous exposerons comment configurer les stratégies de sécurité. On peut noter qu'à ce stade l'application n'a pas encore reçu la permission de s'exécuter. D'ailleurs il se peut qu'au vu du jeu de preuves cette permission ne lui soit

finalement pas accordée. On peut aussi signaler que le mécanisme de résolution des permissions n'accorde aucune permission par défaut.

Lorsqu'un ensemble de permissions a été accordé à l'assemblage, le code de l'assemblage peut modifier cet ensemble et le comportement de la gestion de la sécurité durant l'exécution. Naturellement **ces modifications ne peuvent jamais accorder plus de permissions qu'il n'en a été accordé à l'assemblage**.

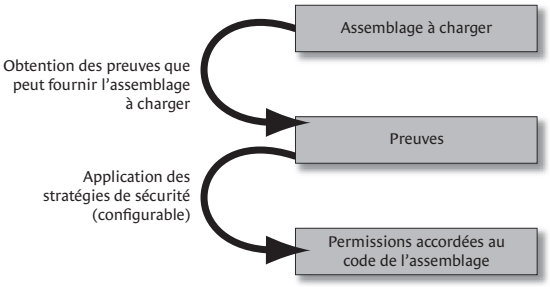


*Figure 6-1 : Attribution des permissions à un assemblage*

## *Vérification des permissions à l'exécution*

Avant d'effectuer une opération critique telle que l'accès à un fichier, le code du *framework* .NET demande au CLR de vérifier si le code appelant en a la permission. Le code appelant n'est pas seulement représenté par la méthode qui demande au *framework* .NET d'effectuer l'opération critique. Le CLR considère que le code appelant est l'ensemble des méthodes qui constitue la pile du thread courant. Le CLR vérifie donc que tous les assemblages contenant toutes ces méthodes ont chacun la permission requise. Ce comportement est appelé *parcours de la pile d'appels* (*stack walk* en anglais). Le parcours de la pile empêche la manipulation frauduleuse d'assemblages ayant un haut degré de confiance (comme ceux développés par *Microsoft*) par des assemblages dans lesquels on ne fait pas confiance.

Dans la figure suivante, on voit que la méthode `File.OpenRead()` demande au CLR de vérifier que tous les appelants ont la permission `FileIOPermissionAccess.Read` sur un fichier avant de procéder à son ouverture. Dans cet exemple, il faudrait que cette permission soit `P3`. Sinon une exception de type `SecurityException` serait automatiquement lancée par le CLR.

Nous aurons l'occasion d'expliquer qu'une permission peut être matérialisée par un objet .NET et que l'on peut appeler la méthode `Demand()` sur un tel objet afin de s'assurer qu'au stade de l'appel, la permission est accordée à tous les appelants.

# *CAS : Preuves et permissions*

## *Qu'est ce qu'une preuve ?*

Une preuve est une information extraite à partir d'un assemblage. Le mot preuve est utilisé dans le sens où si l'on extrait telle ou telle information à partir de l'assemblage, alors cela prouve de

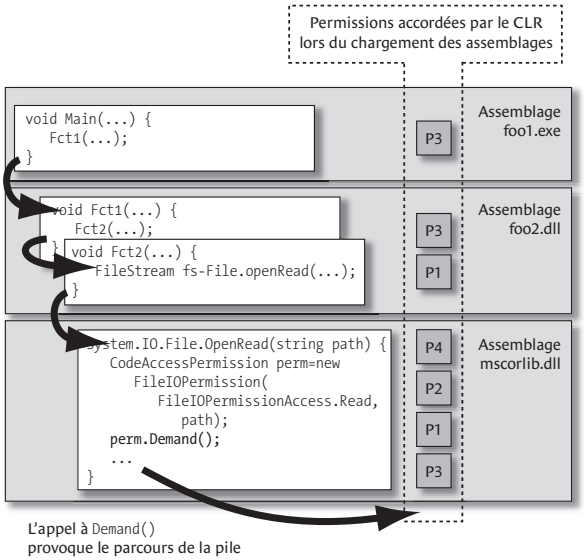


*Figure 6-2 : Le parcours de la pile*

manière irréfutable un fait. Ces faits concernent la provenance et la non falsification de l'assemblage entre le moment où il a été créé par un compilateur chez l'éditeur de logiciel et le moment où il est exécuté chez le client.

## *Les types de preuves standard du Framework .NET*

Voici la liste des huit types de preuves que l'on peut extraire d'un assemblage, et donc, la liste des faits qui peuvent être prouvés à partir d'un assemblage. Chacune de ces preuves peut être matérialisée par une instance d'une classe du *framework* .NET que nous précisons. Ces classes font partie de l'espace de noms System.Security.Policy.

- On peut prouver qu'un assemblage est stocké dans un certain répertoire de la machine. Ce type de preuve peut être représenté par une instance de la classe System.Security.Policy.ApplicationDirectory.
- On peut prouver qu'un assemblage est stocké dans le GAC. Ce type de preuve peut être représenté par une instance de la classe System.Security.Policy.Gac.
- On peut prouver qu'un assemblage a été obtenu/téléchargé à partir d'un certain site (par exemple *www.smacchia.com*). Ce type de preuve peut être représenté par une instance de la classe System.Security.Policy.Site.
- On peut prouver qu'un assemblage a été obtenu à partir d'une certaine URL (par exemple *www.smacchia.com/asm/UnAssemblage.dll*). Ce type de preuve peut être représenté par une instance de la classe System.Security.Policy.Url.
- On peut prouver qu'un assemblage a été obtenu à partir d'une certaine *zone*. .NET présente cinq zones :
  - Internet.

- Un site internet que vous avez ajouté dans votre zone *Sites sensibles* (*untrusted site*) d' *Internet Explorer*.
- Un site internet que vous avez ajouté dans votre zone *Sites de confiance* (*trusted site*) d'*Internet Explorer*.
- Intranet local.
- Le système de stockage local (My Computer).

Chacune de ces zones correspond à une valeur de l'énumération System.Security.SecurityZone. Ce type de preuves peut être représenté par une instance de la classe System.Security.Policy.Zone.

- Si l'assemblage a été signé par son éditeur avec la technologie Authenticode, on peut établir une preuve à partir de ce certificat. Cette technologie est décrite en page 230. Ce type de preuves peut être représenté par une instance de la classe System.Security.Policy.Publisher.
- Si l'assemblage a un nom fort, on peut établir une preuve à partir de ce nom fort. Ce type de preuves peut être représenté par une instance de la classe System.Security.Policy.StrongName. La composante « culture » du nom fort n'est pas prise en compte dans cette preuve. Voici un programme permettant de construire et d'afficher les noms forts des assemblages contenus dans le domaine d'application courant. Notez l'utilisation de la classe System.Security.Permissions.StrongNamePublicKeyBlob pour récupérer la clé publique. Notez qu'en informatique *blob* signifie *Binary Large Objet*. On rappelle qu'une clé publique contient 128 octets en plus des 32 octets d'en-tête.

*Exemple 6-1 :*

```
using System ;
using System.Security.Permissions ;
using System.Security.Policy ;
using System.Reflection ;
[assembly: AssemblyKeyFile("Cles.snk")]
class Program {
   static void DisplayStrongName(Assembly assembly) {
      AssemblyName name = assembly.GetName() ;
      byte[] publicKey = name.GetPublicKey() ;
      StrongNamePublicKeyBlob blob =
         new StrongNamePublicKeyBlob(publicKey);
      StrongName sn = new StrongName(blob, name.Name, name.Version);
      Console.WriteLine(sn.Name) ;
      Console.WriteLine(sn.Version) ;
      Console.WriteLine(sn.PublicKey) ;
   }
   static void Main() {
      Assembly[] assemblies = AppDomain.CurrentDomain.GetAssemblies() ;
      foreach (Assembly assembly in assemblies)
         DisplayStrongName(assembly) ;
   }
}
```

- On peut établir une preuve à partir de la *valeur de hachage* d'un assemblage. La valeur de hachage d'un assemblage permet d'identifier le résultat d'une compilation d'un assemblage, un peu comme un numéro de version sauf que la valeur de hachage ne contient

pas d'information d'ordonnancement temporel comme le fait une version (par exemple la version 2.3 vient toujours après la version 2.1). En outre, même un changement mineur dans le code d'un assemblage suffit à modifier complétement la valeur de hachage, contrairement à la version. Ce type de preuves peut être représenté par une instance de la classe System.Security.Policy.Hash.

Vous pouvez ajouter à cette liste vos propres preuves. Celles-ci doivent être impérativement ajoutées à l'assemblage avant qu'il soit signé. L'idée est de vous permettre de configurer totalement le mécanisme de sécurité. Ce sujet dépasse le cadre de cet ouvrage.

Une instance de la classe System.Security.Policy.Evidence représente une collection de preuves. En fait chaque instance de cette classe contient deux collections de preuves :

- Une collection pour stocker les preuves présentées par le *framework* .NET (un des huit types décrits ci-dessus).
- Une collection pour stocker les preuves propriétaires.

En pratique les développeurs ont peu d'intérêt à manipuler les preuves. Les instances de la classe Evidence sont manipulées en interne par le *framework* .NET. Notamment, nous rappelons qu'une telle collection de preuves est attribuée à chaque assemblage lors de son chargement.

Voici un programme qui affiche les types des preuves fournies par les assemblages du domaine d'application courant. Pour ne pas compliquer inutilement cet exemple, nous n'affichons pas directement les preuves sur la console. Nous préférons afficher le type des preuves :

*Exemple 6-2 :* *PreuveTest.cs*

```
using System ;
using System.Reflection ;
[assembly: AssemblyKeyFile("Cles.snk")]
class Program {
   static void DisplayEvidence(Assembly assembly) {
      Console.WriteLine(assembly.FullName) ;
      foreach (object obj in assembly.Evidence)
         Console.WriteLine("   " + obj.GetType()) ;
   }
   static void Main() {
      Assembly[] assemblies = AppDomain.CurrentDomain.GetAssemblies() ;
      foreach (Assembly assembly in assemblies)
         DisplayEvidence(assembly) ;
   }
}
```

Ce programme affiche ceci :

```
mscorlib, Version=2.0.XXXXX.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
   System.Security.Policy.Zone
   System.Security.Policy.Url
   System.Security.Policy.StrongName
   System.Security.Policy.Hash
PreuveTest, Version=0.0.0.0, Culture=neutral, PublicKeyToken=e0a058df80c8a007
```

```
System.Security.Policy.Zone
System.Security.Policy.Url
System.Security.Policy.StrongName
System.Security.Policy.Hash
```

## *Qui fournit les preuves ?*

Les *preuves* (*evidences* en anglais) d'un assemblage sont fournies au CLR juste avant qu'un assemblage soit chargé dans un domaine d'application :

- Soit par l'*hôte d'un domaine d'application* juste avant le chargement du premier assemblage du domaine d'application. Dans ce cas l'assemblage qui contient l'hôte du domaine d'application doit avoir obtenu la « méta-permission » ControlEvidence. En général vous n'avez pas à vous soucier de cela. En effet, la plupart du temps vous utilisez un hôte de domaine d'application développé par *Microsoft*, en qui les stratégies de sécurité font entièrement confiance. Pour l'instant *Microsoft* en fournit quatre. Ils sont décrits en page 95.
- Soit par le *chargeur de classes* juste avant le chargement d'un assemblage qui contient un type demandé par un assemblage déjà chargé dans le domaine d'application. Puisque le chargeur de classe fait partie intégrante du CLR, le CLR lui fait entièrement confiance et lui accorde la « méta-permission » ControlEvidence.

Dans tous les cas le mécanisme d'obtention de preuves a impérativement besoin de la « méta-permission » ControlEvidence (voir SecurityPermission dans la liste des permission ci-dessous).

## *Les permissions*

Comme son nom l'indique, une *permission* permet à du code d'exécuter un ensemble d'actions. Certains ouvrages qualifient les permissions de *privilèges*, *d'autorisations* ou de *droits*. Ces termes sont cependant très connotés par le vocabulaire *Windows* aussi nous utiliserons le terme de permission dans le présent ouvrage.

On verra dans la section suivante quel est l'algorithme qui permet d'obtenir l'ensemble des permissions pour un assemblage en fonction des preuves apportées par l'assemblage. En .NET, il existe quatre catégories de permissions : les permissions standard, les permission d'identité, les méta-permissions et les permissions propriétaires.

## *Les permissions standard*

Une trentaine de permissions standard permettent de définir l'ensemble des ressources systèmes exploitées par un assemblage. Chacune de ces permissions est matérialisée par une classe qui dérive de la classe System.Security.CodeAccessPermission. Cette classe contient des méthodes qui permettent à partir du code de s'assurer qu'on a une permission, de demander une permission de refuser une permission etc. Bien évidemment ces méthodes ne permettent pas d'obtenir une permission que les stratégies de sécurité ne nous accordent pas. Nous détaillerons l'utilisation de ces classes à la fin de cette section. Voici la liste des classes de permissions standard :

```
System.Security.Permissions.EnvironmentPermission
System.Security.Permissions.FileDialogPermission
System.Security.Permissions.FileIOPermission
System.Security.Permissions.IsolatedStoragePermission
```

```
System.Security.Permissions.ReflectionPermission
System.Security.Permissions.RegistryPermission
System.Security.Permissions.UIPermission
System.Security.Permissions.DataProtectionPermission
System.Security.Permissions.KeyContainerPermission
System.Security.Permissions.StorePermission
System.Security.Permissions.SecurityPermission
System.Configuration.UserSettingsPermission
System.Security.Permissions.ResourcePermissionBase
   System.Diagnostics.EventLogPermission
   System.Diagnostics.PerformanceCounterPermission
   System.DirectoryServices.DirectoryServicesPermission
   System.ServiceProcess.ServiceControllerPermission
System.Net.DnsPermission
System.Net.SocketPermission
System.Net.WebPermission
System.Net.NetworkInformation.NetworkInformationPermission
System.Net.Mail.SmtpPermission
System.Web.AspNetHostingPermission
System.Messaging.MessageQueuePermission
System.Drawing.Printing.PrintingPermission
System.Data.Common.DBDataPermission
   System.Data.OleDb.OleDbPermission
   System.Data.SqlClient.SqlClientPermission
   System.Data.Odbc.OdbcPermission
System.Data.OracleClient.OraclePermission
System.Data.SqlClient.SqlNotificationPermission
System.Transactions.DistributedTransactionPermission
```

## *Les permissions d'identité*

Les *permissions d'identité* (*identity permission* en anglais) : Pour pratiquement chaque preuve apportée par un assemblage, le CLR accorde à l'assemblage une permission d'identité. Les classes prévues pour les permissions d'identité sont :

```
System.Security.Permissions.PublisherIdentityPermission
System.Security.Permissions.SiteIdentityPermission
System.Security.Permissions.StrongNameIdentityPermission
System.Security.Permissions.UrlIdentityPermission
System.Security.Permissions.GacIdentityPermission
System.Security.Permissions.ZoneIdentityPermission
```

Ces classes dérivent aussi de la classe CodeAccessPermission ce qui permet de les traiter comme toutes les autres permissions à partir du code. Ce qui différencie les permissions d'identité des permissions présentées ci-dessus, est le fait que l'accord ou non d'une permission d'identité ne dépend pas d'une stratégie de sécurité mais seulement des preuves fournies par l'assemblage. En outre, une permission d'identité ne vous permet pas de réaliser une action que vous n'auriez pu réaliser sans elle. Une telle permission n'est utilisée que dans le cadre de vérifications.

## *Les méta-permissions*

Les « meta-permissions » ou *permissions de sécurité* : Ce sont des permissions allouées au gestionnaire de sécurité lui-même. La liste des meta-permissions est disponible à l'article **SecurityPermissionFlag Enumeration** des **MSDN**. On peut citer la méta-permission d'exécuter du code non géré (valeur `UnmanagedCode`), la méta-permission de fournir des preuves à partir d'un assemblage (présentée un peu plus haut, valeur `ControlEvidence`), la méta-permission d'exécuter du code non protégé (valeur `SkipVerification`) etc. La classe System.Security.Permissions.SecurityPermission qui dérive de la classe CodeAccessPermission permet de manipuler les méta-permissions à partir du code, sans toutefois pouvoir s'auto-attribuer de telles permissions.

Comprenez que la méta-permission `UnmanagedCode` est une espèce de *super permission* puisqu'elle permet de s'affranchir de toutes les autres permissions en donnant l'accès à l'API win32. De même la méta-permission `SkipVerification` peut être utilisée de façon à contourner les vérifications du CLR. **En conséquence, du code mobile ne devrait jamais avoir une des permissions `UnmanagedCode` ou `SkipVerification`.**

## *Les permissions propriétaires*

Vous pouvez définir vos propres permissions pour l'accès à vos ressources. L'article **Implementing a Custom Permission** des **MSDN** décrit l'utilisation de cette possibilité en détail.

# *CAS : Accorder des permissions en fonction des preuves avec les stratégies de sécurité*

Nous allons clarifier dans la présente section ce qu'est une stratégie de sécurité, de quoi une stratégie de sécurité se compose, selon quel algorithme une stratégie de sécurité est appliquée et enfin, qu'elles sont les configurations par défaut des stratégies de sécurité.

## *Les niveaux de stratégie de sécurité*

Appliquer une *stratégie de sécurité* (*security policy* en anglais) à un assemblage permet d'obtenir un ensemble de permissions accordées en fonction des preuves que le mécanisme CAS a pu obtenir à partir de l'assemblage.

.NET présente quatre stratégies de sécurité. L'ensemble des permissions accordées à un assemblage est l'intersection des ensembles des permissions accordées par chacune de ces stratégies à cet assemblage. Le choix de l'intersection au profit de l'union a été fait car le modèle de sécurité est basé sur l'accord de permissions, et non le retrait de permissions.

| Stratégie de sécurité | Configurée par... | S'applique... |
|---|---|---|
| Entreprise | un administrateur. | au code géré contenu dans les assemblages situés sur les machines d'une entreprise. |

| | | |
|---|---|---|
| Machine | un administrateur. | au code géré contenu dans les assemblages stockés sur la machine. |
| Utilisateur | un administrateur ou l'utilisateur concerné. | au code géré contenu dans les processus qui s'exécutent avec les droits de l'utilisateur concerné. |
| Domaine d'application | l'hôte du domaine d'application. | au code géré contenu dans le domaine d'application. |

Il existe une hiérarchie dans les stratégies de sécurité. Concrètement elles sont appliquées les unes après les autres, dans l'ordre dans lequel elles sont énumérées ci-dessus (de « entreprise » à « domaine d'application »). Pour cette raison on parle de *niveaux de stratégie de sécurité*. L'application d'une stratégie de sécurité peut imposer que les stratégies de sécurité des niveaux suivant ne s'appliquent pas. Par exemple l'application de la stratégie de sécurité « machine » peut empêcher l'application des stratégies de sécurité « utilisateur » et « domaine d'application ». En général, on constate que la plupart des règles de sécurités se trouvent dans la stratégie de sécurité « machine » (d'ailleurs, par défaut les stratégies des autres niveaux accordent toutes les permissions).

## *Quelle est la composition d'une stratégie de sécurité ?*

Une stratégie de sécurité se compose comme ceci :

- Des *groupes de code* (*code groups* en anglais) stockés dans une arborescence,
- Une liste d'*ensembles de permissions* (*permission sets* en anglais),
- Une liste d'assemblages auxquels la stratégie de sécurité donne son entière confiance (*policy assemblies* ou *fully trusted assemblies* en anglais).

À partir de ces éléments et des preuves présentées par un assemblage, on peut calculer un ensemble de permissions. C'est l'application de la stratégie de sécurité. Avant d'exposer l'algorithme utilisé pour cela, il faut expliquer ce que sont les groupes de code.

Un groupe de code associe à une preuve un ensemble de permissions de la liste d'ensemble de permissions de la stratégie de sécurité. Les groupes de code sont stockés dans une arborescence dans une stratégie de sécurité, c'est-à-dire qu'un groupe de code parent peut avoir zéro, un ou plusieurs groupes de code enfants. Il n'y a pas d'obligation de relation entre la preuve d'un groupe enfant et la preuve de son groupe parent, il en va de même pour les permissions accordées. Cependant afin de faciliter l'administration de la sécurité, il est recommandé (dans la mesure du possible) de positionner les relations (parent enfant) avec des liens logiques et de définir les permissions accordées de façon hiérarchique.

Pour comprendre pourquoi les groupes de code sont stockés dans une arborescence, il faut se pencher sur l'algorithme utilisé lors de l'application d'une stratégie de sécurité.

Les documentations officielles utilisent ce vocabulaire : si l'une des preuves d'un assemblage est identique à la preuve d'un groupe de code, on dit que l'assemblage est membre de ce groupe de code. Cela justifie l'appellation groupe de code. La preuve d'un groupe de code permet de définir un groupe d'assemblages (de code) : ce groupe est constitué par les assemblages sui vérifient la preuve.

Sachez que dans le cas où vous fabriqueriez vos propres preuves, il faudrait fabriquer vos propres groupes de code pour les exploiter.

## *Algorithme utilisé lors de l'application d'une stratégie de sécurité*

- Si l'assemblage fait partie de la liste d'assemblages auxquels la stratégie de sécurité donne son entière confiance, elle lui accorde la *super permission* nommée FullTrust que nous décrivons un peu plus loin.
- Sinon, l'algorithme commence à parcourir l'arborescence des groupes de code selon les règles suivantes :
  - L'algorithme vérifie si l'assemblage est membre de chaque groupe de code racine.
  - Les groupes de code enfants d'un groupe de code sont pris en compte par l'algorithme seulement si l'assemblage est membre du groupe de code parent.

  L'ensemble des permissions accordées à l'assemblage par une stratégie de sécurité est l'union des ensembles de permissions des groupes de code dont l'assemblage est membre.
- Chaque groupe de code peut être marqué de façon à finaliser la stratégie de sécurité à laquelle il appartient. Dans ce cas, si un assemblage appartient à ce groupe de code le mécanisme d'évaluation n'ira pas évaluer les niveaux de stratégies de sécurité suivantes.
- Chaque groupe de code peut être marqué comme exclusif. Dans ce cas, si un assemblage appartient à ce groupe de code il ne bénéficiera que des permissions associées à ce groupe.

Si un assemblage appartient à deux groupes de code exclusifs de la même stratégie de sécurité, aucune permission ne lui sera accordée.

Comprenez bien que nous avons parlé ici de l'algorithme de l'application d'un seul niveau de stratégie de sécurité. Rappelez-vous qu'au final, l'ensemble de permissions accordé à l'assemblage est l'intersection des ensembles de permissions accordés par chaque niveau de stratégie de sécurité. Une conséquence est qu'un assemblage à qui une stratégie de sécurité donne son entière confiance n'aura pas obligatoirement toutes les permissions. Il suffit qu'au moins une autre stratégie de sécurité appliquée ne lui fasse pas entièrement confiance.

## *Configuration par défaut des stratégies de sécurité*

Par défaut, la stratégie de sécurité « machine » est configurée avec l'arborescence de groupe de code de la Figure 6-3. Comme vous pouvez le voir, il y a un groupe de code selon la zone d'où provient l'assemblage. On rappelle qu'une zone .NET définit la provenance d'un assemblage et que tout assemblage fournit une preuve quant à la zone dont il est issu. Notez aussi que par défaut, la stratégie de sécurité « machine » fait entièrement confiance dans les assemblages signés avec la clé privée correspondant au jeton de clé publique de *Microsoft* ou de l'*ECMA*.

Par défaut les autres stratégies de sécurité (entreprise, utilisateur et domaine d'application) font entièrement confiance à tous les assemblages. Donc par défaut tous se passe comme s'il n'y avait que la stratégie de sécurité « machine ».

## *Configurer des stratégies de sécurité*

Il existe deux outils permettant de configurer les stratégies de sécurité :

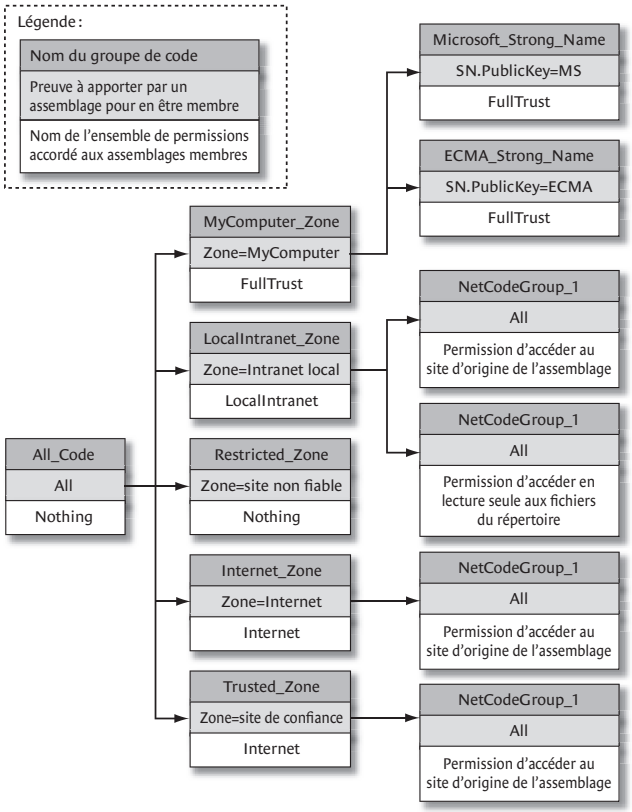


*Figure 6-3 : Configuration par défaut de la stratégie de sécurité 'machine'*

- L'outil graphique de configuration : *.NET Framework Configuration Tool* mscorcfg.msc. Cet outil gère aussi d'autres aspects de .NET comme le Remoting. Vous pouvez lancer cet outil comme ceci : *Panneau de configuration* ▶ *Outils d'administration* ▶ *Microsoft .NET Framework 2.0 Configuration.*
- Un outil utilisable en ligne de commande : caspol.exe.

Au niveau de la sécurité .NET, ces deux outils ont exactement la même fonctionnalité : la configuration des stratégies de sécurité « entreprise », « machine » et « utilisateur » de la machine. La stratégie de sécurité d'un domaine d'application ne peut se faire que programmatiquement, en utilisant les classes adéquates du *framework* .NET.

La Figure 6-4 présente une vue générale de mscorcfg.msc qui permet de retrouver immédiatement les notions présentées que l'on vient d'exposer :

Lorsque vous êtes un administrateur, pour chaque stratégie de sécurité vous pouvez (ou lorsqu'un utilisateur veut configurer la stratégie de sécurité le concernant) :

- Ajouter/modifier/supprimer des ensembles de permissions.

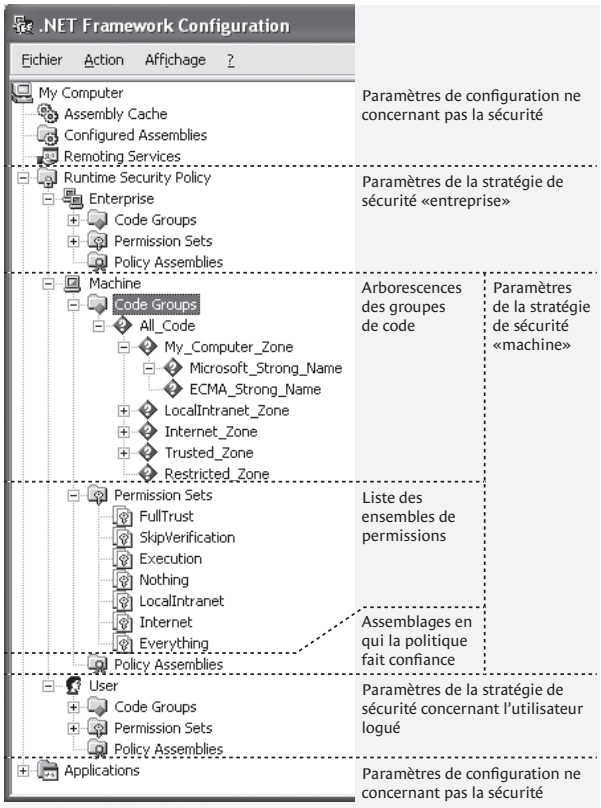


*Figure 6-4 : Vue générale de l'outil de configuration de .NET Framework*

- Ajouter/supprimer des assemblages en qui la politique fait confiance.
- Ajouter/modifier/supprimer des groupes de code dans l'arborescence.
- Exporter ou importer la stratégie de sécurité dans un (ou à partir d'un) fichier de déploiement .msi. Ces fonctionnalités sont accessibles dans les menus *Stratégies de sécurité* ▶ *Créer un fichier de déploiement...* et *Stratégies de sécurité* ▶ *Ouvrir ...*
- Pour un assemblage donné, vous pouvez obtenir la liste des groupes de code (d'une politique ou de toutes les politiques) dont il est membre, et la liste des permissions qui lui sont accordées par la ou les stratégies de sécurité concernées. Cette fonctionnalité est accessible dans le menu *Stratégies de sécurité* ▶ *Evaluer un assemblage...*

Sur une machine donnée, les paramètres d'une stratégie de sécurité sont stockés dans des fichiers de configuration au format XML. Voici leur localisation :

| **Stratégie de sécurité « entreprise »** | |
|---|---|
| Windows XP/2000/NT | %runtime install path%\vXXXX\Config**Enterprisesec.config** |

| | |
|---|---|
| Windows 98/Me | %runtime install path%\vXXXX\Config**Enterprisesec.config** |
| **Stratégie de sécurité « machine »** | |
| Windows XP/2000/NT | %runtime install path%\vXXXX\Config**Security.config** |
| Windows 98/Me | %runtime install path%\vXXXX\Config**Security.config** |
| **Stratégie de sécurité « utilisateur »** | |
| Windows XP/2000/NT | %USERPROFILE%\Application data\Microsoft\CLR security config\vXXXX**Security.config** |
| Windows 98/Me | %WINDIR%*username*\CLR security config\vXXXX**Security.config** |

Ces paramètres sont stockés pour chaque version du CLR. Ainsi, si plusieurs versions du CLR cohabitent, chacune à ses propres paramètres de sécurité.

Le fait que ces paramètres de configuration soient stockés au format XML offre des possibilités intéressantes, comme l'import de groupes de code ou d'ensembles de permissions présentées au format XML. Les trois articles **Importing a Permission Using an XML File**, **Importing a Permission Set Using an XML File** et **Importing a Code Group Using an XML File** des **MSDN** traitent ce sujet en détail.

Les directives de l'outil caspol.exe accessibles en ligne de commande, sont décrites dans les **MSDN** à l'article **Code Access Security Policy Tool**. À travers les nombreuses directives de cet outil, vous y retrouverez toutes les fonctionnalités présentées ci-dessus.

# *CAS : La permission FullTrust*

Dans la section précédente, nous avons cité la permission particulière `FullTrust` que l'on pourrait traduire par *confiance aveugle*. Cette permission permet de désactiver les vérifications du mécanisme CAS. Les assemblages qui ont cette permission ont par conséquent toutes les permissions standard et personnalisées. Du point de vue de CAS il y a deux types d'assemblages : ceux qui ont la permission `FullTrust` et ceux qui ne l'ont pas. CAS n'accorde à ces derniers qu'une *confiance partielle*, (*partially trusted assemblies* en anglais). En particulier, CAS n'accorde qu'un confiance partielles aux assemblage qui ont l'ensemble de permission nommé `Everything`. En effet, cet ensemble accorde par défaut toutes les permissions standard mais ne prend pas en compte les permissions personnalisées.

Par défaut, le code des assemblages signés ne peut être invoqué que par des assemblages qui ont la permission `FullTrust`. Cela vient du fait que seuls les assemblages signés peuvent être placés dans le GAC et peuvent donc être exploités malicieusement par du code mobile. En effet, les assemblages non signés déjà présents sur une machine ne peuvent pas être exploités par du code mobile puisque celui-ci ne peut pas anticiper l'endroit où ils sont stockés, ni deviner les fonctionnalités implémentées.

Cet aspect de la technologie CAS peut se révéler limitatif, aussi, vous avez la possibilité de le désactiver en marquant votre assemblage signé avec l'attribut d'assemblage `System.Security.AllowPartiallyTrustedCallersAttribute`. **Soyez conscient que vous exposez vos clients à de gros risques si vous distribuez largement une bibliothèque de classes marquées avec cet**

**attribut**. Il faut vraiment être certain que le code distribué ne peut pas être détourné. D'ailleurs, seuls certains assemblages standard de *Microsoft* sont marqués avec cet attribut.

Enfin, soyez conscient que ne pas utiliser cet attribut ne représente pas une garantie ultime contre le détournement de votre code. En effet, votre code peut toujours être invoqué à partir d'un assemblage qui a la permission `FullTrust` qui lui même est invoqué à partir d'un assemblage qui n'a pas la permission `FullTrust`.

# *CAS : Vérifier les permissions impérativement à partir du code source*

Vérifier les permissions à partir du code ne signifie absolument pas s'octroyer des permissions que les stratégies de sécurité ne nous ont pas accordé. Ceci est bien évidemment impossible.

Nous allons commencer par exposer comment vérifier les permissions impérativement à partir du code (i.e en appelant des méthodes spécialisées dans la vérification). Nous nous intéresserons ensuite aux attributs permettant de vérifier déclarativement les permissions (i.e le code d'une méthode marquée par un tel attribut doit avoir telle permission). Nous conclurons par une comparaison de ces deux façons de faire.

## *Les classes CodeAccessPermissions et PermissionSet*

La classe CodeAccessPermission ainsi que ses classes dérivées permettent d'effectuer des opérations sur les permissions accordées au code en cours d'exécution principalement au moyen des méthodes `Demand()`, `Deny()`/`RevertDeny()`, `PermitOnly()`/`RevertPermitOnly()` et `Assert()`/`RevertAssert()`.

Les instances de la classe System.Security.PermissionSet représentent des collections de permissions. Cette classe présente aussi les quatre méthodes citées. Elle permet donc de faciliter les opérations portant sur plusieurs permissions.

## *La méthode Demand()*

La méthode Demand() vérifie que le code courant dispose des permissions représentées par le jeu de permission. Une remontée de la pile des appels est alors déclenchée afin de retrouver la hiérarchie de toutes les méthodes responsable de cet appel. Chaque méthode de cette pile est également testée pour le jeu de permissions. Si l'une d'entre elle ne dispose pas de toutes les permissions requises, alors une exception de type `SecurityException` est levée. L'exécution du code s'arrête et la suite de la méthode à l'origine de la remontée de la pile n'est pas exécutée. Le programme suivant s'assure qu'il a la permission de lire un fichier avant de le faire :

*Exemple 6-3 :*

```
using System.Security ;
using System.Security.Permissions ;
class Program {
   static void Main() {
      string sFichier = @"C:\data.txt" ;
      CodeAccessPermission cap =
         new FileIOPermission(FileIOPermissionAccess.Read, sFichier) ;
```

```
      try{
         cap.Demand();
         // Lis le fichier "C:\data.txt".
      }
      catch ( SecurityException ){
         // Vous n'avez pas la permission de lire "C:\data.txt".
      }
   }
}
```

L'intérêt de demander explicitement les permissions est d'anticiper un éventuel refus et d'adapter le comportement de l'application en conséquence. La demande explicite permet également de mettre en place des stratégies plus sophistiquées dans lesquelles le jeu de permissions testées est éventuellement construit en fonction du contexte comme le rôle de l'utilisateur.

## *Les méthodes Deny() RevertDeny() PermitOnly() et RevertPermitOnly()*

La méthode Deny() des classes CodeAccessPermission et PermissionSet permet de signaler les permissions dont notre code n'a pas besoin. Bien que la plupart des développeurs aient d'autres tâches à faire que de signaler les permissions dont ils n'ont pas besoin, ceci constitue une bonne pratique. La méthode Deny() permet de s'assurer que du code tiers que l'on appelle n'aura pas certaines permissions. Cette pratique permet aussi d'avoir une vision globale de ce qu'une application utilise et permet dès le départ d'un projet de fixer des restrictions. Voici un exemple qui montre comment refuser les permissions dangereuses de modification du répertoire système et de la base des registres.

*Exemple 6-4 :*

```
using System.Security ;
using System.Security.Permissions ;
class Program {
   static void Main() {
      PermissionSet ps = new PermissionSet(PermissionState.None) ;
      ps.AddPermission( new FileIOPermission(
           FileIOPermissionAccess.AllAccess,@"C:\WINDOWS")) ;
      ps.AddPermission( new RegistryPermission(
           RegistryPermissionAccess.AllAccess, string.Empty )) ;
      ps.Deny();
      // Ici on ne peut modifier les fichiers systèmes
      // et les données de la base des registres.
      CodeAccessPermission.RevertDeny();
   }
}
```

Aucune exception ne sera levée si votre code ne disposait pas de ces permissions avant l'appel à Deny().Il est a noter que les restrictions liées à l'appel de la méthode `Deny()` ne sont appliquées que dans la méthode qui l'invoque ainsi que dans les méthodes appelées à partir de celle-ci. Dans tous les cas, à la sortie de la méthode ayant invoquée les restrictions de droits, les paramètres par

défaut sont restaurés. Il est aussi possible d'annuler les limitations de droits avec le méthode `RevertDeny()`.

Une alternative existe au couple de méthodes Deny()/`RevertDeny()`. Le couple de méthode PermitOnly()/`RevertPermitOnly()` permet aussi de modifier temporairement l'ensemble des permissions accordées à la méthode courante. La différence entre ces deux manières est que Deny() spécifie les permissions à ne pas accorder alors que PermitOnly() spécifie les permissions à accorder.

## *Les méthodes Assert() et RevertAssert()*

La méthode `Assert()` permet de spécifier qu'un appelant n'a pas besoin d'avoir une ou plusieurs permissions. Pour cela, la méthode `Assert()` supprime le parcours de la pile d'appel pour ces permissions à partir de là où elle est appelée. La méthode qui appelle `Assert()` doit avoir la méta-permission `SecurityPermission(Assertion)`. En outre elle doit aussi avoir la/les permissions(s) concernée(s) pour que la suppression du parcours de la pile d'appel ait effectivement lieu. Dans le cas contraire l'appel à `Assert()` n'a aucun effet et aucune exception n'est lancée.

L'appel de la méthode `Assert()` peut introduire des vulnérabilités dans le code qui l'appelle mais dans certaines situations son utilisation se révèle absolument nécessaire. Par exemple, la classe standard `FileStream` utilise en interne le mécanisme P/Invoke pour accéder aux fichiers et toutes ses méthodes suppriment le parcours de la pile pour la méta-permission `SecurityPermission(UnmanagedCode)`. Sans cet artifice, tout code qui souhaiterait avoir accès à un fichier devrait avoir la méta-permission `SecurityPermission(UnmanagedCode)` en plus de la permission `FileIOPermission`, ce qui n'est clairement pas acceptable.

Pour supprimer le parcours de la pile d'appel pour vérifier que tous les appelants ont la méta-permission `SecurityPermission(UnmanagedCode)`, il est préférable de marquer la méthode concernée ou sa classe avec l'attribut `System.Security.Suppress\-Unmanaged\-Code\-Security\-Attribute` plutôt que d'utiliser la méthode `Assert()`. En effet, cet attribut indique au compilateur JIT qu'il ne faut pas produire le code pour vérifier que tous les appelants ont la méta-permission `SecurityPermission(UnmanagedCode)` lors d'un appel à du code non géré.

L'exemple suivant supprime le parcours de la pile d'appel pour la permission de lire la base des registres :

*Exemple 6-5 :*

```
using System.Security ;
using System.Security.Permissions ;
class Program {
   static void Main() {
      CodeAccessPermission cap = new RegistryPermission(
         RegistryPermissionAccess.NoAccess, string.Empty ) ;
      cap.Assert() ;
      // Lis la base des registres.
      RegistryPermission.RevertAssert() ;
   }
}
```

Vous ne pouvez appeler Assert() plusieurs fois consécutivement dans une même méthode. Ceci provoque la levée d'une exception. Pour appeler Assert() plusieurs fois consécutivement dans

une même méthode, il faut qu'entre chaque appel, la méthode statique CodeAccessPermission.RevertAccess() soit appelée. Pour supprimer le parcours de la pile d'appel pour plusieurs permissions dans une même méthode il est donc obligatoire d'appeler Assert() sur une instance de PermissionSet.

Par prudence il vaut mieux n'invoquer la méthode `Assert()` qu'au moment où l'on en a besoin et pas, par exemple en début de méthode. Il faut dans le même esprit invoquer la méthode `RevertAssert()` le plus tôt possible. Le mieux est en général de servir d'une structure `try/finally`

Notez enfin que contrairement aux méthodes `Deny()`/`RevertDeny()`, `PermitOnly()`/`RevertPermitOnly()` et `Assert()`/`RevertAssert()`, la méthode `Demand()` est la seule qui n'influe pas sur le déroulement ultérieur des opérations.

## *Les méthodes FromXml() et ToXml()*

Les méthodes FromXml() et ToXml() permettent de construire ou de sauver un ensemble de permissions complexe à l'aide de documents XML.

## *L'interface System.Security.IPermission*

L'interface System.Security.IPermission permet de faire des opérations ensemblistes sur un ensemble de permission. Cette interface est implémentée par la classe `PermissionSet`, la classe CodeAccessPermission ainsi que ses classes dérivées. Voici sa définition :

```
public interface System.Security.IPermission {
    IPermission Union(IPermission rhs) ;
    IPermission Intersect(IPermission rhs) ;
    bool        IsSubsetOf(IPermission rhs) ;
    IPermission Copy() ;
    void        Demand() ;
}
```

Avec la méthode IsSubsetOf() vous pouvez calculer des relations d'inclusion entre permissions. Par exemple la permission qui donne tous les accès au dossier **"C:\MonRep"** inclue la permission qui donne tous les accès au dossier **"C:\MonRep\patrick\"**.

*Exemple 6-6 :*

```
using System.Security ;
using System.Security.Permissions ;
class Program {
   static void Main() {
      string rep1 = @"C:\Monrep" ;
      string rep2 = @"C:\Monrep\Patrick" ;
      CodeAccessPermission cap1 = new FileIOPermission(
         FileIOPermissionAccess.AllAccess, rep1) ;
      CodeAccessPermission cap2 = new FileIOPermission(
         FileIOPermissionAccess.AllAccess, rep2) ;
      bool b = cap2.IsSubsetOf(cap1);
      // Ici b vaut true.
   }
}
```

Vous pouvez aussi calculer des nouvelles permissions à partir d'intersections ou d'unions de permissions avec les méthodes Union() et Intersect(). Vous pouvez ainsi réutiliser des permissions évoluées. Ces types d'opérations ensemblistes sur les permissions sont très utilisés par le gestionnaire de sécurité. En pratique les développeurs n'ont pas beaucoup d'occasions de les utiliser.

# *CAS : Vérifier les permissions déclarativement à partir du code source*

Une alternative existe à l'utilisation impérative des classes PermissionSet et CodeAccessPermission pour manipuler les permissions directement à partir du code source. Cette alternative utilise des attributs standard qui peuvent s'appliquer éventuellement aux méthodes, aux types ou à l'assemblage lui-même. Chacune des classes dérivées de la classe CodeAccessPermission représentant un type de permission a un attribut standard qui lui correspond. Par exemple l'attribut RegistryPermissionAttribute représente les permissions relatives à l'accès de la base des registres, exactement comme la classe RegistryPermission. L'Exemple 6-5 peut ainsi être réécrit comme ceci :

*Exemple 6-7 :*

```
using System.Security.Permissions ;
class Program{
   [RegistryPermission(SecurityAction.Assert)]
   static void Main(){
      // Lis la base des registres.
   }
}
```

Les valeurs de l'énumération System.Security.Permissions.SecurityAction permettent de spécifier la manipulation souhaitée. On peut citer les valeurs `Demand`, `Deny`, `PermitOnly` et `Assert` qui ont les mêmes effets que leurs méthodes homonymes expliquées dans la section précédentes. Cependant, l'énumération SecurityAction présente des valeurs qui permettent de réaliser des opérations non présentées par les classes CodeAccessPermission et PermissionSet :

| **Valeur de SecurityAction** | **Description de l'action** |
|---|---|
| InheritanceDemand | Lorsqu'un assemblage est chargé, permet d'imposer que les types dérivés du type sur lequel s'applique cette action aient les permissions spécifiées. |
| LinkDemand | Force le compilateur JIT à vérifier qu'une ou plusieurs permissions sont accordées à la méthode, sans prendre en compte les permissions accordées aux méthodes appelantes. Cette action est plus permissive que Demand mais moins coûteuse aussi, puisqu'elle n'est vérifiée qu'à la compilation de la méthode par le JIT et pas à chaque appel. |

## *Manipuler les permissions au chargement de l'assemblage*

L'énumération SecurityAction présente aussi les trois valeurs suivantes destinées à être utilisés au niveau d'un assemblage entier. On les utilise pour signifier au CLR que lorsqu'il charge l'assemblage, il doit procéder à des opérations sur l'ensemble des permissions :

| Valeur de SecurityAction | Description de l'action |
|---|---|
| RequestMinimum | Spécifie une ou plusieurs permissions sans lesquels l'assemblage ne peut être chargé. |
| RequestOptional | Spécifie une ou plusieurs permissions requises pour exécuter correctement l'assemblage. Cependant l'assemblage est chargé même si ces permissions ne lui sont pas accordées. On parle de permission optionnelle. |
| RequestRefuse | Lorsqu'un assemblage est chargé, spécifie une ou plusieurs permissions qui ne doivent pas être accordées à l'assemblage. |

Par exemple, un assemblage qui réalise des accès à une base de données à besoin de la permission `SqlClientPermission`. Il peut avoir besoin de la permission `RegistryPermission` pour récupérer des paramètres mais ceci peut être optionnel si il prévoit des paramètres par défaut. Enfin, il se peut qu'il n'ait absolument pas besoin de permissions telles que `WebPermission` ou `UIPermission`. Il faudrait donc qu'il soit marqué par les attributs suivants :

*Exemple 6-8 :* *Program.cs*

```
using System.Security.Permissions ;
using System.Data.SqlClient ;
[assembly: SqlClientPermission(SecurityAction.RequestMinimum)]
[assembly: RegistryPermission(SecurityAction.RequestOptional)]
[assembly: UIPermission(SecurityAction.RequestRefuse)]
[assembly: System.Net.WebPermission(SecurityAction.RequestRefuse)]
class Program { public static void Main() { } }
```

L'outil `permview.exe` vous permet de visualiser ces attributs. L'outil `permcalc.exe` décrit en page 82 va plus loin et permet de calculer l'ensemble des permissions requis par un assemblage.

## *Impérative vs. Déclarative*

L'utilisation d'attributs .NET présente les inconvénients suivants (par rapport à la vérification impérative des permissions) :

- Lors de l'échec d'une demande ou d'une assertion de permissions vous ne pouvez pas rattraper d'exception à l'endroit où l'erreur a eu lieu.
- Les arguments passés aux permissions (comme le nom d'un répertoire pour gérer les permissions d'accès à ce répertoire) doivent être connus à la compilation. Plus généralement, vous ne pouvez pas mettre en place une logique de sécurité dynamique (i.e basée sur des informations connues seulement qu'à l'exécution telles que le rôle de l'utilisateur courant par exemple).

Les avantages d'utiliser des attributs pour manipuler des permissions sont :

- La possibilité d'avoir accès à ces attributs et aux paramètres de ces attributs au travers des métadonnées de l'assemblage ou en utilisant l'outil `permview.exe`.
- La possibilité d'utiliser certains de ces attributs sur l'assemblage entier.

# *CAS : Facilités pour tester et déboguer votre code mobile*

.NET 2.0 présente de nouvelles facilités pour tester et déboguer votre code mobile. La classe, `System.Security.SecurityException` présente une dizaine de nouvelles propriétés permettant de récolter beaucoup plus d'information lorsque l'on rattrape une exception de ce type. On peut citer la propriété `AssemblyName FailedAssemblyInfo{get;}` qui contient le nom de l'assemblage qui est à la source de l'échec des vérifications de sécurité. Comprenez bien que cette facilité est à double tranchant : si un telle exception est analysée par un individu mal intentionné, elle lui fourniront autant d'information pour lui permettre d'exploiter les vulnérabilités de votre code.

En page 81 nous exposons les facilités présentées par *Visual Studio 2005* pour la prise en compte des permissions CAS lors du développement d'une application.

# *CAS : La permission de faire du stockage isolé*

La présente section a pour objectif d'expliquer une permission particulière que l'on peut accorder ou non à un assemblage. Cette permission de faire du *stockage isolé* (*isolated storage* en anglais) est la réponse à la problématique suivante :

Donner la permission à une application d'accéder au disque dur témoigne d'une très grande confiance dans cette l'application. *A priori*, peu d'applications dont le code est mobile peuvent prétendre à un tel degré de confiance. Cependant, la plupart des applications ont besoin de stocker des données de manière persistantes, donc sur le disque dur. Ces données sont souvent des journaux d'activité ou des préférences utilisateur. Ne pas autoriser ces applications à stocker leurs données peut les rendre instables, et autoriser ces applications à accéder au disque dur est très dangereux.

Autoriser une application à faire du stockage isolé consiste à lui permettre d'accéder à un répertoire qui lui est réservé sur le disque dur. Vous pouvez spécifier une taille maximale pour ce répertoire. L'application ne peut pas avoir accès aux fichiers qui ne sont pas situés dans ce répertoire. Deux applications différentes auront chacune leur répertoire pour faire du stockage isolé. La responsabilité de nommer ou de localiser un tel répertoire sur une machine où l'application est installée incombe totalement à .NET. Un tel répertoire est parfois nommé *bac à sable* (*sandbox* en anglais).

En fait, le mécanisme de stockage isolé va un peu plus loin que ce qui vient d'être dit. Le choix du nom et de la localisation du répertoire peut non seulement se faire sur l'identité de l'assemblage, mais aussi sur l'identité de l'utilisateur exécutant l'application ou (non exclusif) sur l'identité du domaine d'application contenant l'assemblage. Chacune de ces identités est appelée *portée* (*scope* en anglais). Concrètement l'application aura plusieurs répertoires de stockage isolé, un pour chaque condition d'exécution (i.e un pour chaque valeur du produit cartésien des portées). Chaque fois que l'application s'exécute dans les mêmes conditions, elle utilise le même répertoire. La classe `System.IO.IsolatedStorage.IsolatedStorageFile` présente plusieurs mé-

thodes statiques telles que `GetUserStoreForAssembly()`, `GetMachineStoreForAssembly()`, `GetMachineStoreForDomain()` ou `GetMachineStoreForApplication()` permettant d'obtenir le répertoire correspondant à la portée désirée.

Voici un exemple montrant comment accéder au répertoire de stockage isolé :

*Exemple 6-9 :*

```
using System.IO ;
using System.IO.IsolatedStorage ;
class Program {
   static void Main() {
      // Obtient le répertoire en fonction de l'utilisateur
      // et de l'assemblage courant.
      IsolatedStorageFile isf =
         IsolatedStorageFile.GetUserStoreForAssembly() ;
      IsolatedStorageFileStream isfs = new IsolatedStorageFileStream(
                        "pref.txt", FileMode.Create, isf) ;
      StreamWriter sw = new StreamWriter(isfs) ;
      sw.WriteLine("Mettre ici vos préférences...") ;
      sw.Close() ;
   }
}
```

Le répertoire utilisé par ce programme pour faire du stockage isolé sous mon identité est celui de la Figure 6-5:

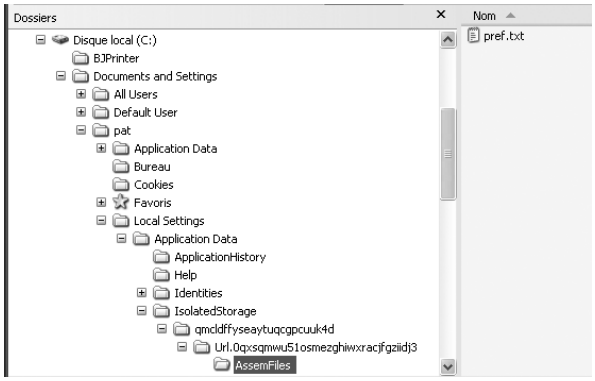


*Figure 6-5 : Répertoire pour faire du stockage isolé*

# *Support .NET pour les utilisateurs et rôles Windows*

La plupart des applications présentent différents niveaux d'utilisations. Par exemple, dans une banque, tous les caissiers utilisateurs d'une application bancaire ne sont pas autorisés à transférer une somme de l'ordre du million d'euros. De même, tous les utilisateurs d'une telle application ne sont pas autorisés à configurer les protocoles d'accès aux bases de données. Enfin

chaque client peut être autorisé à consulter son compte (à partir d'internet). Une telle application nécessite de séparer les utilisateurs entre les clients, les simples caissiers, les caissiers avec des responsabilités et les administrateurs. Chacune de ces catégories est appelée rôle. Pour l'application chaque utilisateur joue zéro, un ou plusieurs rôles. En fonction du cahier des charges de l'application bancaire, les développeurs doivent pouvoir vérifier directement à partir du code le rôle de l'utilisateur courant. Dans le code, cette vérification se fait avant toutes les exécutions d'une fonctionnalité critique. Dans ce contexte, pour une application, accorder de la confiance à un utilisateur revient à déterminer quels sont les rôles qu'il joue.

## *Introduction à la sécurité sous Windows*

Les systèmes d'exploitation *Windows 95/98/Me* n'ont pas de contexte de sécurité. En revanche, sous les systèmes d'exploitation *Windows NT/2000/XP/2003/Vista* toute exécution de code s'effectue dans un contexte de sécurité. À chaque processus *Windows* est associé une identité *Windows* que l'on nomme *principal*. Pour simplifier, vous pouvez considérer qu'un principal est un utilisateur *Windows*. Pour l'instant, nous considèreront qu'un thread s'exécute dans le contexte du principal de son processus. Nous verrons que sous certaines conditions cette règle peut être mise en défaut.

*Windows* associe à chacune de ses ressources (fichiers, registre etc) un ensemble de règles d'accès. Lorsque qu'un thread tente d'accéder à une ressource *Windows*, *Windows* vérifie que le principal associé au thread est autorisé par les règles d'accès.

*Windows* présente la notion de *groupe d'utilisateurs*. Chaque utilisateur appartient à un ou plusieurs groupes. Les règles d'accès aux ressources peuvent être configurées en fonction des utilisateurs et en fonction des groupes. Ainsi, un administrateur peut autoriser ou refuser l'accès à une ressource à tous les utilisateurs d'un groupe en spécifiant un seul groupe plutôt qu'en spécifiant N utilisateurs.

Parmi les *rôles* classiques *Windows* on peut citer le rôle administrateur, utilisateur, invité etc. La notion de groupe permet d'implémenter naturellement le concept de rôle joué par un utilisateur. À chaque rôle correspond un groupe *Windows* et un utilisateur *Windows* joue un rôle s'il fait partie du groupe correspondant.

À chaque authentification d'un utilisateur *Windows* crée une *session logon*. Une session logon est matérialisée par un *jeton de sécurité* (*security token* en anglais). Le principal d'un processus est aussi matérialisé par un jeton de sécurité. Lorsqu'un processus crée un nouveau processus, ce dernier hérite automatiquement du jeton de sécurité de son créateur et évolue donc dans le même contexte de sécurité. Nous précisons qu'en page 135 nous expliquons comment le *framework* .NET vous permet de créer un processus dans un contexte de sécurité différent de celui de son processus parent.

Lors du démarrage, *Windows* crée automatiquement trois sessions logon : la session système, la session locale et la session réseau. Ceci explique le fait que des applications peuvent s'exécuter sur une machine sans qu'un utilisateur n'est été logué (notamment on utilise pour cela la notion de service *Windows*). Cette possibilité est particulièrement exploitée dans le cas de serveurs qui sont susceptibles d'être rebootés automatiquement.

.NET présente plusieurs espaces de noms et de nombreux types permettant d'exploiter programmatiquement la sécurité *Windows*. Comprenez bien que ces types ne font qu'encapsuler les fonctions et les structures *Win32* dédiées à la sécurité.

## *Les interfaces IIdentity et IPrincipal*

Le *framework* .NET présente les deux interfaces suivantes qui permettent de représenter les notions proches d'identité et de principal :

```
interface System.Security.Principal.IIdentity{
   string        AuthenticationType{get;}
   bool          IsAuthenticated{get;}
   string        Name{get;}
}
interface System.Security.Principal.IPrincipal{
   IIdentity    Identity {get;}
   bool         IsInRole(string role) ;
}
```

Tandis que l'interface `IIdentity` représente l'aspect authentification (qui on est ?) l'interface `IPrincipal` représente l'aspect autorisation (qu'est ce qu'on est autorisé à faire ?). Ces interfaces sont notamment utilisées pour manipuler la sécurité *Windows* avec les implémentations System.Security.Principal.WindowsIdentity et System.Security.Principal.WindowsPrincipal. Par exemple le programme suivant récupère l'identité de l'utilisateur associé au contexte de sécurité *Windows* sous-jacent :

*Exemple 6-10 :*

```
using System.Security.Principal ;
class Program{
   static void Main(){
      IIdentity id = WindowsIdentity.GetCurrent();
      System.Console.WriteLine( "Nom : "+id.Name) ;
      System.Console.WriteLine( "Authentifié ? : "+id.IsAuthenticated) ;
      System.Console.WriteLine(
                  "Type d'authentification : "+id.AuthenticationType) ;
   }
}
```

Ce programme affiche :

```
Nom : PSMACCHIA\pat
Authentifié ? : True
Type d'authentification : NTLM
```

Le *framework* .NET présente d'autres implémentations des interfaces `IIdentity` et `IPrincipal` relatives à d'autres mécanismes de sécurité et vous pouvez fournir vos propres implémentations pour développer vos propres mécanismes. On peut ainsi citer la classe System.Web.Security.FormsIdentity exploitée par ASP.NET et la classe System.Web.Security.PassportIdentity exploitée par le mécanisme *Passport*.

Le couple de classes System.Security.Principal.GenericIdentity et System.Security.Principal.GenericPrincipal peut être utilisé comme base à l'implémentation de vos propres mécanismes d'authentification/autorisation mais rien ne vous empêche d'implémenter directement les interfaces `IIdentity` et `IPrincipal`.

## *Les identificateurs de sécurité Windows*

Pour identifier les utilisateurs et les groupes, *Windows* utilise des *identificateurs de sécurité* (SID). Les SID peuvent être vus comme des gros nombres uniques dans le temps et dans l'espace, un peu comme les GUID. Les SID peuvent être représentés par une chaîne de caractères au format *SDDL* (*Security Descriptor Definition Language* un format de représentation textuel non XML) donnant quelques précisions quant à l'entité représentée. Par exemple on peut déduire du SID suivant "S-1-5-21-1950407961-2111586655-839522115-500" qu'il représente un administrateur car il contient le nombre 500 en dernière position (501 aurait indiqué un invité).

Le *framework* .NET présente les trois classes suivantes dont les instances représentent des SID :

```
System.Object
   System.Security.Principal.IdentityReference
      System.Security.Principal.NTAccount
      System.Security.Principal.SecurityIdentifier
```

La classe NTAccount permet de représenter le SID sous une forme lisible par les humains tandis que la classe SecurityIdentifier est utilisée pour communiquer un SID à *Windows*. Chacune de ces classes présente une méthode IdentityReference Translate(Type targetType) permettant d'obtenir une représentation différente d'un même SID.

La classe WindowsIdentity présente la propriété SecurityIdentifier User{get;} permettant de récupérer le SID de l'utilisateur *Windows* courant. L'énumération System.Security.Principal.WellKnownSidType représente la plupart des groupes *Windows* fournis par défaut. Enfin, la classe SecurityIdentifier présente la méthode bool IsWellKnown(WellKnownSidType) qui permet de tester si le SID courant appartient au groupe *Windows* précisé. L'exemple suivant exploite tout ceci pour tester si l'utilisateur courant est un administrateur :

*Exemple 6-11 :*

```
using System.Security.Principal ;
class Program {
    static void Main() {
      WindowsIdentity id = WindowsIdentity.GetCurrent() ;
      SecurityIdentifier sid = id.User ;
      NTAccount ntacc = sid.Translate(typeof(NTAccount)) as NTAccount ;
      System.Console.WriteLine( "SID:       " + sid.Value) ;
      System.Console.WriteLine( "NTAccount: " + ntacc.Value) ;
      if ( sid.IsWellKnown(WellKnownSidType.AccountAdministratorSid) )
         System.Console.WriteLine("...is administrator.") ;
   }
}
```

Cet exemple affiche :

```
SID:       S-1-5-21-1950407961-2111586655-839522115-500
NTAccount: PSMACCHIA\pat
...is administrator.
```

## *Impersonation du thread Windows*

Par défaut, un thread *Windows* évolue dans le contexte de sécurité de son processus. Néanmoins, on peut, à partir du code, associer le contexte de sécurité d'un thread avec un utilisateur en

utilisant la fonction *win32* `WindowsIdentity.Impersonate`(IntPtr jeton). Il faut au préalable loguer l'utilisateur et obtenir un jeton de sécurité avec la fonction LogonUser(). Cet utilisateur n'est pas nécessairement l'utilisateur du contexte de sécurité du processus. Lorsque le contexte de sécurité d'un thread est associé à un utilisateur, on dit que le thread effectue un *emprunt d'identité* (*impersonation* en anglais). Cette technique est exposée par l'exemple suivant :

*Exemple 6-12 :*

```
using System.Runtime.InteropServices ;
using System.Security.Principal ;
class Program{
   [DllImport("Advapi32.Dll")]
   static extern bool LogonUser(
      string   sUserName,
      string   sDomain,
      string   sUserPassword,
      uint   dwLogonType,
      uint   dwLogonProvider,
      out   System.IntPtr Jeton);
   [DllImport("Kernel32.Dll")]
   static extern void CloseHandle(System.IntPtr Jeton) ;

   static void Main(){
      WindowsIdentity id1 = WindowsIdentity.GetCurrent() ;
      System.Console.WriteLine( "Avant impersonation  : " +id1.Name) ;
      System.IntPtr pJeton ;
      if( LogonUser(
            "invité" ,    // login
            string.Empty, // domaine Windows
            "invitépwd" , // mot de passe
            2,            // LOGON32_LOGON_INTERACTIVE
            0,            // LOGON32_PROVIDER_DEFAUT
            out pJeton) ) {
         WindowsIdentity.Impersonate(pJeton) ;
         WindowsIdentity id2 = WindowsIdentity.GetCurrent() ;
         System.Console.WriteLine("Pendant impersonation : "+id2.Name) ;
         // Ici, le thread Windows sous-jacent
         // a l'identité de l'utilisateur 'invité'.
         CloseHandle(pJeton) ;
      }
   }
}
```

Ce programme affiche :

```
Avant impersonation  : PSMACCHIA\pat
Pendant impersonation : PSMACCHIA\invité
```

La méthode `WindowsIndentity.GetCurrent()` accepte une surcharge prenant un paramètre un booléen. Lorsque le booléen est à `true` cette méthode nous retourne l'identité de l'utilisateur seulement si le thread est en cours d'emprunt d'identité. Lorsque le booléen est à `false` cette

méthode nous retourne l'identité de l'utilisateur seulement si le thread n'est pas en cours d'emprunt d'identité.

# *Support .NET pour les contrôles des accès aux ressources Windows*

## *Introduction au contrôle des accès aux ressources Windows*

Après avoir introduit les notions de groupes et d'utilisateurs *Windows* nous allons nous intéresser à la seconde partie de la sécurité sous *Windows* : les contrôles des accès aux ressources.

Une ressource *Windows* peut être un fichier, un objet de synchronisation Windows (mutex, événement...), une entrée de la base des registres etc. Chaque type de ressource présente des *droits d'accès* qui lui sont propres. Par exemple, on peut citer le droit d'accès d'ajouter des données à un fichier et le droit d'accès d'obtenir la possession d'un mutex. Aucun de ces droits d'accès n'a de sens sorti du contexte de son type de ressource.

Chaque ressource contient physiquement des informations qui permettent à Windows de déduire quel utilisateur a quel droit d'accès sur la ressource. Ces informations sont contenues dans une structure associée à la ressource que l'on nomme *descripteur de sécurité* (SD). Un SD contient notamment un SID représentant l'utilisateur qui a créé ou qui détient la ressource et une liste nommée *Discretionary Access Control List* (DACL). Bien que stocké sous une forme binaire, un SD peut être représenté par une chaîne de caractères au format SDDL.

Lorsqu'un thread qui s'exécute dans le contexte de sécurité d'un utilisateur tente d'obtenir un ou plusieurs droits d'accès sur une ressource, *Windows* détermine s'il peut obtenir les droits à partir de la DACL de la ressource.

Une DACL est une liste ordonnée d'*Access Control Element* (ACE). Un ACE est une structure qui associe un SID à une liste de droits d'accès. Une DACL contient deux types d'ACE :

- les ACE qui accordent leurs droits d'accès à leur SID ;
- les ACE qui refusent leurs droits d'accès à leur SID.

Lorsqu'un thread tente d'obtenir des droits d'accès à une ressource, *Windows* établit son verdict à partir du SID du thread et de la DACL de la ressource. Les ACE sont évalués dans l'ordre dans lequel ils sont stockés dans la DACL. Chaque ACE accorde ou retire des droits d'accès lorsque le SID du thread est inclus dans son SID. L'ensemble des droits d'accès demandés est accordé dés que tous les droits d'accès ont été accordés durant l'évaluation. Les droits d'accès sont tous refusés dés qu'un seul des droits d'accès demandés est refusé par un ACE. Comprenez bien que l'ordre de stockage des ACE dans la DACL importe et que *Windows* n'évalue pas nécessairement tous les ACE lors d'une demande de droits d'accès.

Pour certains types de ressources *Windows* permet l'héritage de SD. Cette possibilité se révèle essentielle par exemple pour un administrateur qui souhaiterait configurer les SD de milliers de fichiers contenus dans un répertoire en une seule manipulation.

Chaque SD d'une ressource Windows contient une seconde liste d'ACE nommée *System Access Control List (SACL)*. Cette seconde liste est exploitée par *Windows* pour auditer les accès à une ressource. À l'instar des ACE d'une DACL les ACE d'une SACL associent chacun un SID à une liste de droits d'accès. Au contraire des ACE d'une DACL les ACE d'une SACL contiennent deux informations binaires qui peuvent s'interpréter comme ceci :

- l'événement *un de mes droits d'accès a été accordé à un SID inclus dans mon SID* doit-il être logué ?
- l'événement *un de mes droits d'accès a été refusé à un SID inclus dans mon SID* doit-il être logué ?

Clairement, l'ordre de stockage des ACE dans une SACL n'importe pas.

Le nouvel espace de noms `System.Security.AccessControl` définit des types permettant d'exploiter les SD. Après avoir présenté les types dédiés à la manipulation des SD de ressources spécifiques *Windows* nous présenterons des types prévus pour exploiter les SD d'une manière générique (i.e indépendante du type de ressource *Windows* sous jacent).

## .NET et la manipulation de SD spécifiques

Les types relatifs à des ressources spécifiques consistent en une hiérarchie de types représentant les SD, une hiérarchie de types représentant les ACE et des énumérations représentant les droits d'accès. Les classes suivantes permettent de représenter les descripteurs de sécurité :

```
System.Object
   System.Security.AccessControl.ObjectSecurity
      System.Security.AccessControl.DirectoryObjectSecurity
         System.DirectoryServices.ActiveDirectorySecurity
      System.Security.AccessControl.CommonObjectSecurity
         Microsoft.Iis.Metabase.MetaKeySecurity
         System.Security.AccessControl.NativeObjectSecurity
            System.Security.AccessControl.EventWaitHandleSecurity
            System.Security.AccessControl.FileSystemSecurity
            System.Security.AccessControl.DirectorySecurity
            System.Security.AccessControl.FileSecurity
            System.Security.AccessControl.MutexSecurity
            System.Security.AccessControl.RegistrySecurity
            System.Security.AccessControl.SemaphoreSecurity
```

Ces classes acceptent des paramètres spécifiques représentant les ACE pour remplir les DACL et SACL. Notez qu'il y a des classes qui représentent les ACE de la DACL (*access rule*) et des classes qui représentent les ACE de la SACL pour l'audit (*audit rule*) :

```
System.Object
   System.Security.AccessControl.AuthorizationRule
      System.Security.AccessControl.AccessRule
         Microsoft.Iis.Metabase.MetaKeyAccessRule
         System.Security.AccessControl.EventWaitHandleAccessRule
         System.Security.AccessControl.FileSystemAccessRule
         System.Security.AccessControl.MutexAccessRule
         System.Security.AccessControl.ObjectAccessRule
            System.DirectoryServices.ActiveDirectoryAccessRule
               System.DirectoryServices.[*]AccessRule
         System.Security.AccessControl.RegistryAccessRule
         System.Security.AccessControl.SemaphoreAccessRule
      System.Security.AccessControl.AuditRule
         Microsoft.Iis.Metabase.MetaKeyAuditRule
```

```
          System.Security.AccessControl.EventWaitHandleAuditRule
          System.Security.AccessControl.FileSystemAuditRule
          System.Security.AccessControl.MutexAuditRule
          System.Security.AccessControl.ObjectAuditRule
             System.DirectoryServices.ActiveDirectoryAuditRule
          System.Security.AccessControl.RegistryAuditRule
          System.Security.AccessControl.SemaphoreAuditRule
```

Voici la liste des énumérations représentant les droits d'accès. Par exemple, l'énumération `FileSystemRights` contient une valeur `AppendData` tandis que l'énumération `MutexRights` contient une valeur `TakeOwnership`.

```
Microsoft.Iis.Metabase.MetaKeyRights
System.Security.AccessControl.EventWaitHandleRights
System.Security.AccessControl.FileSystemRights
System.Security.AccessControl.MutexRights
System.Security.AccessControl.RegistryRights
System.Security.AccessControl.SemaphoreRights
```

Enfin les différents types du *framework* .NET représentant directement les ressources *Windows* concernées (`System.Threading.Mutex`, `System.IO.File` etc) ont des nouveaux constructeurs acceptant des ACL et des méthodes `Set/GetAccessControl` permettant de positionner et d'obtenir les ACL d'une instance. Voici un exemple illustrant tout ceci lors de la création d'un fichier avec une DACL :

*Exemple 6-13 :*

```
using System.Security.AccessControl ;
using System.Security.Principal ;
using System.IO ;
class Program {
   static void Main() {
      // Crée la DACL.
      FileSecurity dacl = new FileSecurity() ;
      // Rempli la DACL avec un ACE.
      FileSystemAccessRule ace = new FileSystemAccessRule(
         WindowsIdentity.GetCurrent().Name,
         FileSystemRights.AppendData | FileSystemRights.ReadData,
         AccessControlType.Allow) ;
      dacl.AddAccessRule(ace) ;
      // Crée un nouveau fichier avec cette DACL.
      System.IO.FileStream fileStream = new System.IO.FileStream(
            @"fichier.bin" , FileMode.Create , FileSystemRights.Write ,
            FileShare.None, 4096 , FileOptions.None, dacl ) ;
      fileStream.Write(new byte[] { 0, 1, 2, 3 }, 0, 4) ;
      fileStream.Close() ;
   }
}
```

Vous pouvez visualiser les droits d'accès aux fichiers `fichier.bin` comme ceci : *Propriété du fichier fichier.bin* ▶ *Sécurité* ▶ *Paramètres avancés* ▶ *Autorisations* ▶ *Modifier les autorisations spéciales*

*accordées au principal avec lequel vous avez exécuté le programme ► lecture de données et ajout de données.*

Si l'onglet *Sécurité* ne s'affiche pas il faut effectuer cette manipulation : *Panneau de configuration ► Options des dossiers ► Affichage ► Utiliser le partage de fichiers simple (recommandé)*.

## .NET et la manipulation de SD génériques

.NET présente des types généraux pour la manipulation des SD. La hiérarchie de types suivante permet de représenter des SD :

```
System.Object
   System.Security.AccessControl.GenericSecurityDescriptor
      System.Security.AccessControl.CommonSecurityDescriptor
      System.Security.AccessControl.RawSecurityDescriptor
```

La hiérarchie de type suivante permet de représenter des ACL, des DACL et des SACL :

```
System.Object
   System.Security.AccessControl.GenericAcl
      System.Security.AccessControl.CommonAcl
         System.Security.AccessControl.DiscretionaryAcl
         System.Security.AccessControl.SystemAcl
      System.Security.AccessControl.RawAcl
```

La hiérarchie de type suivante permet de représenter des ACE :

```
System.Object
   System.Security.AccessControl.GenericAce
      System.Security.AccessControl.CustomAce
      System.Security.AccessControl.KnownAce
         System.Security.AccessControl.CompoundAce
         System.Security.AccessControl.QualifiedAce
            System.Security.AccessControl.CommonAce
            System.Security.AccessControl.ObjectAce
```

L'exemple suivant montre comment créer un SD, comment ajouter des ACE à son DACL, puis comment transformer ce SD en un SD de ressource spécifique *Windows* (le type mutex en l'occurrence) :

*Exemple 6-14 :*

```
using System ;
using System.Security.AccessControl ;
using System.Security.Principal ;
class Program {
   static void Main() {
      // Crée un nouveau descripteur de sécurite.
      CommonSecurityDescriptor csd = new CommonSecurityDescriptor(
                                          false, false, string.Empty) ;
      DiscretionaryAcl dacl = csd.DiscretionaryAcl ;
      // Ajoute un ACE a son DACL.
```

```
      dacl.AddAccess(
         AccessControlType.Allow, // Allow OU Deny.
         WindowsIdentity.GetCurrent().Owner, // Utilisateur courant.
         0x00180000, // Masque : TakeOwnerShip ET Synchronize
      //                          équivalent à
      //(int) MutexRights.TakeOwnership | (int) MutexRights.Synchronize
         InheritanceFlags.None,  // Désactive...
         PropagationFlags.None); // ...l'heritage d'ACE.

      string sSDDL = csd.GetSddlForm( AccessControlSections.Owner ) ;
      Console.WriteLine("Security Descriptor : " + sSDDL) ;

      MutexSecurity mutexSec = new MutexSecurity() ;
      mutexSec.SetSecurityDescriptorSddlForm(sSDDL);
      AuthorizationRuleCollection aces = mutexSec.GetAccessRules(
                                    true, true, typeof(NTAccount)) ;
      foreach (AuthorizationRule ace in aces){
         if (ace is MutexAccessRule){
            MutexAccessRule mutexAce = (MutexAccessRule)ace ;
            Console.WriteLine("-->SID : " +
                              mutexAce.IdentityReference.Value) ;
            Console.WriteLine("   Type de droits d'accès : " +
                              mutexAce.AccessControlType.ToString()) ;
            if (0xffffffff == (uint)mutexAce.MutexRights)
               Console.WriteLine("   Tous les droits !") ;
            else
               Console.WriteLine("   Droits : " +
                                 mutexAce.MutexRights.ToString()) ;
         }
      }
   }
}
```

Cet exemple affiche :

```
Security Descriptor : D:(A;;0xffffffff;;;WD)(A;;0x180000;;;LA)
-->SID : TOUT LE MONDE
   Type de droits d'accès : Allow
   Tous les droits !
-->SID : PSMACCHIA\pat
   Type de droits d'accès : Allow
   Droits : TakeOwnership, Synchronize
```

On remarque que par défaut un DACL d'un nouveau SD contient l'ACE qui donne tous les droits à tout le monde. Vous pouvez utiliser la méthode `CommonAcl.Purge(SecurityIdentifier)` pour supprimer les ACE relatives à un SID dans un ACL.

# .NET et la notion de rôle

De la même façon qu'un thread *Windows* s'exécute dans un contexte de sécurité *Windows*, un thread géré a la possibilité de s'exécuter dans un contexte de sécurité de votre choix. Vous pouvez alors exploiter un mécanisme de sécurité de type rôle/utilisateur autre que celui de *Windows*, tel que celui d'ASP.NET par exemple. Ceci est possible car la classe `Thread` présente la propriété `IPrincipal CurrentPrincipal{get;set;}`. Un principal peut être associé à un thread géré de trois façons différentes :

- Soit vous associez explicitement un principal à un thread géré avec la propriété `Thread.CurrentPrincipal`.
- Soit vous définissez une *politique de principal* pour un domaine d'application. Lorsqu'un thread géré exécutera le code du domaine d'application, la politique de principal du domaine lui associera éventuellement un principal, sauf dans le cas où le thread a été associé explicitement à un principal, auquel cas celui-ci n'est pas modifié.
- Soit vous pouvez décider que tous les threads qui sont créés dans un domaine ou qui pénètrent dans un domaine sans avoir de principal .NET auront un principal particulier. Pour cela il suffit de préciser ce principal avec la méthode `void AppDomain.SetThreadPrincipal (IPrincipal)`.

Ces trois opérations nécessitent la méta-permission CAS `SecurityPermissionFlag.ControlPrincipal` pour être menées à bien.

## Positionner la politique de principal d'un domaine d'application

L'exemple suivant positionne la politique de principal du domaine d'application courant à WindowsPrincipal. C'est-à-dire que lorsqu'un thread géré exécute le code contenu dans le domaine d'application, s'il n'y a pas eu de principal associé explicitement à ce thread géré, alors le principal associé au contexte de sécurité *Windows* sous-jacent lui est associé :

*Exemple 6-15 :*

```
using System.Security.Principal ;
class Program{
   static void Main(){
      System.AppDomain.CurrentDomain.SetPrincipalPolicy(
                  PrincipalPolicy.WindowsPrincipal);
      IPrincipal pr = System.Threading.Thread.CurrentPrincipal;
      IIdentity  id = pr.Identity ;
      System.Console.WriteLine( "Nom : "+id.Name) ;
      System.Console.WriteLine( "Authentifié ? : "+id.IsAuthenticated) ;
      System.Console.WriteLine(
                  "Type d'authentification : "+id.AuthenticationType) ;
   }
}
```

Ce programme affiche :

```
Nom : PSMACCHIA\pat
Authentifié ? : True
Type d'authentification : NTLM
```

Les autres politiques de principal possibles pour un domaine d'application sont :

- Pas de principal associé au thread' (PrincipalPolicy.NoPrincipal). Dans ce cas la propriété `Thread.CurrentPrincipal` vaut `null` par défaut.
- Un principal non authentifié associé au thread (PrincipalPolicy.UnauthenticatedPrincipal). Dans ce cas le CLR associe une instance de `GenericPrincipal` non authentifiée à la propriété `Thread.CurrentPrincipal`.

Cette dernière alternative constitue la politique de principal prise par défaut par tous les domaines d'application.

## *Vérifier l'appartenance à un rôle*

Vous pouvez vérifier le rôle d'un principal d'un thread géré de trois façons différentes :

- Vous pouvez utiliser la méthode IsInRole() présentée par l'interface IPrincipal.

  *Exemple 6-16 :*

```
using System.Security.Principal ;
class Program{
   static void Main(){
      IPrincipal pr = System.Threading.Thread.CurrentPrincipal ;
      if( pr.IsInRole( @"BUILTIN\Administrators" ) ){
         // Ici, le principal est un administrateur.
      }
      else
         System.Console.WriteLine(
    "L'exécution de ce programme requiert les droits d'Administrateur") ;
   }
}
```

  Le lecteur attentif aura remarqué que dans l'Exemple 6-11 nous avons exploité une autre technique pour vérifier qu'un utilisateur *Windows* est membre d'un groupe *Windows*. Cette technique, basée sur l'énumération `WellKnownSidType`, est préférable dans le cas particulier d'utilisateurs et de rôles *Windows*. La raison est que le nom d'un groupe *Windows* diffère selon la langue utilisée en paramètre (par exemple *Administrateurs* en français, *Administrators* en anglais).

- Vous pouvez utiliser la classe System.Security.Permissions.PrincipalPermission. Bien que cette classe ne dérive pas de la classe CodeAccessPermission, elle implémente l'interface IPermission. Cette classe présente aussi les méthodes classiques permettant de manipuler les permissions (FromXml() ToXml() etc). Cette technique offre l'avantage pour les développeurs, de gérer les rôles d'une manière intègre par rapport à la gestion des permissions.

  *Exemple 6-17 :*

```
using System.Security.Permissions ;
class Program{
   static void Main(){
      try{
         PrincipalPermission prPerm = new PrincipalPermission(
```

```
            null, @"BUILTIN\Administrators" );
         prPerm.Demand();
         // Ici, le principal est un administrateur.
      }
      catch(System.Security.SecurityException){
         System.Console.WriteLine(
   "L'exécution de ce programme requiert les droits d'Administrateur") ;
      }
   }
}
```

Un autre avantage de cette technique est qu'elle permet de vérifier plusieurs rôles d'un seul coup :

*Exemple 6-18 :*

```
...
   PrincipalPermission prPermAdmin = new PrincipalPermission(
            null, @"BUILTIN\Administrators" ) ;
   PrincipalPermission prPermUser = new PrincipalPermission(
            null, @"BUILTIN\Users" ) ;
   System.Security.IPermission prPerm = prPermAdmin.Union(prPermUser);
   prPerm.Demand() ;
...
```

À l'instar de la gestion des permissions, vous pouvez utiliser l'attribut .NET PrincipalPermission spécialement conçu pour gérer les rôles .NET :

*Exemple 6-19 :*

```
using System.Security.Permissions ;
class Program{
   [PrincipalPermission(   SecurityAction.Demand,
            Role= @"BUILTIN\Administrators")]
   static void Main(){
      // Ici, le principal est un administrateur.
   }
}
```

La comparaison entre la technique d'utilisation de la classe PrincipalPermission et la technique d'utilisation d'attributs .NET est faite dans la section page 204.

## *Sécurité basée sur les rôles sous COM+*

COM+ est une technologie *Microsoft* permettant à une classe (.NET ou non) d'utiliser des fonctionnalités appelées services d'entreprise. Parmi ces services d'entreprise, il existe un service d'entreprise de gestion de la sécurité à partir de rôles. Pour chaque composant servi (i.e qui utilise COM+), vous pouvez associer les rôles requis pour l'exploitation du composant puis affecter des rôles à chaque utilisateurs. Les rôles COM+ peuvent éventuellement être différents des rôles *Windows* mais en pratique, on utilise souvent les rôles *Windows*. Lorsqu'un assemblage contient des composants servis, il peut vérifier l'appartenance d'un utilisateur à un rôle en utilisant

l'attribut System.EnterpriseServices.SecurityRole sur l'assemblage tout entier ou seulement sur certaines classes ou interfaces de l'assemblage. Pour plus d'information quant à l'exploitation du service d'entreprise de gestion des rôles COM+ et quant à l'unification des notions de rôle *Windows* et de rôle COM+ vous pouvez vous référer à l'article **Unify the Role-Based Security Models for Enterprise and Application Domains with .NET** de *Juval Lowy* disponible à l'URL `http://msdn.microsoft.com/msdnmag/issues/02/05/rolesec/` .

# *.NET et les algorithmes symétriques de cryptographie*

## *Un peu de théorie*

Nous allons expliquer comment Julien et Mathieu peuvent s'échanger des messages d'une manière confidentielle en utilisant un *algorithme symétrique* d'encryptions. Les algorithmes symétriques sont basés sur un système de paire de clés. Avant de pouvoir encrypter un message `M`, Julien et Mathieu doivent choisir un algorithme symétrique et se fabriquer une paire de clés `(S,P)`. Nommons `P(M)` un message `M` encrypté avec la clé `P` et `S(M)` un message `M` encrypté avec la clé `S`. Les propriétés d'un algorithme symétrique sont les suivantes :

- `S(P(M)) = P(S(M)) = M`
- On ne peut obtenir `M` si l'on détient `P(M)` sans connaître la paire de clés `(S,P)`.
- On ne peut obtenir `M` si l'on détient `S(M)` sans connaître la paire de clés `(S,P)`.

On voit que les clés `S` et `P` jouent un rôle symétrique d'où le nom de ce type d'algorithme. Pour que Julien envoi le message `M` à Mathieu d'une manière confidentielle, il doit lui envoyer une des deux versions cryptées du message. Mathieu pourra alors obtenir le message original en appliquant l'algorithme symétrique sur le message crypté avec les deux clés. En pratique, Julien et Mathieu se seront mis d'accord sur la clé utilisée pour l'encryption. Si un tiers intercepte la version cryptée du message, il ne pourra pas obtenir le message original puisqu'il ne détient pas la paire de clé `(S,P)`. Tout ceci est résumé dans la figure suivante :

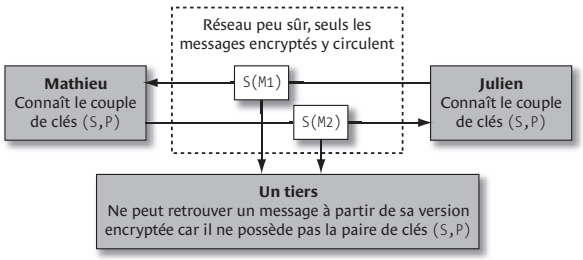


*Figure 6-6 : Echange de messages cryptés avec un algorithme symétrique*

Les algorithmes symétriques ne sont pas secrets et sont largement disséqués dans de nombreuses publications. Dans la mesure où la publication de l'algorithme permet à des milliers de mathématiciens d'en examiner les faiblesses éventuelles on peut même dire que publier l'algorithme de cryptographie participe à la robustesse de l'algorithme.

Seules les clés doivent être gardées confidentielles. Ceci est un principe de la cryptographie apparu relativement récemment, il y a une trentaine d'années. C'est vraiment une nouveauté

puisque pendant des siècles, les algorithmes ont constitué la pierre angulaire de la cryptographie.

## *Le framework .NET et les algorithmes symétriques*

Le *framework* .NET présente une implémentation pour les algorithmes symétriques les plus connus à savoir les algorithmes *DES*, *RC2*, *Rinjdael* et *Triple DES*. Voici la hiérarchie de classes. Seules les classes qui ne sont pas en gras sont abstraites. Ainsi, vous pouvez décider de fournir votre propre implémentation de ces algorithmes.

```
System.Object
    System.Security.Cryptography.SymmetricAlgorithm
        System.Security.Cryptography.DES
            System.Security.Cryptography.DSECryptoServiceProvider
        System.Security.Cryptography.RC2
            System.Security.Cryptography.RC2CryptoServiceProvider
        System.Security.Cryptography.Rijndael
            System.Security.Cryptography.RinjdaelManaged
        System.Security.Cryptography.TripleDES
            System.Security.Cryptography.TripleDESCryptoServiceProvider
```

L'algorithme *DES* (*Digital Encryption Standard*) est l'algorithme symétrique le plus utilisé. L'exemple suivant illustre l'utilisation de la classe `DESCryptoServiceProvider` pour crypter et décrypter une chaîne de caractère. En plus d'une clé, nous fournissons à cet algorithme un *vecteur d'initialisation* (*Initialization Vector* en anglais *IV*). Un vecteur d'initialisation peut être vu comme un nombre en général choisi aléatoirement et utilisé pour initialiser l'algorithme :

*Exemple 6-20 :*

```
using System.Security.Cryptography ;
class Program{
   static void Main() {
      string sMsg = "Le message à encrypter !" ;
      string sEnc, sDec ;
      DESCryptoServiceProvider des = new DESCryptoServiceProvider();
      System.Text.Encoding utf = new System.Text.UTF8Encoding();
      byte[] key = utf.GetBytes("12345678");
      byte[] iv = { 1, 2, 3, 4, 5, 6, 7, 8 };
      ICryptoTransform encryptor = des.CreateEncryptor(key, iv);
      ICryptoTransform decryptor = des.CreateDecryptor(key, iv);
      {
         byte[] bMsg = utf.GetBytes(sMsg) ;
         byte[] bEnc = encryptor.TransformFinalBlock(bMsg, 0,
                                                     bMsg.Length);
         sEnc = System.Convert.ToBase64String(bEnc) ;
      }
      {
         byte[] bEnc = System.Convert.FromBase64String(sEnc) ;
         byte[] bDec = decryptor.TransformFinalBlock(bEnc, 0,
                                                     bEnc.Length);
```

```
            sDec = utf.GetString(bDec) ;
        }
        System.Console.WriteLine("Message : " + sMsg) ;
        System.Console.WriteLine("Encrypté : " + sEnc) ;
        System.Console.WriteLine("Decrypté : " + sDec) ;
    }
}
```

Cet exemple affiche ceci :

```
Message : Le message à encrypter !
Encrypté : iNnbHD1R3nci5tGElIvKIBapaTmfqHEV
Decrypté : Le message à encrypter !
```

Dans l'exemple précédent, notez que les objets `encryptor` et `decryptor` implémentent tous les deux l'interface `ICryptoTransform`. Ceci est une conséquence de l'aspect symétrique des algorithmes.

La classe `DESCryptoServiceProvider` peut aussi être utilisée pour construire une clé et un vecteur d'initialisation. Ainsi l'exemple précédent peut être réécrit comme suit pour exploiter cette possibilité :

*Exemple 6-21 :*

```
...
      des.GenerateKey() ;
      des.GenerateIV() ;
      ICryptoTransform encryptor = des.CreateEncryptor() ;
      ICryptoTransform decryptor = des.CreateDecryptor() ;
...
```

# .NET et les algorithmes asymétriques de cryptographie (clé publique/clé privée)

## Un peu de théorie

Les *algorithmes symétriques* présentent deux faiblesses :

- La paire de clés doit être connue des deux parties souhaitant s'échanger des messages. À un moment donné, il faut que cette paire de clés circule sur un canal de communication entre les deux parties.
- Une paire de clé n'est valable qu'entre deux parties. Si Julien souhaite échanger des messages cryptés avec Mathieu et avec Sébastien il doit détenir deux paires de clés : une pour crypter les échanges avec Mathieu et une pour crypter les échanges avec Sébastien. On voit donc apparaître un problème de gestion de clés.

Les algorithmes asymétriques résolvent ces deux problèmes. Un algorithme asymétrique a les trois propriétés d'un algorithme symétrique que nous rappelons en reprenant les notations de la section précédente :

- `S(P(M)) = P(S(M)) = M`
- On ne peut obtenir `M` si l'on détient `P(M)` sans connaître la paire de clés `(S,P)`.
- On ne peut obtenir `M` si l'on détient `S(M)` sans connaître la paire de clés `(S,P)`.

En plus de ces trois propriétés un algorithme asymétrique a les deux propriétés suivantes :

- Il est aisé de calculer la clé `S` lorsque l'on connaît la clé `P`.
- Il est très difficile de calculer la clé `P` si l'on connaît la clé `S`.

Nous avons donc introduit une dissymétrie dans notre paire de clé d'où le nom de ce type d'algorithme.

On comprend maintenant comment Mathieu et Julien peuvent exploiter ce type d'algorithme pour s'échanger des messages sans s'échanger de clés et sans avoir à gérer un grand nombre de clés. Il suffit qu'ils calculent chacun une paire de clé que nous nommons `(Sj,Pj)` et `(Sm,Pm)`. Mathieu diffuse la clé `Sm` tandis que Julien diffuse la clé `Sj`. Toute personne souhaitant envoyer un message `M` à Mathieu d'une manière confidentielle peut utiliser la clé `Sm` pour l'encrypter. Si Mathieu a pris le soin de garder privée la clé `Pm`, il est alors le seul à pouvoir décrypter `Sm(M)` en calculant `Pm(Sm(M))`. Tout ceci est résumé dans la figure ci-dessous

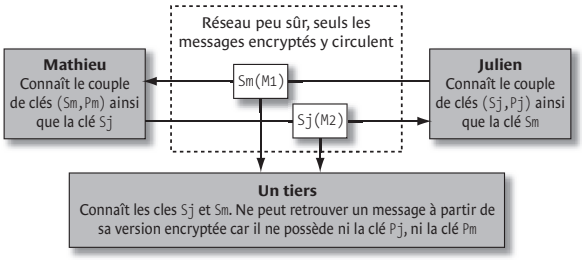


*Figure 6-7 : Echange de messages cryptés avec un algorithme symétrique*

On dit que `S` est la *clé publique* ou partagée (`S` pour *Shared* en anglais). On dit que `P` est la *clé privée* (`P` pour *Private* en anglais). On nomme parfois les algorithmes asymétriques les *algorithmes à clés publiques/clés privées*.

Un problème d'authentification subsiste néanmoins. Puisque la clé `Sm` est connu de tous un tiers peut parfaitement envoyer un message crypté `Sm(M)` à Mathieu en se faisant passer pour Julien. Ce problème peut être contourné en utilisant l'astuce suivante : Julien peut envoyer le message crypté `Sm(Pj(M))` à Mathieu. Mathieu peut alors décrypter ce message puisqu'il détient les clé `Pm` et `Sj` et que `Sj(Pm( Sm(Pj(M)) )) = Sj(Pj(M)) = M`. En outre Mathieu peut être certains que Julien est l'envoyeur puisque seul Julien connaît la clé `Pj`.

## *Notion de session sécurisée*

Un autre problème subsiste : le coût des calculs des algorithmes asymétriques connus est environs 1000 fois plus élevé que le coût des calculs des algorithmes symétriques connus. En pratique, les protocoles sécurisés d'échange de messages utilisent un algorithme asymétrique pour échanger une paire de clé d'un algorithme symétrique puis utilise l'algorithme symétrique pour crypter les messages. La paire de clé de l'algorithme symétrique n'est alors valable que pour une session d'échange de messages. On parle alors de *clé de session* et de *session sécurisée*.

## L'algorithme RSA

L'algorithme *RSA* créé en 1977, est à l'heure actuelle l'algorithme asymétrique le plus utilisé. La protection des cartes bancaires ou des messages militaires l'utilise. La plateforme .NET l'utilise aussi. Le nom de RSA provient du nom de ses inventeurs, *R.L. Rivest*, *A. Shamir* et *L.M Adelman*.

L'algorithme RSA est basé sur une propriété des grands nombres premiers. Soit deux grands nombres premiers `A` et `B`. Il est très facile de calculer le produit de `A` et de `B`. En revanche, lorsque l'on ne connaît que le produit `AB`, il est très difficile de calculer les nombres `A` et `B`. Sans rentrer dans les détails de l'algorithme RSA, vous pouvez considérer que la paire de nombre `(A,B)` définit la clé privée tandis que le produit de `A` et de `B` définit la clé publique.

Tant que l'on ne saura obtenir une paire de nombres premiers `A` et `B` à partir de leur produit qu'en temps polynomial, l'algorithme RSA restera extrêmement fiable. En fait, on ne sait même pas prouver qu'il existe un algorithme en un temps meilleur que le temps polynomial. La plupart des mathématiciens contemporains supposent que ce problème restera inaccessible pendant encore de nombreuses décennies. Néanmoins, l'histoire a montré qu'il est quasiment impossible d'estimer quand un problème mathématique sera résolu.

Sachez enfin que l'on utilise des algorithmes statistiques efficaces pour le calcul de grands nombres premiers. Ce type d'algorithme permet de déterminer si un grand nombre est premier à un degré de certitude qui peut être aussi grand que l'on veut sans jamais être égal à 100%.

## Algorithme asymétriques et signature numérique

En plus de la possibilité de crypter des données, les propriétés des algorithmes asymétriques peuvent être utilisées pour signer numériquement des données. Signer numériquement des données signifie qu'un consommateur de données peut être absolument certains que celui qui a produit des données détient une certaine clé privée. On parle d'authentification de données si le consommateur peut en plus être certains que seul celui qui a produit la donnée détient la clé privée.

Pour comprendre ce qui va être exposé il est nécessaire de comprendre ce qu'est une *valeur de hachage*. Une valeur de hachage est un nombre calculé à partir d'un ensemble de données. Ce calcul a la particularité de fournir deux nombres différents pour deux ensembles de données différents, de manière quasiment sûre. Le *framework* .NET présente dans l'espace de noms `System.Security.Cryptography` les implémentations des principaux algorithmes de hachage. On peut citer les algorithmes *SHAx*, *RIPEMD160* ou *MD5*.

Supposons que Mathieu veuille convaincre Julien qu'il est l'auteur du fichier `FOO`. Il lui faut d'abord calculer une valeur de hachage `x` du fichier. Ensuite Mathieu calcule Pm(x) à partir de sa clé privée. Enfin Mathieu intègre la valeur Pm(x) et sa clé publique `Sm` dans le fichier `FOO` (par exemple au début du fichier).

Julien connaît au préalable la clé publique `Sm` de Mathieu. Julien récupère le fichier `FOO`. Il extrait Pm(x) et la clé publique du fichier. Il vérifie que c'est bien la clé publique de Mathieu. Il peut donc calculer x de deux manières différentes :

- En calculant la valeur de hachage du fichier `FOO` (auquel il a ôté Pm(x) et la clé publique).
- En calculant `x` à partir de Pm(x) et de la clé publique `Sm` car `Sm( Pm(x) )=x`.

Si par ces deux calculs Julien obtient la même valeur, il peut être sûr que l'auteur de `FOO` détient la clé privée Pm. On dit que Pm(x) constitue une *signature numérique* du fichier `FOO`. Si Mathieu

a pu convaincre Julien qu'il est le seul détenteur de la clé privée Pm, Julien peut être sûr que l'auteur de FOO est Mathieu.

Une faille dans cet algorithme existe néanmoins. Nous avons dit que « *Le calcul de la valeur de hachage à la particularité de fournir de manière quasiment sûre deux nombres différents pour deux fichiers différents* ». Si un tiers parvient à trouver une séquence d'octets qui produit la même valeur de hachage, il peut utiliser la signature numérique précédente dans un fichier contenant cette séquence d'octets. Julien n'aura aucun moyen de savoir que ce fichier n'a pas été produit par un détenteur de la clé privée Pm. Cependant la taille des valeurs de hachage est de l'ordre de 20 octets. Il y a donc moins d'une chance sur 10 puissance 48 pour qu'une séquence d'octets prise au hasard fournisse une valeur de hachage donnée.

En page 28 nous expliquons comment la plateforme .NET permet de signer numériquement les assemblages. En page 230 nous verrons que cette technique est utilisée dans les environnements *Windows* depuis bien avant .NET pour authentifier des fichiers. Nous présenterons alors une technologie permettant de pouvoir convaincre qu'on est le seul détenteur d'une certaine clé privée.

## *Le framework .NET et l'algorithme RSA*

Le *framework* .NET présente deux classes permettant d'exploiter l'algorithme RSA : La classe `RSACryptoServiceProvider` pour l'encryption de données et la classe `DSACryptoService-Provider` pour signer numériquement des données (DSA pour *Digital Signature Algorithm*). Voici la hiérarchie des classes :

```
System.Object
    System.Security.Cryptography.AsymmetricAlgorithm
        System.Security.Cryptography.DSA
            System.Security.Cryptography.DSACryptoServiceProvider
        System.Security.Cryptography.RSA
            System.Security.Cryptography.RSACryptoServiceProvider
```

L'exemple suivant expose l'utilisation de la classe `RSACryptoServiceProvider` pour crypter une chaîne de caractères. La méthode `ExportParameter(bool)` permet de récupérer la clé publique ou la paire de clé publique/privée selon qu'elle est appelée avec la valeur `false` ou `true` :

*Exemple 6-22 :*

```
using System.Security.Cryptography ;
class Program {
   static void Main() {
      string sMsg = "Le message à encrypter !" ;
      string sEnc, sDec ;
      System.Text.Encoding utf = new System.Text.UTF8Encoding() ;
      RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
      RSAParameters publicKey = rsa.ExportParameters(false);
      RSAParameters publicAndPrivateKey = rsa.ExportParameters(true);
      {
         RSACryptoServiceProvider rsaEncryptor = new
                                    RSACryptoServiceProvider();
         rsaEncryptor.ImportParameters(publicKey);
         byte[] bMsg = utf.GetBytes(sMsg) ;
```

```
            byte[] bEnc = rsaEncryptor.Encrypt(bMsg, false);
            sEnc = System.Convert.ToBase64String(bEnc) ;
        }
        {
            RSACryptoServiceProvider rsaDecryptor = new
                                        RSACryptoServiceProvider();
            rsaDecryptor.ImportParameters(publicAndPrivateKey);
            byte[] bEnc = System.Convert.FromBase64String(sEnc) ;
            byte[] bDec = rsaDecryptor.Decrypt(bEnc, false);
            sDec = utf.GetString(bDec) ;
        }
        System.Console.WriteLine("Message : " + sMsg) ;
        System.Console.WriteLine("Encrypté : " + sEnc) ;
        System.Console.WriteLine("Decrypté : " + sDec) ;
    }
}
```

Cet exemple affiche ceci :

```
Message : Le message à encrypter!
Encrypté: WVswU5B1JMKdrGrESNngo+s7K/+kvz3o8UaxB5EOsjdejNDmjsuvGEKMP
P3q3OuRXB4k7B5yLwcnaJK2guVmK3ysN+OgmsheOXOUlqUBlzp2EzVsaqpzUQGHxe6k
toOBILR4PU1Jqyq1kESSTfMx9jfTDnMEJ3l1Op+wpQX5DFMs=
Decrypté: Le message à encrypter!
```

## *La taille des clés*

Traditionnellement, la taille des clés des algorithmes symétriques est de 40, 56 et 128 bits. La taille des clés pour les algorithmes asymétriques a tendance à être plus élevée. Pour vous donner une idée, une clé de 40 bits ne résiste que quelques minutes à une attaque déterminée tandis qu'à ma connaissance, aucune clé de 128 bits n'a encore été « craquée ». Pour chaque implémentation du *framework* d'un algorithme vous pouvez obtenir la taille des clés utilisées. Certaines implémentations vous permettent de fixer cette taille.

Bien entendu plus une clé contient de bits plus elle est sûre. Légalement, vous ne pouvez pas utiliser n'importe qu'elle taille. Aussi, les implémentations des algorithmes d'encryptions fournissent la propriété `int[] LegalKeySizes[]{get;}`.

# *L'API de protection des données (Data Protection API)*

## *L'API de protection de données de Windows*

Depuis *Windows 2000*, les systèmes d'exploitation *Windows* présentent une API de cryptographie nommée *DPAPI* (*Data Protection API*). Cette API est implémentée dans la DLL système `crypt32.dll`. Elle a ceci de particulier qu'elle se base sur les crédits accordés au couple login/mot de passe de l'utilisateur courant pour gérer les clés. Elle peut aussi se baser sur l'identité d'un processus, l'identité d'une session *Windows* ou l'identité de la machine courante. En effet, bien souvent nous souhaitons crypter des données pour garantir leur confidentialité au niveau d'un utilisateur, d'un processus, d'une session ou d'une machine. Dans ces cas, l'utilisation de DPAPI nous évite d'avoir à gérer des clés.

Cette API est capable de gérer les modifications de mots de passe. Autrement dit, si vous stockez des données en les encryptant pour un utilisateur donné, vous serez capable de les exploiter même lorsque le mot de passe de cet utilisateur aura été modifié. Ceci est possible grâce à un système de stockage des clés expirées. Plus de détails à ce sujet sont disponibles dans l'article **Windows Data Protection** des **MSDN**.

## *La classe System.Security.Cryptography.ProtectedData*

L'exemple suivant montre comment utiliser la classe `System.Security.Cryptography.ProtectedData` pour protéger des données au niveau d'un utilisateur. Nous aurions pu utiliser la valeur `DataProtectionScope.LocalMachine` pour protéger ces données au niveau de la machine courante. Dans cet exemple, nous exploitons l'option d'ajouter de l'*entropie* à l'encryption. Cela signifie qu'un processus s'exécutant dans le contexte adéquat (i.e sous le bon utilisateur ou sur la bonne machine) n'aura pas la possibilité de décrypter les données s'il ne connaît pas l'entropie utilisée pour les encrypter. Vous pouvez donc considérer l'entropie comme une sorte de clé secondaire :

*Exemple 6-23 :*

```
using System.Security.Cryptography ;
class Program{
   static void Main() {
      string sMsg = "Le message à encrypter !" ;
      string sEnc, sDec ;
      System.Text.Encoding utf = new System.Text.UTF8Encoding() ;
      byte[] entropy = new byte[] { 1, 2, 3, 4, 5, 6, 7, 8 };
      {
         byte[] bMsg = utf.GetBytes(sMsg) ;
         byte[] bEnc = ProtectedData.Protect(
                  bMsg , entropy , DataProtectionScope.CurrentUser);
         sEnc = System.Convert.ToBase64String(bEnc) ;
      }
      {
         byte[] bEnc = System.Convert.FromBase64String(sEnc) ;
         byte[] bDec = ProtectedData.Unprotect(
                   bEnc, entropy, DataProtectionScope.CurrentUser);
         sDec = utf.GetString(bDec) ;
      }
      System.Console.WriteLine("Message : " + sMsg) ;
      System.Console.WriteLine("Encrypté : " + sEnc) ;
      System.Console.WriteLine("Decrypté : " + sDec) ;
   }
}
```

Cet exemple affiche ceci :

```
Message : Le message à encrypter!
Encrypté: AQAAANCMnd8BFdERjHoAwE/Cl+sBAAAA3uy2/pifMEGRALpANT44y
QAAAAACAAAAAAADZgAAqAAAABAAAAA4UMUfUFqt5Xrz5U6hAVJ1AAAAAASAAACg
AAAAEAAAAHMxZCILz7K6JJc4Mmd/4P8YAAAAiiE+UBJdtaReGyP9vMsmw6HsqHM
```

```
LksdXFAAAAMRqfALSa8aaMm1mkestfBudeX91
Decrypté: Le message à encrypter!
```

## *La classe System.Security.Cryptography.ProtectedMemory*

La classe `System.Security.Cryptography.ProtectedMemory` permet de protéger des données au niveau de portées plus fines que celles présentées par la classe `ProtectedData`. Les options présentées par l'énumération `MemoryProtectionScope` sont les suivantes :

- `SameProcess` : Spécifie que seul du code invoqué dans le même processus que celui qui a encrypté les données sera à même de les décrypter.
- `SameLogon` : Spécifie que seul du code invoqué dans un processus dans le même contexte utilisateur que celui qui a encrypté les données sera à même de les décrypter. Cela implique notamment qu'il faut que les opérations d'encryptions et de décryptions d'une même donnée aient lieux durant la même session *Windows*.
- `CrossProcess` : Spécifie que les données peuvent être décryptées par n'importe quel code exécuté dans n'importe quel processus à la condition que le système d'exploitation n'ait pas été redémarré entre l'opération d'encryption et l'opération de décryption.

L'exemple suivant illustre l'utilisation de cette classe. Il faut que les données à encrypter soient stockées sur un tableau d'octets d'une taille multiple de 16 :

*Exemple 6-24 :*

```
using System.Security.Cryptography ;
class Program {
   static void Main() {
      string sMsg = "01234567890123456789012345678901" ;
      System.Text.Encoding utf = new System.Text.UTF8Encoding() ;
      System.Console.WriteLine("Message : " + sMsg) ;
      byte[] bMsg = utf.GetBytes(sMsg) ;
      ProtectedMemory.Protect(bMsg,
                              MemoryProtectionScope.SameProcess);
      System.Console.WriteLine("Encrypté : " + utf.GetString(bMsg)) ;
      ProtectedMemory.Unprotect(bMsg,
                              MemoryProtectionScope.SameProcess);
      System.Console.WriteLine("Decrypté : " + utf.GetString(bMsg)) ;
   }
}
```

Cet exemple affiche ceci :

```
Message : 01234567890123456789012345678901
Encrypté : m;SH<^"?vfn6b{m.Op%L
Decrypté : 01234567890123456789012345678901
```

## *La classe System.Security.SecureString*

La manipulation des chaînes de caractères avec la classe `String` présente plusieurs vulnérabilités si l'on considère qu'un individu mal intentionné peut avoir accès aux pages mémoires d'un processus *Windows* :

- Une même chaîne de caractères peut être dupliquée dans la mémoire du processus du fait que le ramasse miettes se réserve le droit de bouger les instances de types référence.
- Les chaînes de caractères sont stockées en mémoire sans aucune sorte d'encryptions.
- Le fait que les chaînes de caractères soient immuables implique qu'à chaque modification d'une chaîne, son ancienne version réside en mémoire pendant une durée que l'on ne maîtrise pas.
- Une autre conséquence du caractère immuable des chaînes de caractères est que vous n'avez pas la possibilité de nettoyer les octets utilisés pour la stocker lorsque vous n'en avez plus besoin (à moins d'utiliser la technique décrite en page 508).

Pour pallier cette vulnérabilité, le *framework* .NET présente la classe `System.Security.SecureString`. Cette classe dont l'implémentation se base sur les services DPAPI, permet de stocker en mémoire une chaîne de caractère sous une forme encryptée.

Vous pouvez initialiser une instance de `SecureString` soit à partir d'un tableau d'octets (que vous pouvez par la suite mettre à zéro) soit en la construisant caractère après caractère. La classe `Marshal` présente plusieurs méthodes pour récupérer une chaîne de caractères encryptée dans une instance de `SecureString`.

L'exemple suivant montre comment utiliser une instance de `SecureString` pour stocker un mot de passe saisi par un utilisateur sur la console. Nous affichons ensuite cette chaîne de caractères sur la console. Pour les besoins de l'exemple, nous convertissons l'instance de `BSTR` contenant la chaîne de caractère décryptée en une instance de `String`. En pratique, il faut éviter cette manipulation. Fatalement, pour exploiter une chaîne de caractères sécurisée il faut à un moment ou à un autre la décrypter en mémoire. Il faut alors la stocker dans une zone mémoire que l'on maîtrise pour pouvoir la détruire dés que possible et pour éviter les problèmes posés par la classe `String` vus précédemment :

*Exemple 6-25 :*

```
using System ;
using System.Runtime.InteropServices ;
class Program {
   static void Main() {
      System.Security.SecureString pwd = new
                              System.Security.SecureString();
      ConsoleKeyInfo nextKey = Console.ReadKey(true) ;
      while(nextKey.Key != ConsoleKey.Enter){
         pwd.AppendChar(nextKey.KeyChar);
         Console.Write("*") ;
         nextKey = Console.ReadKey(true) ;
      }
      Console.WriteLine() ;
      pwd.MakeReadOnly();
      IntPtr bstr = Marshal.SecureStringToBSTR(pwd) ;
      // Récupère une instance de la classe String
      // pour les besoins de l'exemple.
      try{     Console.WriteLine( Marshal.PtrToStringAuto(bstr) ) ; }
      finally{ Marshal.ZeroFreeBSTR(bstr) ; }
   }
}
```

## *Protection des données dans vos configurations*

La classe `System.Configuration.Configuration` permet de stocker des données encryptées dans le fichier de configuration de l'application. L'exemple suivant montre comment stocker une version encryptée de la chaîne de caractères `MonPassword` dans un paramètre de configuration nommé `TagPassword`. Notez que seule la sauvegarde nécessite une manipulation spéciale pour préciser que l'on souhaite stocker une version encryptée. La décryption de la donnée est effectuée d'une manière transparente lors de son chargement :

*Exemple 6-26 :*

```
using System.Configuration ;
class Program {
   static void Main() {
      SavePwd("MonPassword") ;
      System.Console.WriteLine(LoadPwd()) ;
   }
   static void SavePwd(string pwd) {
      Configuration cfg = ConfigurationManager.OpenExeConfiguration(
                          ConfigurationUserLevel.None) ;
      cfg.AppSettings.Settings.Add("TagPassword", pwd) ;
      cfg.AppSettings.SectionInformation.ProtectSection(
                          "RsaProtectedConfigurationProvider" );
      cfg.Save() ;
   }
   static string LoadPwd() {
      Configuration cfg = ConfigurationManager.OpenExeConfiguration(
                          ConfigurationUserLevel.None) ;
      return cfg.AppSettings.Settings["TagPassword"].Value ;
   }
}
```

Voici le fichier de configuration généré par l'exemple précèdent. On voit qu'il contient une section `protectedData` qui déclare que la section `appSettings` est encryptée et une section `appSetings` qui contient les données encryptées :

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <protectedData>
      <protectedDataSections>
         <add name="appSettings"
              provider="RsaProtectedConfigurationProvider"
              inheritedByChildren="false" />
      </protectedDataSections>
   </protectedData>
   <appSettings>
      <EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element"
          xmlns="http://www.w3.org/2001/04/xmlenc#">
         <EncryptionMethod
          Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc" />
         <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
```

```
            ...
               <CipherData>
                  <KeyName>Rsa Key</KeyName>
                  <CipherValue>EuZpuz7Rj(...)qalkG3VbQ=</CipherValue>
               </CipherData>
            ...
         </KeyInfo>
           <CipherData>
              <CipherValue>5OcJLWVB(...)JHsGvLTY</CipherValue>
           </CipherData>
       </EncryptedData>
    </appSettings>
</configuration>
```

## *Protection des données échangées par l'intermédiaire d'un réseau*

Dans le chapitre consacré au flot de données, nous présentons en page 660 les différents protocoles d'échange sécurisé de données par l'intermédiaire d'un réseau.

# *Authentifier vos assemblages avec la technologie Authenticode et les certificats X.509*

## *Authenticode vs. Nom fort*

Bien avant la distribution publique de la plateforme .NET, les systèmes *Windows* exploitaient la technologie connue sous le nom d'*Authenticode* pour authentifier les fichier exécutables. Cette technologie s'apparente à la technologie du nom fort qui fait l'objet de la section page 28 dans le sens où elle permet de signer numériquement un assemblage. Cependant ces deux technologies ont des buts différents :

- La technologie Authenticode doit être utilisée pour identifier l'auteur ou l'entreprise qui a fabriqué le fichier exécutable. Elle permet d'exécuter un programme sans crainte d'une éventuelle falsification.
- La technologie du nom fort doit être utilisée pour identifier un assemblage (i.e pour nommer d'une manière unique chaque assemblage et même, chaque version d'un même assemblage). Elle permet notamment le stockage d'assemblages côte à côte.

En outre ces deux technologies présentent des différences notables :

- La technologie Authenticode permet de signer toutes sortes de fichiers exécutables, les assemblages .NET comme les fichiers exécutable *Windows* contenant du code natif. La technologie du nom fort ne se cantonne quant à elle qu'aux assemblages.
- La vérification de la validité d'un fichier exécutable par la technologie Authenticode n'est effectuée par *Windows* qu'une seule fois, lorsque l'on télécharge le fichier ou lorsqu'on l'installe (par exemple à partir d'un CD). La vérification du nom fort est effectuée par le CLR à chaque chargement d'un assemblage signé.

Cependant, la différence principale entre ces deux technologies réside dans le concept de certificat.

## *Les certificats et les Certificate Authorities*

Un *certificat* contient une clé publique et des informations sur un éditeur. Il permet de garantir que la clé publique a bien été fabriquée par l'éditeur référencé. Un certificat contient aussi d'autres informations telles que la date limite de validité. La technologie du nom fort ne présente pas de certificat. Bien que certaines clés publiques soient bien connues et permettent donc d'identifier un éditeur (comme la clé publique de *Microsoft* par exemple) il n'y a pas de moyens standard pour obtenir la certitude qu'une clé publique utilisée dans la technologie du nom fort a été fabriquée par un éditeur lambda.

Plusieurs standards de certificats existent. La technologie Authenticode utilise le standard *X.509* qui stocke les certificats dans des fichiers d'extension `.cer`. Le standard *CMS/Pkcs7* qui stocke les certificats dans des fichiers d'extension `.p7b` est aussi très répandu car il est utilisé par le protocole *Secure/Multipurpose Internet Mail Extensions* (*S/MIME*) pour signer et crypter les mails.

Un fichier signé avec la technologie Authenticode contient un certificat X.509. *Windows* peut alors demander à une autorité extérieure si le certificat est valide. Ces autorités extérieures sont nommées *Certificate Authorities* (*CA*) ou parfois aussi *tiers de confiance*. Ce sont des entreprises dont le but est de valider ou non une certification. On compte parmi les CA les plus connus les organismes *VeriSign* ou *EnTrust*. Concrètement, un éditeur de logiciel paie un CA pour que ce dernier valide son certificat auprès de ses clients. Le CA aura au préalable effectué un ensemble de vérification sur le sérieux de l'éditeur de logiciel.

## *Les certificats racines*

En pratique, il existe un mécanisme permettant à *Windows* de ne pas contacter systématiquement un CA lors de la vérification d'un certificat. En effet, chaque système *Windows* maintient une liste de certificats (nommée *Certificate Store* en anglais). Les certificats contenus dans cette liste sont nommés les *certificats racines*. La liste des certificats racines présentée par défaut par *Windows* est disponible dans l'article **Microsoft Root Certificate Program Members** des **MSDN**.

Vous pouvez visualiser ajouter ou retirer des certificats dans cette liste avec l'outil `certmgr.exe`. Lorsqu'un éditeur de logiciel demande à un CA de publier son certificat, ce dernier signe le certificat de l'éditeur avec son propre certificat. Si le certificat du CA est un certificat racine, *Windows* n'a pas besoin d'effectuer un accès réseau pour demander au CA de valider le certificat de l'éditeur puisqu'il a mathématiquement tous les éléments nécessaires pour effectuer la vérification lui même.

On peut imaginer que le CA contacté par l'éditeur est de moindre importance. Dans ce cas, le certificat de ce CA n'est probablement pas un certificat racine des systèmes *Windows*. Cependant, rien n'empêche à ce CA de moindre importance de demander à un CA bien connu de lui signer son certificat. On voit donc s'amorcer une chaîne de certifications qui en pratique, évite aux centaines de millions de machines utilisant *Windows* de contacter les CA lors de l'installation de logiciels.

## *Les implications de la technologie Authenticode*

Lorsque *Windows* installe ou télécharge un logiciel qui n'a pas de certificat ou dont le certificat ne peut être vérifié, il affiche une fenêtre de dialogue vous demandant si vous autorisez l'exécution du programme.

Le mécanisme CAS est capable d'obtenir une preuve à partir d'un assemblage signé avec un certificat X.509 (voir page 189). Vous pouvez ainsi décider que les assemblages signés avec un certificat peuvent s'exécuter avec plus (ou moins) de permissions que les autres.

Les détails de la création d'un certificat, de la signature d'un exécutable par un certificat et de la vérification d'un certificat dépassent le cadre du présent ouvrage. Tout ceci est décrit dans l'article **Signing and Checking Code with Authenticode** des **MSDN**.

En outre, sachez que le framework .NET présente les espaces de noms `System.Security.Cryptography.X509Certificates` et `System.Security.Cryptography.Pkcs` qui contiennent des types spécialisés dans la manipulation de ces standards ainsi que dans la manipulation des listes de certificats.

# 7

# *Réflexion, liens tardifs, attributs*

Nous avons parlé page 17 des métadonnées et de la façon dont elles sont physiquement stockées dans les assemblages. Nous allons voir dans ce chapitre qu'elles constituent la base des mécanismes de réflexion et d'attributs.

## *Le mécanisme de réflexion*

Le mécanisme de *réflexion* dénomme l'utilisation durant l'exécution, des métadonnées de type d'un assemblage. En général cet assemblage est chargé explicitement lors de l'exécution d'un autre assemblage mais il peut être aussi construit puis chargé dynamiquement.

Le mot réflexion est utilisé pour montrer que l'on utilise l'image d'un assemblage (comme une image dans un miroir). Cette image est constituée par les métadonnées de type de l'assemblage.

### *Quand utilise-t-on le mécanisme de réflexion ?*

Nous avons recensé quelques catégories d'utilisation du mécanisme de réflexion. Elles font l'objet des prochaines sections du présent chapitre. Le mécanisme de réflexion peut être utilisé dans les cas suivants :

- Lors de la découverte des types d'un assemblage à l'exécution par l'analyse dynamique de ses métadonnées de type. Par exemple, les outils `ildasm.exe` ou `Reflector` chargent explicitement un module d'un assemblage et analyse son contenu (voir page 21).
- Lors de l'utilisation de *liens tardifs*. Cette technique consiste à utiliser une classe située dans un assemblage qui n'est pas connu à la compilation. La technique du lien tardif est particulièrement utilisée par les langages interprétés comme les langages de script.
- Lorsque l'on souhaite exploiter les informations contenues dans les attributs.

- Lorsque l'on souhaite accéder aux membres non publics d'une classe à partir de l'extérieur de la classe. Bien évidemment cette pratique est à éviter, mais il est nécessaire parfois d'y avoir recours, par exemple pour faire des tests unitaires qui ne peuvent se satisfaire des seuls membres non publics.
- Lors de la construction dynamique d'un assemblage. Pour utiliser les classes d'un assemblage construit dynamiquement on utilise la technique du lien tardif explicite.

Le mécanisme de réflexion est utilisé par le CLR et le framework dans de multiples cas. Par exemple, l'implémentation par défaut de la méthode `Equals()` sur un type valeur utilise la réflexion pour comparer deux instances champs à champs.

La réflexion est également utilisée par le CLR lors de la sérialisation d'un objet afin de connaître les champs à sérialiser, ou encore par le ramasse-miettes, qui s'en sert pour construire l'arbre de référencement lors d'une collecte d'objets.

## *Qu'est ce que le mécanisme de réflexion apporte de nouveau ?*

L'idée sous-jacente du mécanisme de réflexion n'est pas nouvelle. Cela fait longtemps que l'on peut analyser dynamiquement le contenu d'un exécutable, notamment grâce à des informations d'auto description. Le format TLB (présenté page 279) a été conçu à cet effet. Précisons que les donnés au format TLB sont produites à partir de données au format IDL (Interface Definition Language). Le langage IDL peut donc aussi être vu comme un format d'auto description. Le mécanisme de réflexion proposé par .NET va beaucoup plus loin que le format TLB et le format IDL :

- Il est très simple à utiliser grâce à certaines classes de base.
- Il est plus abstrait que le format TLB et le langage IDL. Par exemple il n'utilise pas d'adresses physiques. Ainsi il peut être utilisé à la fois sur des machines 32 et 64 bits.
- Contrairement aux métadonnées du format TLB, les métadonnées .NET sont toujours physiquement contenues dans le module qu'elles décrivent.
- Le niveau de détail de description des données est beaucoup plus poussé que dans le format TLB. Concrètement on peut très simplement avoir toutes les informations possibles sur n'importe quel élément déclaré dans un assemblage (par exemple le type d'un argument d'une méthode d'une classe).

Le niveau de détail sans précédent de la réflexion .NET est dû à de nombreuses classes de base du Framework .NET qui permettent d'extraire et d'utiliser les métadonnées de type d'un assemblage contenu dans un domaine d'application. La plupart de ses classes se trouvent dans l'espace de noms `System.Reflection`. Il y a une classe pour chaque type d'élément d'un assemblage :

- Il y a une classe dont les instances représentent des assemblages (`System.Reflection.Assembly`) ;
- une classe dont les instances représentent des classes (`System.Type`) ;
- une classe dont les instances représentent les méthodes des classes (`System.Reflection.MethodInfo`) ;
- une classe dont les instances représentent les champs des classes (`System.Reflection.FieldInfo`) ;

- une classe dont les instances représentent les paramètres des méthodes (`System.Reflection.ParameterInfo`).
- etc.

Finalement ces classes ne représentent rien d'autre qu'un moyen de visualiser logiquement l'ensemble des métadonnées de type. La visualisation n'est pas physique dans le sens où certains éléments utilisés pour l'organisation interne d'un assemblage (comme les jetons de métadonnées) ne sont pas représentés.

Toutes les classes de l'espace de noms `System.Reflection` s'utilisent d'une manière très logique. Par exemple, à partir d'une instance de `System.Reflection.Assembly`, on peut obtenir un tableau d'instances de `System.Type`. À partir d'une instance de `System.Type` on peut obtenir un tableau d'instances de `System.Reflection.MethodInfo`. À partir d'une instance de `System.Reflection.MethodInfo`, on peut obtenir un tableau d'instances de `System.Reflection.ParameterInfo`. Tout ceci est illustré par la Figure 7-1.

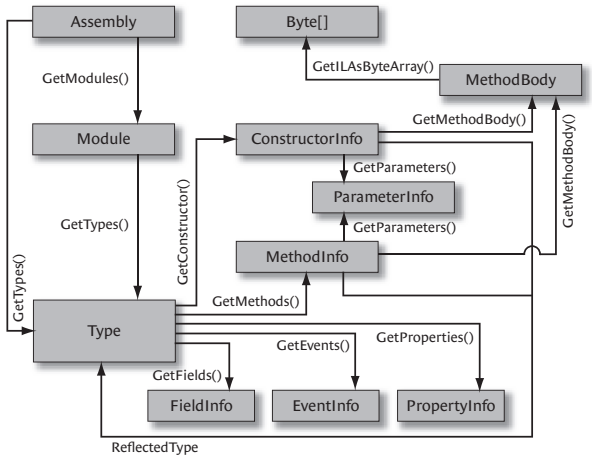


*Figure 7-1 : Classes utilisées par le mécanisme de réflexion*

On peut regretter le fait que l'on ne puisse pas descendre au niveau de l'instruction IL mais seulement au niveau de d'un tableau d'octets représentant le corps d'une méthode en IL. Pour cela, vous devez avoir recours à certaines bibliothèques telles que *Cecil* (développée par *Jean-Baptiste Evain*), *ILReader* (développée par *Lutz Roeder*) ou *Rail* (développée par l'université de *Coimbra*). Sachez cependant qu'en .NET 2005 la réflexion sait traiter les types génériques, les méthodes génériques ainsi que les contraintes (voir page 498).

## *Analyser les assemblages contenus dans un domaine d'application*

Voici un exemple qui expose l'analyse des métadonnées de type avec les classes de l'espace de noms `System.Reflection`. En fait, nous présentons ici un assemblage qui analyse ses propres métadonnées de type. Pour obtenir une instance de la classe `Assembly` représentant cet assemblage, nous avons recours à la méthode statique `Assembly.GetExecutingAssembly()` :

*Exemple 7-1 :*

```
using System ;
using System.Reflection ;
class Program{
   public static void Main(){
      Assembly assembly = Assembly.GetExecutingAssembly() ;
      foreach (Type type in assembly.GetTypes()){
         Console.WriteLine("Classe : " + type) ;
         foreach (MethodInfo method in type.GetMethods()){
            Console.WriteLine("  Méthode : "  + method) ;
            foreach (ParameterInfo param in method.GetParameters())
               Console.WriteLine("      Param : " +
                                 param.GetType()) ;
         }
      }
   }
}
```

Voici l'affichage de ce programme :

```
Classe : Program
  Méthode : Void Main()
  Méthode : System.Type GetType()
  Méthode : System.String ToString()
  Méthode : Boolean Equals(System.Object)
      Param : System.Reflection.ParameterInfo
  Méthode : Int32 GetHashCode()
```

## *Extraire des informations à partir des métadonnées*

L'objet de la présente section est de présenter un petit programme qui exploite le mécanisme de réflexion pour afficher l'ensemble des classes d'exceptions contenues dans les assemblages `System` et `mscorlib`. Nous nous sommes basés sur le fait que toutes les classes d'exception dérivent de la classe `System.Exception`. Les types d'exceptions qui ne dérivent pas directement de la classe `System.Exception` sont marqués avec une apostrophe.

On pourrait simplement modifier ce programme pour afficher l'ensemble des classes d'attribut du Framework en se basant sur le fait que toutes ces classes dérivent de la classe `System.Attribute`.

*Exemple 7-2 :*

```
using System ;
using System.Reflection ;
class Program {
   static void Main() {
      // Fabrique le nom fort de l'assemblage 'System'.
      // Le numéro de version est le même pour tous les asm systèmes.
      // C'est celui de 'mscorlib' qui contient System.Object.
      string systemAsmStrongName = "System, Culture = neutral, " +
```

```
         "PublicKeyToken=b77a5c561934e089, Version=" +
         typeof(System.Object).Assembly.GetName().Version.ToString() ;

      // Charge explicitement l'assemblage 'System'.
      // Nul besoin de charger l'assemblage 'mscorlib' car il est
      // automatiquement et implicitement chargé par le CLR.
      Assembly.ReflectionOnlyLoad(systemAsmStrongName) ;

      // Pour chaque assemblage du domaine d'application courant...
      foreach (Assembly a in AppDomain.CurrentDomain.GetAssemblies()){
         Console.WriteLine("\nAssemblage:" + a.GetName().Name) ;

         // Pour chaque type de cet assemblage...
         foreach (Type t in a.GetTypes()){
            // Seules les classes publiques sont retenues...
            if (!t.IsClass || !t.IsPublic) continue ;
            bool bDeriveDException = false ;
            bool bDeriveDirectement = true ;

            // System.Exception est-il un type de base de ce type ?
            Type baseType = t.BaseType ;
            while (baseType != null && !bDeriveDException){
               // Pour trouver les classes d'attributs remplacer cette
               // ligne par if( baseType == typeof(System.Attribute))
               if (baseType == typeof(System.Exception))
                  bDeriveDException = true ;
               else bDeriveDirectement = false ;
               baseType = baseType.BaseType ;
            }// end while

            // Affiche le nom de la classe si elle représente un
            // attribut. Place une apostrophe si la classe ne
            // dérive pas directement de System.Excception.
            if (bDeriveDException)
               if (bDeriveDirectement)
                  Console.WriteLine("  " + t) ;
               else
                  Console.WriteLine("  '" + t) ;
         }// end foreach(Type...
      }// end foreach(Assembly...
   }// end Main
}
```

Notez l'utilisation de la méthode `Assembly.ReflectionOnlyLoad()` dans l'exemple précédent. Cette méthode permet de spécifier au CLR que l'assemblage chargé ne sera utilisé que par le mécanisme de réflexion. En conséquence, le CLR interdira l'exécution du code d'un assemblage chargé avec cette méthode. En outre, le chargement d'un assemblage en mode *Reflection Only* est sensiblement plus rapide car le CLR s'affranchit des vérifications relatives à la sécurité. La

classe `Assembly` présente la propriété `bool ReflectionOnly{get;}` qui permet de savoir qu'un assemblage a été chargé par cette méthode.

# *Les liens tardifs*

Avant d'aborder cette section, il est préférable d'avoir de bonnes bases en programmation objet, notamment en ce qui concerne le mécanisme de polymorphisme. Ce sujet est traité dans le chapitre 12.

## *Que veut dire « créer un lien avec une classe » ?*

Tout d'abord, il faut s'entendre sur ce que veut dire « créer un lien avec une classe » (*to bind a class* en anglais). On va parler de couches logicielles plutôt que d'assemblages car ce qui va être dit est valable pour d'autres technologies.

Un lien avec une classe est créé entre la couche logicielle utilisatrice de la classe (qui instancie la classe puis utilise les instances) et la couche logicielle qui définit la classe. Concrètement, ce lien est la correspondance entre les appels aux méthodes de la classe dans la couche logicielle utilisatrice, et l'emplacement physique des méthodes dans la couche logicielle qui définit la classe. C'est ce lien qui permet au thread d'aller continuer sa course dans le code de la méthode d'une classe lorsque celle-ci est appelée.

En général, on distingue trois types de liens avec une classe : les *liens précoces* (*early bind* en anglais) créés pendant la compilation, les *liens dynamiques* (*dynamic bind* en anglais) dont une partie est créée à la compilation et l'autre à l'exécution, et les *liens tardifs* (*late binding* en anglais) créés pendant l'exécution.

## *Liens précoces et liens dynamiques*

Les liens précoces sont créés par le compilateur, lors de la fabrication d'un assemblage à partir du code source écrit dans un langage .NET. On ne peut donc pas créer de liens précoces avec une méthode virtuelle ou abstraite. En effet, le mécanisme de *polymorphisme* décide à l'exécution quel est le corps d'une méthode virtuelle ou abstraite à appeler, à partir de la classe réelle de l'objet sur lequel la méthode est appelée. Dans ce cas, on qualifie le lien de *lien dynamique*. Dans la première édition du présent ouvrage ainsi que dans d'autres documents, les liens dynamiques sont parfois nommés *lien tardif implicite* puisqu'ils sont créés implicitement par le mécanisme de polymorphisme et puisqu'ils sont finalisés tardivement, durant l'exécution.

Intéressons-nous ici aux liens précoces, ceux créés avec des méthodes statiques ou avec des méthodes d'instances ni virtuelles ni abstraites. Si l'on s'en tient à la définition de « *créer un lien avec une classe* » de la section précédente, il n'existe pas de liens précoces en .NET. En effet, il faut attendre que le corps d'une méthode soit « JIT compilée » en langage machine avant de connaître son emplacement physique dans l'espace d'adressage du processus. Cette information d'emplacement physique du corps de la méthode ne peut donc pas être connue du compilateur qui crée l'assemblage. Nous avons vu page 111 que pour résoudre ce problème, le compilateur qui crée l'assemblage insère le jeton de métadonnées correspondant à la méthode à appeler, à l'endroit du code IL où l'appel a lieu. Lorsque le corps de la méthode est « JIT compilée » à l'exécution, le CLR stocke en interne la correspondance entre la méthode et l'emplacement physique

du corps de la méthode en langage machine. Cette information est physiquement stockée dans une zone mémoire associée à la méthode, appelée *stub*.

Cette constatation est très importante puisque dans un langage comme C++, lorsqu'une méthode n'est ni virtuelle ni abstraite (i.e ni virtuelle pure en terminologie C++), le compilateur calcule l'emplacement physique du corps de la méthode en langage machine. Ensuite, le compilateur insère un pointeur vers cet emplacement à chaque appel de la méthode concernée. Cette différence confère un avantage certains à .NET, puisque les compilateurs n'ont plus à se soucier de détails techniques tels que la représentation d'une adresse mémoire. Le code IL est totalement indépendant de la couche physique qui l'exécute.

En ce qui concerne les liens dynamiques, presque tout se passe comme pour les liens précoces. Le compilateur insère le jeton de métadonnée correspondant à la méthode virtuelle (ou abstraite) à appeler, à l'endroit du code IL où l'appel a lieu. On parle ici du jeton de métadonnée de la méthode définie dans le type de la référence sur laquelle l'appel à lieu. C'est ensuite au CLR de déterminer à l'exécution vers quelle méthode se brancher en fonction de l'implémentation exacte de l'objet référencé. C'est le polymorphisme.

Cette technique d'insertion par le compilateur du jeton de métadonnée pour les liens précoces et dynamiques est utilisée dans les trois cas suivant :

- Lorsque le code contenu dans un module appelle une méthode définie dans le même module.
- Lorsque le code contenu dans un module appelle une méthode définie dans un autre module du même assemblage.
- Lorsque le code contenu dans un module d'un assemblage appelle une méthode définie dans un autre assemblage référencé à la compilation. Si ce dernier assemblage n'est pas déjà chargé lors de l'appel de la méthode, le CLR le charge implicitement.

## *Liens tardifs*

En .NET, le code d'un assemblage A peut instancier et utiliser à l'exécution un type défini dans un assemblage B non référencé à la compilation de A. On qualifie ce type de lien *lien tardif*. Dans la première édition du présent ouvrage ainsi que dans d'autres documents, les liens tardifs sont parfois nommés *liens tardifs explicites*. Le lien est tardif dans le sens où, comme nous allons le voir, il est créé après la compilation, durant l'exécution. Le lien est explicite dans le sens où le nom de la méthode à appeler doit être précisé explicitement dans une chaîne de caractères.

L'idée de lien tardif n'est pas nouvelle dans le monde du développement *Microsoft*. Tout le mécanisme dit d'*Automation* dans la technologie COM, qui utilise l'interface `IDispatch`, n'est rien d'autre qu'une « usine à gaz » faite pour pouvoir créer des liens tardifs à partir de langages de scripts ou peu typés, comme VB. La notion de liens tardifs existe aussi en Java.

Le concept de lien tardif fait partie des idées que les développeurs habitués au langage C++ ont du mal à assimiler. En effet, en C++ seul les liens précoces et les liens dynamiques existent. Le problème de compréhension vient du fait suivant : on comprend bien que les informations nécessaires pour créer un lien (i.e les jetons de métadonnées) sont dans l'assemblage B contenant la classe à appeler, mais on ne comprend pas pourquoi le développeur ne profite pas de la capacité du compilateur à créer des liens précoces et dynamiques en référençant B à la compilation de A. Il existe plusieurs raisons différentes :

- La raison la plus courante est que pour certains langages il n'y a pas de compilateur ! Dans un langage type script, les instructions sont interprétées une à une. Dans ce cas il ne peut y avoir que des liens tardifs. Les liens tardifs permettent donc d'utiliser des classes compilées et contenues dans des assemblages, à partir de langages interprétés. Le fait que la technique des liens tardifs soit très facile à utiliser en .NET, fait que l'on peut facilement créer des langages interprétés propriétaires (tel que le langage *IronPython* `http://www.ironpython.com/`).
- On peut souhaiter utiliser la technique de liens tardifs à partir de programmes écrits en langages compilés, tels que C#. L'idée est que l'emploi d'un lien tardif permet d'introduire un degré de flexibilité dans l'architecture générale de votre application. Cette technique est en fait un design pattern populaire nommé *plugin* que nous présenterons à la fin de la présente section.
- Certaines applications ont pour vocation d'appeler le code de n'importe quel assemblage qui lui est proposé. L'exemple typique est l'outil *open-source* `NUnit` qui permet de tester le code de n'importe quel assemblage en invoquant ses méthodes. Nous approfondissons ce sujet lors de la construction d'un attribut personnalisé un peu plus loin dans ce chapitre.
- On doit utiliser des liens tardifs entre le code d'un assemblage A et les classes d'un assemblage B si B n'existe pas lors de la compilation de A. Cette situation est décrite un peu plus loin dans ce chapitre où l'on y parle de construction dynamique d'assemblages.

Certains préconisent l'utilisation de liens tardifs à la place du polymorphisme. En effet, puisque au moment de l'appel seul le nom et la signature de la méthode sont pris en compte, le type de l'objet sur lequel la méthode est appelée importe peu. Il suffit que l'implémentation de l'objet présente une méthode avec ce nom et cette signature. Nous déconseillons néanmoins cette utilisation car elle est très permissive et ne force pas les concepteurs de l'application à se focaliser sur des interfaces abstraites.

À part les cas qui viennent d'être exposés, vous n'aurez pas à utiliser de liens tardifs explicites. Ne les utiliser pas pour le plaisir d'avoir des liens tardifs explicites dans vos applications car :

- Vous perdrez l'énorme bienfait de la vérification syntaxique du compilateur.
- Les performances des liens tardifs explicites sont beaucoup moins bonnes que celles des liens précoces et des liens tardifs implicites (même si vous utilisez les optimisations présentées un peu plus loin).
- On ne peut créer un lien tardif avec une classe obfusquée. En effet, lors de l'obfuscation, le nom de la classe est changé dans l'assemblage qui la contient. Or, la technique du lien tardif retrouve la classe avec laquelle le lien tardif doit être créé grâce à son nom. Comme nous allons le voir, ce nom est stocké du coté du consommateur de la classe dans une chaîne de caractères. Il y a donc incompatibilité entre obfuscation et lien tardif.

## *Création d'une instance d'une classe non connue à la compilation*

Si une classe ou une structure n'est pas connue à la compilation, on ne peut l'instancier avec l'opérateur `new`. Le *framework* .NET prévoit donc des classes qui permettent d'instancier des types non connus à la compilation.

### *Préciser un type*

Intéressons-nous aux différentes techniques qui permettent de préciser un type :

- Certaines méthodes de certaines classes acceptent une chaîne de caractères qui contient le nom complet du type (i.e avec son espace de noms).
- D'autres acceptent une instance de la classe `System.Type`. Dans un domaine d'application, chaque instance de `System.Type` représente un type et il ne peut y avoir deux instances représentant le même type. Pour obtenir une instance de `System.Type`, il existe de nombreuses façons :
  - En C# on utilise souvent le mot-clé `typeof()` qui prend en paramètre un type, et retourne l'instance de `System.Type` adéquate.
  - Vous pouvez aussi utiliser une des surcharges de la méthode statique `GetType()` de la classe `System.Type`.
  - Si un type est encapsulé dans un autre type, vous pouvez utiliser une des surcharges des méthodes non statiques `GetNestedType()` ou `GetNestedTypes()` de la classe `System.Type`.
  - Vous pouvez aussi utiliser les méthodes non statiques `GetType()` `GetTypes()` ou `GetExportedTypes()` de la classe `System.Reflection.Assembly`.
  - Vous pouvez aussi utiliser les méthodes non statiques `GetType()` `GetTypes()` ou `FindTypes()` de la classe `System.Relection.Module`.

Supposons maintenant que le programme suivant soit compilé dans un assemblage `Foo.dll`. Nous allons exposer plusieurs manières permettant de créer une instance de la classe `NMFoo.Calc` à partir d'assemblages qui ne référencent pas `Foo.dll`.

*Exemple 7-3 : Code de l'assemblage Foo.dll*

```
using System ;
namespace NMFoo {
   public class Calc {
      public Calc() {
         Console.WriteLine("Calc.Constructeur appelé") ;
      }
      public int Sum(int a, int b) {
         Console.WriteLine("Méthode Calc.Sum() appelée") ;
         return a + b ;
      }
   }
}
```

## *Utilisation de la classe System.Activator*

La classe `System.Activator` présente les deux méthodes statiques `CreateInstance()` et `CreateInstanceFrom()` qui permettent de créer une instance d'une classe non connue à la compilation. Par exemple :

*Exemple 7-4 :*

```
using System ;
using System.Reflection ;
class Program {
   static void Main() {
```

```
      Assembly assembly = Assembly.Load("Foo.dll") ;
      Type type = assembly.GetType("NMFoo.Calc") ;
      object obj = Activator.CreateInstance(type);
      // 'obj' est une référence vers une instance de NMFoo.Calc.
   }
}
```

Chacune de ces méthodes se décline en de nombreuses versions surchargées (et parfois génériques) avec les arguments suivants :

- Une classe, sous forme d'une chaîne de caractères ou d'une instance de `System.Type` ;
- éventuellement le nom de assemblage qui contient la classe ;
- éventuellement une liste d'arguments pour le constructeur.

Si l'assemblage qui contient la classe n'est pas présent dans le domaine d'application, l'appel à `CreateInstance()` ou `CreateInstanceFrom()` déclenche le chargement de cet assemblage. Pour réaliser ce chargement, une des méthodes `System.AppDomain.Load()` ou `System.AppDomain.LoadFrom()` est appelée en interne, selon que l'on appelle `CreateInstance()` ou `CreateInstanceFrom()`. Un constructeur de la classe est choisi, à partir des arguments passés. Une instance de la classe `ObjectHandle`, renfermant un objet marshallé par valeur est retournée. Dans le chapitre 22 relatif à .NET Remoting, nous présentons une autre utilisation de ces méthodes dans le cadre d'applications distribuées.

Les surcharges de `CreateInstance()` où le type est précisé sous la forme d'une instance de `System.Type`, retournent directement une référence vers un objet.

La classe `System.Activator` présente aussi la méthode `CreateComInstanceFrom()` utilisée pour créer des instances d'objets COM et la méthode `GetObject()` utilisée pour créer des objets distants.

## *Utilisation de la classe System.AppDomain*

La classe `System.AppDomain` présente les quatre méthodes non statiques `CreateInstance()`, `CreateInstanceAndUnWrap()`, `CreateInstanceFrom()` et `CreateInstanceFromAndUnwrap()` qui permettent de créer une instance d'une classe non connue à la compilation. Par exemple :

*Exemple 7-5 :*

```
using System ;
using System.Reflection ;
class Program {
   static void Main() {
      object obj = AppDomain.CurrentDomain.CreateInstanceAndUnwrap(
                                    "Foo.dll", "NMFoo.Calc");
      // 'obj' est une référence vers une instance de NMFoo.Calc.
   }
}
```

Ces méthodes sont similaires aux méthodes de la classe `System.Activator`, présentées précédemment. Cependant, elles permettent de choisir le domaine d'application dans lequel l'objet doit être créé. De plus, les versions « `AndUnwarp()` » fournissent directement une référence vers l'objet. Cette référence est obtenue à partir de l'instance de `ObjectHandle`.

## *Utilisation de la classe System.Reflection.ConstructorInfo*

Une instance de la classe `System.Reflection.ConstructorInfo` référence un constructeur. La méthode `Invoke()` de cette classe construit en interne un lien tardif vers le constructeur, puis l'invoque au moyen de ce lien. Elle permet donc de créer une instance du type du constructeur. Par exemple :

*Exemple 7-6 :*

```
using System ;
using System.Reflection ;
class Program {
   static void Main() {
      Assembly assembly = Assembly.Load ("Foo.dll") ;
      Type type = assembly.GetType("NMFoo.Calc") ;
      ConstructorInfo constructorInfo =
                           type.GetConstructor( new Type[0] ) ;
      object obj = constructorInfo.Invoke(new object[0]) ;
      // 'obj' est une référence vers une instance de NMFoo.Calc.
   }
}
```

## *Utilisation de la classe System.Type*

La méthode non statique `InvokeMember()` de la classe `System.Type` permet de créer une instance d'une classe non connue à la compilation, à la condition qu'elle soit appelée avec la valeur `CreateInstance` de l'énumération `BindingFlags`. Par exemple :

*Exemple 7-7 :*

```
using System ;
using System.Reflection ;
class Program {
   static void Main() {
      Assembly assembly = Assembly.Load("Foo.dll") ;
      Type type = assembly.GetType("NMFoo.Calc") ;
      Object obj = type.InvokeMember(
         null, // pas besoin de nom pour invoquer le constructeur
         BindingFlags.CreateInstance,
         null, // pas besoin de binder
         null, // pas besoin d'un objet cible car on le crée
         new Object[0]) ; // pas de paramètre
      // 'obj' est une référence vers une instance de NMFoo.Calc.
   }
}
```

## *Cas particuliers*

Avec les méthodes présentées, vous pouvez créer une instance de pratiquement n'importe quelle classe ou structure. Deux cas particuliers sont à noter :

- Pour créer un tableau il faut appeler la méthode statique `CreateInstance()` de la classe `System.Array`.
- Pour créer un délégué il faut appeler la méthode statique `CreateDelegate()` de la classe `System.Delegate`.

## *Utiliser les liens tardifs avec le framework*

Maintenant que nous savons créer des instances de types inconnus à la compilation, intéressons-nous à la création de liens tardifs vers les membres de ces types afin de pouvoir utiliser ces instances. Là aussi, plusieurs façons existent.

### *La méthode Type.InvokeMember()*

Revenons vers la méthode `Type.InvokeMember()` qui nous a servi précédemment à créer une instance d'un type inconnu à la compilation en invoquant un de ces constructeurs. Cette méthode accomplit en interne trois tâches :

- Elle cherche un membre du type sur lequel elle est appelée qui correspond aux informations qu'on lui passe.
- Si le membre est trouvé, elle crée un lien tardif.
- Elle utilise le membre (invocation pour une méthode, création d'objet puis invocation pour un constructeur, obtention ou définition de la valeur pour un champ...).

L'exemple suivant montre comment invoquer la méthode `Sum()` sur notre instance de la classe `NMFoo.Calc` (notez que lors de la phase de déboguage, le débogueur est capable de continuer sa course dans le corps de la méthode invoquer grâce à un lien tardif) :

*Exemple 7-8 :*

```
using System ;
using System.Reflection ;
class Program {
   static void Main() {
      object obj = AppDomain.CurrentDomain.CreateInstanceAndUnwrap(
                                 "Foo.dll", "NMFoo.Calc") ;
      Type type = obj.GetType() ;
      object[] parametres = new object[2] ;
      parametres[0] = 7 ;
      parametres[1] = 8 ;
      int result = (int)type.InvokeMember(
                              "Sum",       // Nom de la méthode.
                              BindingFlags.InvokeMethod,
                              null,        // Pas besoin de binder.
                              obj,         // L'objet cible.
                              parametres); // Les paramètres.
      // Ici, result vaut 15.
   }
}
```

La surcharge la plus couramment utilisée de la méthode `Type.InvokeMember()` est :

```
public object InvokeMember(
   string       name,       // Le nom du membre.
   BindingFlags invokeAttr, // Quel membre doit être pris en compte.
   Binder       binder,     // Les règles de recherche du membre.
   object       target,     // L'objet sur lequel invoquer le membre.
   object[]     args        // Les arguments de l'appel.
) ;
```

Le paramètre `invokeAttr` est un indicateur binaire qui signale sur quel type de membre la recherche va se porter. Pour chercher sur les méthodes, nous utilisons l'indicateur `BindingFlags.InvokeMethod`. Les différents indicateurs sont décrits en détail dans les **MSDN**, dans l'article **BindingFlags Enumeration**.

Le paramètre `binder` indique un objet de type `Binder` qui va orienter `InvokeMember()` sur la façon dont elle va effectuer ses recherches. La plupart du temps vous positionnerez ce paramètre à `null` pour indiquer que vous souhaitez utiliser la propriété `System.Type.DefaultBinder`. Un objet de type `Binder` fournit ce genre d'informations :

- Il indique quelles conversions de types sont acceptées ou non pour les arguments. Dans l'exemple précédent, nous aurions pu fournir deux arguments de type `double`. Grâce au `DefaultBinder`, l'appel de la méthode aurait marché car il supporte la conversion du type `double` vers le type `int`.
- Il indique si nous utilisons les paramètres optionnels dans notre liste de paramètres.

Tout ceci (notamment la table de conversion de type) est décrit en détail dans les **MSDN** à l'entrée **Type.DefaultBinder Property**. Vous pouvez créer vos propres objets `Binder` en dérivant de la classe `Binder`. Néanmoins une instance de `DefaultBinder` suffit dans la plupart des cas et cette possibilité est très rarement utilisée.

Si une exception est lancée durant l'appel du membre, `InvokeMember()` intercepte l'exception et relance une exception de type `System.Reflection.TargetInvocationException`. Naturellement l'exception lancée dans la méthode est référencée par la propriété `InnerException` de l'exception relancée.

Notez enfin que lorsque vous créez un lien tardif, vous ne pouvez pas accéder, *a priori*, aux membres non publics. L'exception `System.Security.SecurityException` est alors, en général, lancée. Néanmoins si le bit `TypeInformation` de `System.Security.Permissions.ReflectionPermissionFlags` (accessible par une instance de la classe `System.Security.Permissions.ReflectionPermission`) est positionné, vous avez accès aux membres non publics. Si le bit `MemberAccess` est positionné, vous avez accès aux types non visibles (i.e encapsulés dans d'autres types, d'une manière non publique).

## *Lier une fois, invoquer N fois*

De même que nous avons vu qu'une instance de la classe `ConstructorInfo` permet de se lier tardivement et d'invoquer un constructeur, une instance de la classe `System.Reflection.MethodInfo` permet de se lier tardivement et d'invoquer n'importe quelle méthode. L'avantage d'utiliser la classe `MethodInfo` plutôt que la méthode `Type.InvokeMember()` réside dans l'économie de la recherche du membre à chaque invocation, d'où une légère optimisation. L'exemple suivant illustre tout ceci :

*Exemple 7-9 :*

```
using System ;
using System.Reflection ;
class Program {
   static void Main() {
      object obj = AppDomain.CurrentDomain.CreateInstanceAndUnwrap(
                                   "Foo.dll", "NMFoo.Calc") ;
      Type type = obj.GetType() ;
      // Création du lien tardif.
      MethodInfo methodInfo = type.GetMethod("Sum") ;
      object[] parametres = new object[2] ;
      parametres[0] = 7 ;
      parametres[1] = 8 ;
      int result ;
      // 10 invocations de 'Sum' au travers du lien tardif.
      for (int i = 0 ; i < 10 ; i++)
         result = (int)methodInfo.Invoke(obj, parametres) ;
   }
}
```

## *Le compilateur VB.NET crée des liens tardifs*

En effet, faisons une petite parenthèse VB.NET et observons le fait que ce langage utilise « en secret » des liens tardifs lorsque l'option `Strict` est positionnée à `Off`. Par exemple le programme VB.NET suivant...

*Exemple 7-10 :* *VB.NET et les liens tardifs*

```
Option Strict Off
Module Module1
    Sub Main()
        Dim obj = AppDomain.CurrentDomain.CreateInstanceAndUnwrap(
                                  "Foo.dll", "NMFoo.Calc")
        Dim result As Integer = obj.Sum(7, 8)
    End Sub
End Module
```

...est équivalent au programme C# suivant :

*Exemple 7-11 :*

```
using System ;
using System.Reflection ;
using Microsoft.VisualBasic.CompilerServices;
class Program {
   static void Main() {
      object obj = AppDomain.CurrentDomain.CreateInstanceAndUnwrap(
                                   "Foo.dll", "NMFoo.Calc") ;
      object[] parametres = new object[2] ;
      parametres[0] = 7 ;
      parametres[1] = 8 ;
```

```
      int result = (int)LateBinding.LateGet(obj, null, "Sum",
                                            parametres,  null, null) ;
   }
}
```

Cette « facilité » proposer aux développeurs VB.NET peut vite se révéler catastrophique à la fois pour les performances et si le programme est obfusqué. D'où l'intérêt de positionner l'option `Strict` à `On`.

## *La bonne façon d'utiliser les liens tardifs avec des interfaces*

Il existe une autre manière totalement différente que celles que nous avons exposé pour exploiter une classe ou une structure non connue à la compilation avec des liens tardifs. Cette façon présente l'énorme avantage de ne pas dégrader les performances par rapport à l'utilisation d'un lien précoce ou dynamique. En revanche, vous devez vous contraindre à une certaine discipline pour appliquer cette « recette » (qui est en fait un *design pattern* connu sous le nom de *plugin*).

L'idée est qu'il faut faire en sorte que le type inconnu à la compilation implémente une interface qui elle, est connue à la compilation. Pour cela, nous sommes contraints de créer un troisième assemblage spécialement pour héberger cette interface. Réécrivons notre exemple avec la classe `Calc` au moyen de 3 assemblages :

*Exemple 7-12 : Code de l'assemblage contenant l'interface (InterfaceAsm.cs)*

```
namespace NMFoo {
   public interface ICalc {
      int Sum(int a, int b) ;
   }
}
```

*Exemple 7-13 : Code de l'assemblage contenant la classe cible (ClassAsm.cs)*

```
using System ;
namespace NMFoo {
   public class CalcAvecInterface : ICalc {
      public CalcAvecInterface() {
         Console.WriteLine("Calc.Constructeur appelé") ;
      }
      public int Sum(int a, int b) {
         Console.WriteLine("Méthode Calc.Sum() appelée") ;
         return a + b ;
      }
   }
}
```

*Exemple 7-14 : Code de l'assemblage client de la classe cible, non connue à la compilation (ProgramAsm.cs)*

```
using System ;
using System.Reflection ;
using NMFoo ;
class Program {
```

```
   static void Main() {
      ICalc obj = AppDomain.CurrentDomain.CreateInstanceAndUnwrap(
            "ClassAsm.dll", "NMFoo.CalcAvecInterface") as ICalc ;
      int result = obj.Sum(7, 8);
   }
}
```

Soyez attentif au transtypage explicite en `ICalc` de l'objet retourné par la méthode `CreateInstanceAndUnwrap()` qui nous permet d'utiliser ensuite un lien dynamique sur la méthode `Sum()`. On aurait pu éviter ce transtypage en utilisant la surcharge générique de la méthode `Activator.CreateInstance<ICalc>()`.

La figure suivante reprend l'organisation ainsi que les liens entre nos trois assemblages :

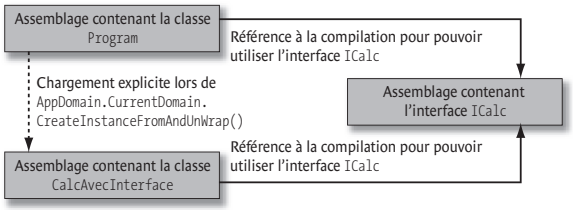


*Figure 7-2 : Organisation des assemblages dans le design pattern plugin*

Dans la philosophie du design pattern *plugin*, les données nécessaires à la méthode `CreateInstanceAndUnwrap()` pour créer une instance (en l'occurrence les deux chaînes de caractères `"Foo.dll"` et `"NMFoo.CalcAvecInterface"`) sont en général stockées dans un fichier de configuration. Tout l'intérêt est alors de pouvoir choisir une implémentation simplement en modifiant un fichier de configuration et sans aucune recompilation.

Une variante du *plugin* consiste à utiliser une classe abstraite plutôt qu'une interface.

Enfin, sachez qu'il est aussi possible d'utiliser un délégué pour pouvoir se lier tardivement vers une méthode. Bien que cette technique soit plus performante que l'utilisation de la classe `MethodInfo`, on lui préfère en général le *plugin*.

# *Les attributs*

## *Qu'est ce qu'un attribut ?*

Un *attribut* est une information qui se rapporte à un élément du code tel qu'une classe ou une méthode. Par exemple, le framework .NET fournit l'attribut `System.ObsoleteAttribute` qui peut être utilisé pour marquer une méthode comme ceci (notez la syntaxe avec les crochets `[]`) :

```
[System.ObsoleteAttribute()]
void Fct() { }
```

L'information « la méthode `Fct()` est marquée avec l'attribut `System.ObsoleteAttribute` » est insérée dans l'assemblage lors de la compilation. Cette information peut alors être exploitée par le compilateur C#. Lorsqu'il rencontre un appel à cette méthode, il peut émettre un avertissement indiquant qu'il vaut mieux éviter d'invoquer une méthode obsolète, qui risque de

disparaître dans les prochaines versions. Sans attribut, vous seriez obligé de commenter et de documenter l'obsolescence de la méthode `Fct()`. La faiblesse de cette démarche est que vous n'auriez aucune garantie que vos clients soient au courant de cette obsolescence.

## *Quand a t-on besoin des attributs ?*

Tout l'intérêt d'utiliser un attribut réside dans le fait que l'information qu'il contient est insérée dans l'assemblage. Cette information peut alors être consommée à divers moments par divers consommateurs pour divers desseins :

- Un attribut peut être consommé par le compilateur. L'attribut `System.ObsoleteAttribute` que nous venons de voir constitue un bon exemple de consommation d'attribut par le compilateur. Certains attributs standard destinés à n'être consommés que par le compilateur ne sont pas stockés dans l'assemblage. Par exemple, l'attribut `SerializationAttribute` ne marque pas directement un type mais indique au compilateur que ce type est sérialisable. En conséquence, le compilateur positionne certains drapeaux sur le type concerné qui seront consommés par le CLR à l'exécution. De tels attributs sont nommés *pseudo-attributs*.
- Un attribut peut être consommé par le CLR à l'exécution. Par exemple, le framework .NET présente l'attribut `System.ThreadStaticAttribute`. Lorsqu'un champ statique est marqué avec cet attribut, le CLR fait en sorte qu'à l'exécution il existe une version de ce champ par thread.
- Un attribut peut être consommé par un débuggeur à l'exécution. Ainsi l'attribut `System.Diagnostics.DebuggerDisplayAttribute` permet de personnaliser l'affichage d'un élément du code (l'état d'un objet par exemple) lors du déboguage.
- Un attribut peut être consommé par un outil. Par exemple, le framework .NET présente l'attribut `System.Runtime.InteropServices.ComVisibleAttribute`. Lorsqu'une classe est marquée avec cet attribut, l'outil `tlbexp.exe` génère un fichier qui permettra ultérieurement à cette classe d'être consommée comme si elle était définie avec la technologie COM.
- Un attribut peut être consommé par votre propre code à l'exécution, en ayant recours au mécanisme de réflexion pour accéder à l'information. Ainsi, il peut être intéressant de définir l'intégrité des champs de vos classes en marquant leurs champs avec des attributs. Tel champ entier doit être dans telle plage de valeur. Tel champ de type référence ne doit jamais être nul. Tel champ doit référencer une chaîne d'au plus 100 caractères etc. Grâce au mécanisme de réflexion, il est aisé d'écrire du code capable de valider l'état de n'importe qu'elle classe dont les champs sont marqués. Nous détaillons un peu plus loin un exemple de consommation d'attributs à l'exécution par du code propriétaire.
- Un attribut peut être consommé par un utilisateur qui analyse un assemblage avec un outil tel que `ildasm.exe`. Ainsi, nous pouvons imaginer un attribut qui permettrait d'associer une chaîne de caractères à un élément du code source. Cette chaîne de caractères étant insérée dans l'assemblage, il devient alors possible de consulter des commentaires sans avoir besoin du code source.

## *Quelques précisions sur les attributs*

- Un attribut est nécessairement définit par une classe qui dérive de la classe `System.Attribute`.

- Une classe d'attribut n'est instanciée que lorsque le mécanisme de réflexion accède à un de ses représentant. Selon l'utilisation, une classe d'attribut n'est donc pas forcément instanciée (à l'instar de la classe `System.ObsoleteAttribute` qui n'a pas à être utilisé avec le mécanisme de réflexion).
- Le Framework .NET met à votre disposition de nombreux attributs. Certains sont destinés à être consommés par le CLR. D'autres sont consommés par le compilateur ou des outils fournis par MS.
- Vous avez la possibilité de créer vos propres classes d'attributs. Ils seront alors nécessairement consommés par vos propres programmes ou outils puisque vous ne pouvez pas modifier ni le compilateur ni le CLR.
- Par convention, le nom d'une classe d'attribut est suffixé par `Attribute`. Cependant, un attribut nommé `XXXAttribute` peut en C# être utilisé à la fois avec l'expression `XXXAttribute` mais aussi avec l'expression `XXX` lorsqu'il marque un élément du code.

Dans le chapitre consacré à la généricité, nous présentons en page 497 les règles relatives au recoupements entre les deux notions d'attribut et de généricité.

## *Eléments du code source sur lesquels s'appliquent les attributs*

Les attributs s'appliquent à des éléments de votre code source. Voici exactement quels sont ces éléments. Ils sont définis par les valeurs de l'énumération `AttributeTargets`.

| Nom des éléments | Champ d'application de l'attribut |
|---|---|
| `All` | Tous les éléments du code source à savoir : l'assemblage lui-même, les classes, les membres des classes, les délégués, les événements, les champs, les interfaces, les méthodes, les modules, les paramètres, les propriétés, les valeurs de retour et les structures. |
| `Assembly` | L'assemblage lui-même. |
| `Class` | Les classes. |
| `Constructor` | Les constructeurs. |
| `Delegate` | Les délégués. |
| `Enum` | Les énumérations. |
| `Event` | Les événements. |
| `Field` | Les champs. |
| `GenericParameter` | Les paramètres génériques. |
| `Interface` | Les interfaces. |
| `Method` | Les méthodes. |
| `Module` | Les modules. |

| | |
|---|---|
| `Parameter` | Les paramètres des méthodes. |
| `Property` | Les propriétés des classes. |
| `ReturnValue` | Les valeurs de retour des méthodes. |
| `Struct` | Les structures. |

## *Quelques attributs standard du Framework .NET*

La bonne compréhension d'un attribut découle de la bonne compréhension de la situation dans laquelle il faut l'utiliser. Aussi, chaque attribut standard est décrit dans le chapitre qui le concerne.

Quelques attributs standard relatifs à la gestion de la sécurité sont présentés page 203.

Quelques attributs standard relatifs au mécanisme P/Invoke sont présentés dans la section page 265.

Quelques attributs standard relatifs à l'utilisation de COM à partir d'applications. NET sont présentés dans la section page 281.

Quelques attributs standard relatifs au mécanisme de sérialisation sont présentés page 790.

Quelques attributs standard relatifs au mécanisme de sérialisation XML sont présentés page 780.

Quelques attributs standard relatifs aux assemblages sont présentés page 25.

Quelques attributs standard permettant de personnaliser le comportement d'un débogueur sont présentés en page 617.

L'attribut `System.Runtime.Remoting.Contexts.Synchronization` qui permet d'implémenter un mécanisme de synchronisation est présenté page 160.

L'attribut `ConditionalAttribute` qui permet d'indiquer au compilateur C# s'il doit compiler ou non certaines méthodes est présenté page 314.

L'attribut `ThreadStaticAttribute` qui sert à modifier le comportement des threads vis-à-vis des champs statiques est présenté page 176.

L'attribut `CLSCompliantAttribute` qui sert à indiquer au compilateur s'il doit faire certaines vérifications est présenté page 131.

L'attribut `ParamArrayAttribute` qui sert à implémenter le mot clé C# `params` est présenté page 405.

L'attribut `CategoryAttribute` est présenté page 685.

## *Exemple d'un attribut personnalisé*

Un *attribut personnalisé* (*custom attribute* en anglais) est un attribut que vous créez vous-même en faisant dériver une de vos classes de la classe `System.Attribute`. À l'instar des attributs validateurs de champs décris au début de cette section, on peut imaginer de multiples situations où l'on peut bénéficier d'attributs personnalisés. L'exemple que nous présentons ici s'inspire de l'implémentation de l'outil *open-source* `NUnit`.

L'outil NUnit permet d'exécuter et donc de tester, n'importe quelle méthode de n'importe quel assemblage. Comme il n'y a pas de sens à exécuter toutes les méthodes d'un assemblage, NUnit n'exécute que les méthodes marquées avec un attribut de type TestAttribute.

Pour implémenter une version simplifiée de ce comportement, nous nous imposons les contraintes suivantes :

- Le test d'une méthode est considéré comme concluant si celle-ci n'envoie aucune exception non rattrapée.
- Nous définissions un attribut TestAttribute qui peut s'appliquer sur les méthodes. L'attribut peut être paramétré par le nombre de fois que la méthode doit être exécutée (propriété int TestAttribute.nTime). L'attribut peut aussi être paramétré pour permettre d'ignorer une méthode marquée (propriété bool TestAttribute.nTime).
- La méthode Program.TestAssembly(Assembly) permet d'exécuter toutes les méthodes contenues dans l'assemblage passé en référence et marquées avec l'attribut TestAttribute. Pour simplifier, nous supposerons que ces méthodes sont publiques, non statiques et ne prennent pas d'arguments. Nous sommes contraint d'utiliser un lien tardif pour invoquer les méthodes marquées.

Le programme suivant satisfait ce cahier des charges.

*Exemple 7-15 :*

```
using System ;
using System.Reflection ;

[AttributeUsage(AttributeTargets.Method,AllowMultiple=false)]
public class TestAttribute : System.Attribute {
   public TestAttribute() {
      Console.WriteLine("TestAttribute.ctor() par défaut.") ;
   }
   public TestAttribute(int nTime) {
      Console.WriteLine("TestAttribute.ctor(int).") ;
      m_nTime = nTime ;
   }
   private int m_nTime = 1 ;
   private bool m_Ignore = false ;
   public bool Ignore { get { return m_Ignore ; }
                        set { m_Ignore = value ; } }
   public int  nTime  { get { return m_nTime ; }
                        set { m_nTime = value ; } }
}

class Program{
   static void Main(){
      // Le programme s'analyse lui-même
      TestAssembly(Assembly.GetExecutingAssembly()) ;
   }
   static void TestAssembly(Assembly assembly) {
      // Pour toutes les méthodes de tous les types de 'assembly'
```

```
      foreach(Type type in assembly.GetTypes() ){
         foreach(MethodInfo method in type.GetMethods() ){
            // Obtient les attributs de type 'TestAttribute'
            // qui marquent la méthode référencée par 'method'.
            // Déclenche l'appel à 'TestAttribute.ctor()'.
            object[] attributes = method.GetCustomAttributes(
                                  typeof(TestAttribute),false) ;
            if( attributes.Length == 1 ){
               // Obtient une référence de type 'TestAttribute'.
               TestAttribute testAttribute =
                                 attributes[0] as TestAttribute ;
               // Si la méthode n'est pas à ignorer.
               if( ! testAttribute.Ignore ){
                  object [] parameters = new object[0] ;
                  object instance = Activator.CreateInstance(type) ;
                  // Invoque la méthode 'nTime' fois.
                  for(int i=0;i< testAttribute.nTime ; i++){
                     try{
                       //Invocation de la méthode avec un lien tardif.
                       method.Invoke(instance,parameters) ;
                     } catch(TargetInvocationException ex) {
                       Console.WriteLine(
                              "La méthode {" + type.FullName + "." +
                              method.Name +
                              "} a lancé une exception de type " +
                              ex.InnerException.GetType() +
                              " lors de l'exécution " + (i+1) + ".") ;
                     }// end catch(...
                  }// end for(...
               }// end if( ! attribute.Ignore )
            }// end if( attributes.Length == 1 )
         }// end foreach(MethodInfo...
      }// end foreach(Type...
   }
}

class Foo {
   [Test()]
   public void Plante() {
      Console.WriteLine("Plante()") ;
      throw new ApplicationException() ;
   }
   int state = 0 ;
   [Test(4)]
   public void PlanteLaDeuxiemeFois() {
      Console.WriteLine("PlanteLaDeuxiemeFois()") ;
      state++ ;
      if (state == 2) throw new ApplicationException() ;
   }
```

```
   [Test()]
   public void NePlantePas() {
      Console.WriteLine("NePlantePas()") ;
   }
   [Test(Ignore = true)]
   public void PlanteMaisIgnore() {
      Console.WriteLine("PlanteMaisIgnore()") ;
      throw new ApplicationException() ;
   }
}
```

Ce programme affiche :

```
TestAttribute.ctor() par défaut.
Plante()
La méthode {Foo.Plante} a lancé une exception de type
                        System.ApplicationException lors de l'exécution 1.
TestAttribute.ctor(int).
PlanteLaDeuxiemeFois()
PlanteLaDeuxiemeFois()
La méthode {Foo.PlanteLaDeuxiemeFois} a lancé une exception de type
                        System.ApplicationException lors de l'exécution 2.
PlanteLaDeuxiemeFois()
PlanteLaDeuxiemeFois()
TestAttribute.ctor() par défaut.
NePlantePas()
TestAttribute.ctor() par défaut.
```

Plusieurs remarques s'imposent :

- On marque notre classe `TestAttribute` avec un attribut de type `AttributeUsage`.
- On se sert de la valeur `Method` de l'énumération `AttributeTarget` pour signifier au compilateur que l'attribut `TestAttribute` ne peut s'appliquer qu'aux méthodes.
- On positionne à `false` la propriété `AllowMultiple` de la classe `AttributeUsage` pour signifier qu'une méthode ne peut accepter plusieurs attributs de type `TestAttribute`. Notez la syntaxe particulière pour initialiser la propriété `AllowMultiple`. On dit que `AllowMultiple` est un *paramètre nommé*.
- On se sert aussi de la syntaxe du paramètre nommé dans la déclaration de l'attribut `Test` qui marque la méthode `Foo.PlanteMaisIgnore()`.
- On se sert du fait que lorsqu'une exception est lancée et non rattrapée lors de l'exécution d'une méthode invoquée au moyen d'un lien tardif, c'est une exception de type `TargetInvocationException` qui est récupérée par l'appelant. L'exception initiale est alors référencée par la propriété `InnerException` de l'exception récupérée.
- Pour éviter d'avoir à éclater le code sur plusieurs assemblages, ce programme se teste lui-même (en fait seules les méthodes de la classe `Foo` sont testée puisque ce sont les seules méthodes marquées avec l'attribut `TestAttribute`). Voici l'organisation des assemblages que l'on aurait si le code était éclaté :
- La ressemblance de ce schéma avec celui de la Figure 7-2 n'est pas fortuite. Dans les deux cas, nous exploitons un élément inconnu à la compilation par l'intermédiaire d'un élément connu de tous à la compilation (un attribut dans ce cas, une interface précédemment).

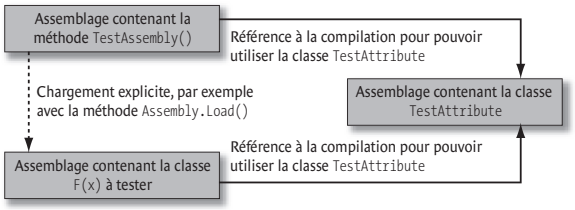


*Figure 7-3 : Organisation des assemblages*

## *Les attributs conditionnels*

C#2 introduit la notion d'*attribut conditionnel*. Un attribut conditionnel n'est pris en compte par le compilateur que si un symbole est défini. L'exemple suivant illustre cette notion d'attribut conditionnel au travers d'un projet constitué de trois fichiers :

*Exemple 7-16 :*

```
[System.Diagnostics.Conditional("_TEST")]
public class TestAttribute : System.Attribute { }
```

*Exemple 7-17 :*

```
#define _TEST
[Test]  // Le compilateur marque la classe Foo1 avec l'attribut Test.
class Foo1 { }
```

*Exemple 7-18 :*

```
#undef _TEST
[Test]  // Le compilateur ne marque pas la classe Foo2
        // avec l'attribut Test.
class Foo2 { }
```

Dans l'exemple de la section précédente, les attributs conditionnels pourraient être exploités pour générer une version de test et une version destinée à la production à partir du même code.

En page 314 nous décrivons une autre utilisation de l'attribut `Conditional`.

# *Construction et utilisation dynamique d'un assemblage*

L'espace de noms `System.Reflection.Emit` présente un ensemble de classes permettant de fabriquer à l'exécution d'un assemblage, de nouveaux assemblages. C'est en fait de la génération de code, mis à part que le code n'est pas généré sous la forme de fichiers sources C# ou C++, mais bel et bien sous la forme d'un assemblage prêt à être exécuté. Vous avez ensuite la possibilité de sauver l'assemblage d'une manière persistante. Le mot « Emit » s'apparente au fait d'émettre du code.

## Pourquoi construire dynamiquement un assemblage ?

Lorsque vous fabriquez un assemblage avec les classes de l'espace de noms `System.Reflection.Emit`, le code est émit en langage IL. Ce langage est brièvement présenté dans la section page 41. Bien que le langage IL soit fondamental, il est maîtrisé par beaucoup moins de développeurs que le langage C# ou le langage VB.NET. Pour la plupart des développeurs, il est donc beaucoup plus fastidieux de générer du code en langage IL, que de générer un fichier source C# qui sera compilé ultérieurement. En fait, il est toujours préférable de générer du code en un langage structuré comme C# ou VB.NET.

Mais alors, pourquoi donc générer dynamiquement des assemblages ?

Il existe en fait au moins trois cas de figure, présentés dans les MSDN à l'article **Reflection Emits Application Scenarios** :

- L'exécution d'un script dans un navigateur web : l'idée est qu'un script dans un page Web construit dynamiquement un assemblage qui est sauvé de façon persistante chez le client.
- L'exécution d'un script dans une page ASP.NET : l'idée est qu'un script dans un page ASP.NET, construit dynamiquement un assemblage qui est sauvé de façon persistante dans le cache du serveur. Ainsi, seule la première visite de la page provoque la création de l'assemblage.
- La compilation d'une expression régulière fournie à l'exécution. Le problème de l'évaluation d'une expression régulière est représentatif de cette classe de problèmes qui admettent une solution globale lente, et des solutions particulières rapides. L'idée est de construire une solution particulière à l'exécution, lorsque l'expression régulière est fournie, plutôt que d'implémenter au développement la solution globale lente. Le *framework* .NET permet la compilation des expressions régulières et tout ceci est expliqué page 625.

Tout ceci peut paraître abstrait aussi allons-nous présenter un exemple concret qui a la mérite d'avoir une utilité potentielle dans l'industrie.

## Un problème concret

### Présentation du problème

L'exemple présenté ici est basé sur l'évaluation d'un polynôme $P$ à coefficients entiers, défini sur les entiers. Imaginez une application où ou un tel polynôme est fourni à l'exécution (par exemple par un utilisateur). Supposons que l'application doive, par la suite, évaluer ce polynôme pour un très grand nombre de valeurs. Nous allons montrer que ce problème concret admet une solution très optimisée, utilisant la fabrication à l'exécution d'un nouvel assemblage.

Par la suite nous allons supposer que le polynôme $P$ saisi par l'utilisateur durant l'exécution de l'application est :

$$P(x) = 66x^3 + 83x^2 - 13\,735x + 30\,139$$

Pour la petite histoire, ce polynôme découvert par les mathématiciens *Dress* et *Landreau* en 1999, a la particularité de ne prendre pour valeurs que des nombres premiers pour $x$ entre $-26$ et 19 (inclus). Nous aurions pu prendre n'importe quel autre polynôme à coefficients entiers pour illustrer notre exemple.

Pour nos tests de performance, nous évaluerons *P* pour chacune de ces valeurs 10 millions de fois. On ne travaillera qu'avec des entiers sur quatre octets signés (le type `int` en C#, qui est aussi le type `System.Int32`). Pour ceux qui ont oublié leurs cours de mathématique nous rappelons que la manière la plus économique en terme d'opérations d'évaluer ce polynôme est de l'écrire sous cette forme :

$$P(x) = 30\,139 + x(-13\,735 + x(83 + 66x))$$

Cette astuce s'appelle la méthode de *Hörner*. Seulement trois multiplications et trois additions sont nécessaires pour évaluer *P* pour une valeur de *x*.

## *Solution1 : La méthode globale*

Fort de cette connaissance, la solution immédiate qui vient à l'esprit est :

*Exemple 7-19 :*

```
using System ;
using System.Diagnostics;
class Program {
   static int Evalue(int x,int[] Coefs) {
      int tmp = 0;
      int degre = Coefs.GetLength(0);
      for(int i=degre-1 ; i>= 0 ; i--)
         tmp = Coefs[i]+x*tmp;
      return tmp;
   }
   static void Main() {
      Stopwatch sw = Stopwatch.StartNew();
      int [] Coefs = {30139,-13735,83,66};
      for( int x = -26 ; x<= 19 ; x++ )
         for( int i = 0 ; i<10000000 ; i++)
            Evalue(x,Coefs);
      Console.WriteLine("Durée :" + sw.Elapsed );
   }
}
```

Puisque nous focalisons nos tests sur la performance, il n'y a pas lieu ici de vérifier la primarité des entiers retournés par la méthode `Evalue()`.

## *Solution 2 : La méthode particulière*

Si nous n'avions pas eu la contrainte de définir les coefficients du polynôme hors de la méthode `Evalue()` nous aurions pu écrire le programme de cette façon :

*Exemple 7-20 :*

```
using System ;
using System.Diagnostics;
class Program {
   static int Evalue(int x) {
```

```
      return 30139-x*(13735-x*(83+x*66));
   }
   static void Main() {
      Stopwatch sw = Stopwatch.StartNew();
      for( int x = -26 ; x<= 19 ; x++ )
         for( int i = 0 ; i<10000000 ; i++)
            Evalue(x);
      Console.WriteLine("Durée :" + sw.Elapsed );
   }
}
```

### *Comparaison des performances des deux solutions*

La durée obtenue sur une machine de référence pour la première solution (la méthode globale) est de 17,81 secondes. La durée obtenue sur la même machine pour la seconde solution (la méthode particulière) est de 3,87 secondes soit un gain de **4,6** par rapport à la première méthode. Beaucoup d'instructions machines, comme le passage par référence du tableau, la recherche des coefficients dans le tableau et la gestion de la boucle, ont disparu dans la seconde solution.

Ces résultats montrent bien que l'évaluation d'un polynôme particulier est beaucoup plus rapide que l'utilisation d'un algorithme global d'évaluation d'un polynôme.

Deux problèmes potentiels pourraient éventuellement fausser ces tests de performance.

- Problème potentiel 1 : Le ramasse-miettes pourrait se déclencher dans un des tests et pas dans l'autre. Nous avons vérifié que le ramasse-miettes ne se déclenche pas, étant donnée la faible quantité de mémoire demandée par ces programmes.
- Problème potentiel 2 : Le compilateur JIT pourrait être assez « intelligent » pour s'apercevoir que ce n'est pas la peine d'appeler la méthode `Evalue()` puisqu'on n'utilise pas ses résultats. Nous avons pu établir que ce n'est pas le cas en vérifiant que si la méthode `Evalue()` incrémente un compteur global, les résultats sont sensiblement équivalents.

## *Une solution performante grâce à la création dynamique d'un assemblage*

Le problème de la méthode particulière par rapport à la méthode globale est que les coefficients dans la méthode `Evalue()` doivent être connus à la compilation. Le polynôme ne peut pas être fourni au moment de l'exécution, mais seulement au moment de la compilation. Grâce au mécanisme de construction d'un assemblage à l'exécution, nous allons pouvoir utiliser la méthode particulière, plus rapide, tout en acceptant le polynôme à l'exécution. En effet, il suffit de générer le code IL de la méthode `Evalue()` de la méthode particulière.

Pour le polynôme donné en exemple dans la section précédente, le code IL généré par le compilateur C# dans la seconde solution est celui-ci :

```
.method private hidebysig static int32  Calc(int32 x) cil managed
{
  // Code size       28 (0x1c)
  .maxstack  7
  .locals init ([0] int32 CS$00000003$00000000)
```

```
  IL_0000:  ldc.i4     0x75bb // coef 30139 mis sur le haut de la pile
  IL_0005:  ldarg.0           // valeur de x mise sur le haut de la pile
  IL_0006:  ldc.i4     0xffffca59// coef -13735 mis sur le haut de la pile
  IL_000b:  ldarg.0           // valeur de x mise sur le haut de la pile
  IL_000c:  ldc.i4.s   83     // coef 83 mis sur le haut de la pile
  IL_000e:  ldarg.0           // valeur de x mise sur le haut de la pile
  IL_000f:  ldc.i4.s   66     // coef 66 mis sur le haut de la pile
  IL_0011:  mul               //
  IL_0012:  add               //
  IL_0013:  mul               //     Evaluation du polynôme en x
  IL_0014:  add               //     avec trois additions et
  IL_0015:  mul               //     trois multiplications
  IL_0016:  add               //
  IL_0017:  stloc.0
  IL_0018:  br.s       IL_001a
  IL_001a:  ldloc.0
  IL_001b:  ret
} // end of method Program::Calc
```

Il suffit maintenant de produire ce code IL pour n'importe quel polynôme, avec les classes de l'espace de noms `System.Reflection.Emit`.

Dans cet exemple, l'instruction IL `ldarg` est utilisée avec le paramètre 0 pour charger le premier argument. Si la méthode n'avait pas été statique `ldarg.0` aurait représenté le pointeur `this` et il aurait fallu utiliser `ldarg.1` pour accéder au premier argument. Par la suite la méthode ne sera pas statique, aussi nous utiliserons `ldarg.1` pour accéder au premier élément.

Voici le code :

*Exemple 7-21 :*

```
using System ;
using System.Reflection ;
using System.Reflection.Emit ;
using System.Threading ;
using System.Diagnostics ;

public interface IPolynome {
   int Evalue(int x) ;
}
class Polynome {
   public IPolynome polynome ;
   public Polynome(int[] coefs) {
      Assembly asm = BuildCodeInternal(coefs) ;
      polynome = (IPolynome)asm.CreateInstance("PolynomeInternal") ;
   }
   private Assembly BuildCodeInternal(int[] coefs) {
      AssemblyName asmName = new AssemblyName() ;
```

```
      asmName.Name = "EvalPolAsm" ;

      // Construit un assemblage dynamique.
      AssemblyBuilder asmBuilder =
         Thread.GetDomain().DefineDynamicAssembly(
            asmName, AssemblyBuilderAccess.Run) ;

      // Construit un module dans l'assemblage.
      ModuleBuilder modBuilder =
         asmBuilder.DefineDynamicModule("MainMod") ;

      // Ajoute la classe PolynomeInternal dans le module.
      TypeBuilder typeBuilder = modBuilder.DefineType(
            "PolynomeInternal", TypeAttributes.Public) ;
      typeBuilder.AddInterfaceImplementation(typeof(IPolynome)) ;

      // Implémente la méthode int Evalue(int)
      MethodBuilder methodBuilder = typeBuilder.DefineMethod(
            "Evalue",
            MethodAttributes.Public | MethodAttributes.Virtual,
            typeof(int),                    // type retour
            new Type[] { typeof(int) }) ;   // argument

      // Genère le code IL à partir du tableau des coefficients.
      ILGenerator ilGen = methodBuilder.GetILGenerator() ;
      int deg = coefs.GetLength(0) ;
      for (int i = 0 ; i < deg - 1 ; i++) {
         ilGen.Emit(OpCodes.Ldc_I4, coefs[i]) ;
         ilGen.Emit(OpCodes.Ldarg, 1) ;
      }
      ilGen.Emit(OpCodes.Ldc_I4, coefs[deg - 1]) ;
      for (int i = 0 ; i < deg - 1 ; i++) {
         ilGen.Emit(OpCodes.Mul) ;
         ilGen.Emit(OpCodes.Add) ;
      }
      ilGen.Emit(OpCodes.Ret) ;

      // Indique que la méthode Evalue() ...
      // ... implémente celle de l'interface IPolynome.
      MethodInfo methodInfo = typeof(IPolynome).GetMethod("Evalue") ;
      typeBuilder.DefineMethodOverride(methodBuilder, methodInfo) ;
      typeBuilder.CreateType() ;
      return asmBuilder ;
   }
}

class Program {
   static void Main() {
      Stopwatch sw = Stopwatch.StartNew();
```

```
      int[] coefs = { 30139, -13735, 83, 66 } ;
      Polynome p = new Polynome(coefs) ;
      for (int x = -26 ; x <= 19 ; x++)
         for (int i = 0 ; i < 10000000 ; i++)
            p.polynome.Evalue(x) ;
      Console.WriteLine("Durée :" + sw.Elapsed);
   }
}
```

### *Comparaison avec les deux méthodes précédentes*

La durée obtenue sur la machine de référence pour cette troisième solution est de 4.36 secondes. C'est 1.12 fois plus lent qu'avec la deuxième solution, mais la création de l'assemblage ne rentre pratiquement pas en ligne de compte, puisqu'elle consomme entre 2 et 4 centièmes de seconde. On peut supposer que cette petite baisse de performance est due à l'utilisation d'une méthode non statique, qui oblige à passer une référence vers une instance à chaque appel de `Evalue()`. De plus l'appel à une méthode définie dans une interface est plus coûteux que l'appel à une méthode non virtuelle et non abstraite. En effet, l'instruction IL `callvirt` est utilisée pour l'appel à une méthode définie dans une interface, alors que l'instruction IL `call` est utilisée pour l'appel à une méthode statique. Or, le compilateur JIT génère du code machine consommant moins de cycle horloge pour l'instruction IL `call` que pour l'instruction IL `callvirt`. Nous verrons un peu plus tard, que cette petite baisse de performance est aussi due au fait que nous ne tenons pas compte que certains coefficients du polynôme peuvent être stockés sur un octet (en l'occurrence, les coefficients 83 et 66). Or, le compilateur C# profite de cette situation pour utiliser des instructions IL spécialement optimisées pour manipuler des entiers sur un octet.

En revanche cette troisième solution est **4.1** fois plus rapide que la première solution, tout en respectant les mêmes contraintes (i.e les coefficients du polynômes sont définies hors de la méthode `Evalue()`). C'est un gain énorme, résultant directement de la création dynamique d'un assemblage. Une optimisation est encore possible si l'on tient compte des coefficients nuls.

### *Description technique du code*

Les types intéressants dans le code précédent sont :

```
AssemblyName
AssemblyBuilder
ModuleBuilder
TypeBuilder
MethodBuilder
ILGenerator
MethodInfo
```

Ils sont généralement utilisés dans cet ordre, de la même façon que dans cet exemple. Il existe de nombreux autres types détaillés dans les **MSDN**. Ces autres types prennent notamment en charge la gestion des exceptions, la gestion des branchements, la gestion des autres éléments d'une classe (propriétés, événements).

## *Possibilité de sauver l'assemblage sur le disque*

Si vous aviez voulu sauver l'assemblage sur le disque il aurait fallu :

- Nommer le module (par exemple `MainMod.dll`).
- Utiliser `AssemblyBuilderAccess.RunAndSave` au lieu de `AssemblyBuilderAccess.Run`.
- Sauver l'assemblage avec la ligne de code suivant, juste avant le retour de la méthode `BuildCodeInternal()` :

```
TheAsm.Save("MainMod.dll") ;
```

Nous pouvons comparer le code IL produit par le compilateur C# et le code IL produit par notre propre programme :

*Code IL produit par le compilateur C#*

```
.method private hidebysig static
int32 Calc(int32 x) cil managed
{
// Code size 28 (0x1c)
 .maxstack 7
 .locals init ([0]
 int32 CS$00000003$00000000)
 IL_0000: ldc.i4 0x75bb
 IL_0005: ldarg.0
 IL_0006: ldc.i4 0xffffca59
 IL_000b: ldarg.0
 IL_000c: ldc.i4.s 83
 IL_000e: ldarg.0
 IL_000f: ldc.i4.s 66
 IL_0011: mul
 IL_0012: add
 IL_0013: mul
 IL_0014: add
 IL_0015: mul
 IL_0016: add
 IL_0017: stloc.0
 IL_0018: br.s IL_001a
 IL_001a: ldloc.0
 IL_001b: ret
} // end of method CCalc::Calc
```

*Code IL produit dynamiquement par notre programme*

```
.method public virtual instance
int32 Evalue(int32 A_1) cil managed
{
 .override [DynamicAssembly3]
 IPolynome::Evalue
// Code size 30 (0x1e)
 .maxstack 7
 IL_0000: ldc.i4 0x75bb
 IL_0005: ldarg.1
 IL_0006: ldc.i4 0xffffca59
 IL_000b: ldarg.1
 IL_000c: ldc.i4 0x53
 IL_0011: ldarg.1
 IL_0012: ldc.i4 0x42
 IL_0017: mul
 IL_0018: add
 IL_0019: mul
 IL_001a: add
 IL_001b: mul
 IL_001c: add
 IL_001d: ret
} // end of method CPolynomeInternal::Evalue
```

Les différences sont dues aux faits suivants :

- Nous avons produit dynamiquement une méthode non statique.
- Nous n'utilisons pas de variables locales, contrairement au code produit par le compilateur C#.
- Nous n'utilisons pas l'instruction optimisée `ldc.i4.s` qui charge sur la pile une valeur d'un octet (i.e entre -128 et 127) dans quatre octets. Nous pourrions optimiser notre programme en utilisant cette instruction.

## *Conclusion*

Vous avez vu ici un exemple représentatif des possibilités de la création dynamique d'assemblages. On pourrait l'adapter simplement à d'autres domaines aussi utiles que le calcul vectoriel (composition avec une matrice, évaluation d'une forme quadratique etc). Il arrive souvent qu'une même matrice, inconnue avant l'exécution, soit composée des millions de fois (par exemple pour calculer les déplacements des points d'une scène 3D).

La majorité des développeurs n'auront jamais à construire d'assemblages dynamiquement, mais certains projets ont beaucoup à gagner à utiliser cette possibilité.

# 8

# *Interopérabilité .NET code natif / COM / COM+*

La plateforme .NET présente plusieurs techniques pour faire interopérer du code géré avec du *code natif* et pour faire coopérer des objets gérés avec des objets COM. Ce besoin est particulièrement présent lors d'un processus de migration d'une application C++ ou VB6 vers .NET. En effet, l'interopérabilité permet de migrer vos projets petit à petit, composant après composant.

## *Le mécanisme P/Invoke*

Le CLR utilisé conjointement avec certaines classes de base du *framework* .NET, permet à du code géré d'appeler des fonctions compilées en code natif. Cette possibilité est nommée *Platform Invoke* ou *P/Invoke*. Vous pouvez l'utiliser pour appeler les fonctions de vos propres DLLs natives. *Microsoft* exploite aussi le mécanisme P/Invoke pour appeler les fonctions de ses propres DLLs natives.

Notons l'existence du mécanisme *internal call* qui a la même finalité que P/Invoke. Ce mécanisme consiste à implémenter les artefacts des appels aux fonctions natives directement (en dur) dans le code du CLR. Il est donc plus performant que P/Invoke mais n'est utilisable que par *Microsoft*.

### *L'attribut DllImport*

Les classes qui permettent d'utiliser P/Invoke se trouvent dans l'espace de noms `System.Runtime.InteropServices`. Pour appeler une fonction d'une DLL native à partir d'un programme C#, il faut d'abord déclarer cette fonction dans une classe C# :

- La déclaration de cette fonction doit être marquée avec l'attribut `System.Runtime.InteropServices.DllImport` qui indique le nom de la DLL.

- Utiliser les mots-clés `static` et `extern` devant la déclaration de la méthode.
- Garder le même nom de fonction que celui spécifié dans la DLL.
- Donner un nom à chaque argument.

## *Un exemple*

Le programme suivant appelle la fonction `Beep()` définie dans la DLL native et standard `Kernel32.dll`.

*Exemple 8-1 :*

```
using System.Runtime.InteropServices ;
class Program {
   [DllImport("Kernel32.dll")]
   public static extern bool Beep(uint iFreq, uint iDuration);
   static void Main() {
      bool b = Beep(100, 100);
   }
}
```

Dans certains cas rares, vous ne pouvez pas spécifier le nom de la fonction définie dans la DLL car vous avez déjà une méthode qui a ce nom. Il est possible de changer le nom d'une fonction implémentée dans une DLL native comme ceci :

*Exemple 8-2 :*

```
using System.Runtime.InteropServices ;
class Program {
   [DllImport("Kernel32.dll", EntryPoint = "Beep")]
   public static extern bool MonBeep(uint iFreq, uint iDuration) ;
   static void Main() {
      bool b = MonBeep(100, 100) ;
   }
}
```

## *Convention d'appel*

Les fonctions implémentées dans les DLLs natives supportent plusieurs conventions d'appel. Ces conventions d'appel indiquent au compilateur comment doit se comporter le passage d'argument. Si vous devez utiliser une fonction qui a une convention d'appel particulière, il faut utiliser l'attribut `DllImport` comme ceci :

```
[DllImport("MaDLL.dll", CallingConvention=XXX)]
```

XXX est une valeur de l'énumération `System.Runtime.InteropServices.CallingConvention`. Les significations des valeurs de cette énumération sont exposées dans les **MSDN** à l'article **CallingConvention Enumeration**. Par défaut cette valeur vaut `StdCall` qui est la convention d'appel standard sur les systèmes d'exploitation *Microsoft* (mis à part *Windows CE* qui admet la convention d'appel standard `Cdecl`).

### *Impact au niveau des performances*

Lors de chaque appel d'une fonction d'une DLL native, deux étapes sont effectuées en interne. L'ensemble de ces étapes est appelé *P/Invoke Marshalling* :

- Il faut parcourir la pile pour vérifier que tous les appelants ont la permission `UnmanagedCode`. Afin d'améliorer les performances, vous pouvez supprimer cette étape en utilisant l'attribut `System.Security.SuppressUnmanagedCodeSecurity` sur une méthode « P/Invokée » ou sur une classe qui contient des méthodes « P/Invokée ». Cependant, l'utilisation de cet attribut peut être compromettante pour la sécurité.
- Il faut créer une *fenêtre de pile* (*stack frame* en anglais) qui soit compatible avec les fenêtres de pile utilisées dans le code non géré. Cette étape est en général très légère.

Puisque la seconde étape est légère et puisque la première étape peut être supprimée si la sécurité ne fait pas partie de vos considérations, vous pouvez faire en sorte que l'utilisation de P/Invoke ait un faible impact sur les performances.

### *Inférer les définitions des fonctions standard*

*Microsoft* ne fournit pas les définitions gérées de ses fonctions natives. Pour obtenir ces définitions, nous vous conseillons de consulter le site `www.pinvoke.net`.

## *Tableau de conversion des types*

Le prototype de `Beep()` dans la DLL `Kernel32.dll` est le suivant :

```
BOOL Beep(DWORD dwFreq,DWORD dwDuration) ;
```

Pour appeler `Beep()`, nous avons dû convertir le type `BOOL` win32 en `bool` .NET et le type `DWORD` win32 en `uint` .NET. Voici le tableau de conversion des types *win32* dans les types .NET.

| Type win32 | Type .NET | Equivalent C# |
|---|---|---|
| LPSTR, LPCSTR, LPWSTR, LPCWSTR | System.StringSystem. StringBuilder | string |
| BYTE | System.Byte | Byte |
| SHORT | System.Int16 | Short |
| WORD | System.UInt16 | ushort |
| DWORD, UINT,ULONG | System.Int32 | uint |
| INT, LONG | System.UInt32 | uint |
| BOOL | System.Bool | bool |
| CHAR | System.Char | char |
| FLOAT | System.Single | float |
| DOUBLE | System.Double | double |

On dit qu'un type est *blittable* si la représentation binaire de ses instances est identique en mode géré et en mode natif. Dans le cadre de l'interopérabilité, l'utilisation des types blittable est donc plus performante. Dans le tableau précédent, seuls les types de chaînes de caractères ne sont pas blittables. Les tableaux monodimensionnels d'éléments de type blittable ainsi que les types seulement composés de champs de types blittables sont aussi des types blittable.

## *Passage d'arguments par pointeurs*

En page 119 nous expliquons que le ramasse-miettes est amené à déplacer les objets de type référence en mémoire à tout moment. Si vous passez l'adresse d'un objet de type référence par l'intermédiaire d'un pointeur à du code non géré, vous vous exposez à priori au risque que le ramasse-miettes bouge cet objet en mémoire durant l'appel. Heureusement le mécanisme P/Invoke est capable de détecter cette situation. Le cas échéant, il épinglera l'objet concerné en mémoire et fournira un pointeur au code natif (la notion d'épinglage d'objet en mémoire est exposée en page 505). L'objet sera automatiquement « défixé » lors du retour de l'appel. Un autre problème se pose alors. Certaine fonctions natives stockent les pointeurs qui leurs sont passés de façon à les utiliser d'une manière asynchrone, après l'appel. Par exemple la fonction win32 `ReadFileEx()` définie comme suit...

```
BOOL ReadFileEx(
  HANDLE hFile,
  LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead,
  LPOVERLAPPED lpOverlapped,
  LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
) ;
```

...lit les données d'un fichier d'une manière asynchrone. Pour cela, elle stocke les pointeurs `lpBuffer` et `lpOverlapped`. Si vous invoquez cette fonction à partir de votre code géré, il est absolument nécessaire d'épingler explicitement les buffers passés durant toute la durée de la lecture asynchrone et pas seulement durant l'appel à la fonction.

Il faut donc être particulièrement vigilant car les problèmes engendrés par la non fixation d'un objet utilisé d'une manière asynchrone par du code géré se manifestent rarement et de manière aléatoires.

## *Passage de chaînes de caractères*

Pour passer une chaîne de caractères lors de l'appel d'une méthode native par l'intermédiaire du mécanisme P/Invoke, il vous suffit d'utiliser le type `System.String`. Sachez cependant que les fonctions natives ayant des paramètres d'entrés ou de retours de type chaînes de caractères existent chacune en deux versions : une version ANSI suffixée par un `A` et une version UNICODE suffixée par un `W`. Pour choisir entre ces deux versions, l'attribut `DllImport` présente le paramètre nommé `CharSet` qui peut prendre les valeurs `Auto`, `Unicode` ou `Ansi`. La valeur `Auto` est équivalente à la valeur `Ansi` sur les systèmes d'exploitation *Windows 98* et *Windows Me*. L'exemple suivant montre comment passer deux chaînes de caractères lors de l'appel de la méthode `MessageBox()` :

*Exemple 8-3 :*

```
using System.Runtime.InteropServices ;
class Program {
   [DllImport("user32.dll", CharSet = CharSet.Auto)]
   public static extern int MessageBox(System.IntPtr hWnd,
                           string text, string caption, uint type) ;
   static void Main() {
      MessageBox(System.IntPtr.Zero, "hello", "caption text", 0) ;
   }
}
```

## *Récupérer une chaîne de caractère*

Nous allons montrer ici comment exploiter une chaîne de caractères retournée par une fonction implémentée dans une DLL native. Dans du code non géré il y a deux façons de procéder :

- Soit le code appelant fournit une mémoire tampon et la taille de cette mémoire tampon. Dans ce cas la fonction appelée remplie la mémoire tampon avec la chaîne de caractères à retourner.
- Soit le code appelant s'attend à recevoir un pointeur à partir de la fonction appelée. Dans ce cas, la fonction appelée alloue la zone mémoire contenant la chaîne de caractères.

Il faut absolument tenir compte de ces différences lorsque l'on appelle au moyen de P/Invoke une fonction dont le code est non géré. La fonction `GetCurrentDirectory()` de la DLL `Kernel32.dll` est une bonne candidate pour illustrer le premier cas.

```
DWORD GetCurrentDirectory(
   DWORD   nBufferSize,
   LPTSTR  lpBuffer) ;
```

En effet, cette fonction admet un pointeur vers une zone de mémoire tampon et sa taille en argument. Voici le code C# permettant d'appeler cette fonction. On ne peut utiliser la classe `string` car les instances de cette classe sont immuables. Il faut donc se servir de la classe `System.Text.StringBuilder`.

*Exemple 8-4 :*

```
using System.Text ;
using System.Runtime.InteropServices ;
class Program {
   [DllImport("Kernel32.dll", CharSet = CharSet.Auto)]
   public static extern uint GetCurrentDirectory(
      uint Taille,
      StringBuilder sTmp);
   static void Main() {
      uint Taille = 255 ;
      StringBuilder sTmp = new StringBuilder( (int) Taille);
      uint i = GetCurrentDirectory(Taille, sTmp) ;
      System.Console.WriteLine(sTmp) ;
   }
}
```

La fonction `GetCommandLine()` de la DLL `Kernel32.dll` est une bonne candidate pour illustrer le second cas.

```
LPTSTR GetCommandLine() ;
```

Cette fonction retourne un pointeur vers la chaîne de caractères fournie en ligne de commande. Si nous utilisons le type `string` comme type de retour, le *P/Invoke marshaller* copiera la chaîne de caractères retournée dans une mémoire tampon allouée par ses soins. Ensuite le *P/Invoke marshaller* désallouera la chaîne de caractères originale. Or, cette chaîne de caractères originale ne doit pas être désallouée, car elle existait avant l'appel de la fonction `GetCommandLine()` et sera sûrement utilisée ultérieurement, dans d'autres appels de fonctions. Il faut donc utiliser la classe `IntPtr`, qui permet de copier la chaîne de caractères retournée dans une instance de la classe `string`, sans désallouer la chaîne de caractères originale :

*Exemple 8-5 :*

```
using System.Runtime.InteropServices ;
class Program {
   [DllImport("Kernel32.dll", CharSet = CharSet.Auto)]
   public static extern System.IntPtr GetCommandLine() ;
   static void Main() {
      System.IntPtr ptr = GetCommandLine() ;
      string sTmp = Marshal.PtrToStringAuto(ptr);
      System.Console.WriteLine(sTmp) ;
   }
}
```

## *Passage de structures et d'unions*

Lors de l'utilisation d'une structure dans un appel « P/Invoké », il faut tenir compte de la façon dont les champs sont stockés dans les instances de la structure. La plupart du temps les champs sont stockés de manière séquentielle dans l'ordre de leur déclaration. C'est-à-dire qu'ils sont stockés les uns après les autres, et le calcul de la position d'un champ par le compilateur se fait en sommant la taille des champs déclarés avant ce champ. Cependant, dans le cas d'une *union*, les champs sont stockés au même emplacement, ils ont tous la même position. Les unions n'existant pas dans .NET, il faut utiliser conjointement les attributs `System.Runtime.InteropServices.StructLayout` `System.Runtime.InteropServices.FieldOffset` lors de la déclaration d'une structure destinée à être utilisée dans des appels de fonctions implémentées dans des DLLs natives.

Par exemple la structure `Point` peut être déclarée en C# comme ceci :

```
[StructLayout(LayoutKind.Sequential)]
public struct Point {
      public int x ;
      public int y ;
}
```

La structure `Point` peut aussi être déclarée en C# comme cela :

```
[StructLayout(LayoutKind.Explicit)]
public struct Point {
```

```
      [FieldOffset(0)] public int x ;
      [FieldOffset(4)] public int y ;
}
```

L'union _Union déclarée comme ceci en C :

```
union _Union {
   int   i ;
   char  c ;
   float f ;
}
```

...peut être déclarée en C# comme cela :

```
[StructLayout(LayoutKind.Explicit)]
public struct Point {
      [FieldOffset(0)] public int i ;
      [FieldOffset(0)] public char c ;
      [FieldOffset(0)] public float f ;
}
```

Notez qu'avec l'attribut `System.Runtime.InteropServices.MarshalAs`, vous pouvez indiquer au *P/Invoke marshaller* comment transformer certains types. Par exemple la structure win32 suivante :

```
typedef struct {
    int   wStructSize ;
    int   x ;
    int   y ;
    int   dx ;
    int   dy ;
    int   wMax ;
    TCHAR rgchMember[2];
} HELPWININFO ;
```

...pourrait être déclarée en C# comme cela :

```
[StructLayout(LayoutKind.Sequential)]
public struct HELPWININFO {
   int   wStructSize ;
   int   x ;
   int   y ;
   int   dx ;
   int   dy ;
   int   wMax ;
   [MarshalAs(UnmanagedType.ByValArry, SizeConst =2)]
   public char[] rgchMember;
}
```

## *Attribut de direction*

Lors de la déclaration d'une méthode « P/Invokée », vous pouvez utiliser les attributs `System.Runtime.InteropServices.In` et `System.Runtime.InteropServices.Out` devant chaque argu-

ment de la fonction. Cela permet d'indiquer au *P/Invoke marshaller* de ne convertir l'argument qu'à l'entrée ou qu'à la sortie de la fonction. Par défaut tous les arguments sont convertis à l'entrée et à la sortie.

## *Délégués et pointeur non géré de fonction*

Vous pouvez invoquer une fonction définie dans une DLL native par l'intermédiaire d'un délégué fabriqué à partir d'un *pointeur non géré de fonction*. En effet, grâce aux méthodes statiques `GetDelegateForFunctionPointer()` et `GetFunctionPointerForDelegate()` de la classe `Marshal` les notions de délégués et de pointeurs sur fonction sont interchangeables :

*Exemple 8-6 :*

```
using System ;
using System.Runtime.InteropServices ;
class Program {
   internal delegate bool DelegBeep(uint iFreq, uint iDuration) ;
   [DllImport("kernel32.dll")]
   internal static extern IntPtr LoadLibrary(String dllname) ;
   [DllImport("kernel32.dll")]
   internal static extern IntPtr GetProcAddress(IntPtr hModule,
                                                String procName) ;
   static void Main() {
      IntPtr kernel32 = LoadLibrary( "Kernel32.dll" );
      IntPtr procBeep = GetProcAddress( kernel32, "Beep" );
      DelegBeep delegBeep = Marshal.GetDelegateForFunctionPointer(
         procBeep , typeof( DelegBeep ) ) as DelegBeep;
      delegBeep(100,100);
   }
}
```

# *Introduction à l'interopérabilité avec C++/CLI*

En plus d'exposer des facilités pour exploiter le mécanisme P/Invoke d'une manière transparente, le langage *C++/CLI* présente la particularité unique de pouvoir fabriquer des assemblages contenant à la fois du code natif et du code géré. Il est alors opportun de réaliser une petite incursion dans ce langage afin de vous donner les clés pour décider si vos besoins d'interopérabilité en justifient l'utilisation.

## *Le mécanisme It Just Works (IJW)*

La finalité du mécanisme *IJW* (acronyme pour *It Just Works* qui pourrait se traduire par *ça marche tout simplement !*) du langage C++/CLI est la même que celle du mécanisme *P/Invoke* présenté dans la section précédente : permettre d'appeler des fonctions natives à partir du code géré. En apparence, ces deux mécanismes sont différents puisque *IJW* n'utilise pas d'attributs. Du point de vue du CLR en revanche, les mécanismes *IJW* et *P/Invoke* sont une seule et même possibilité. Le code IL généré pour un appel avec le mécanisme *P/Invoke* est donc équivalent au code IL généré avec le mécanisme *IJW*. Pour s'en convaincre, réécrivons en C++/CLI l'Exemple 8-1 :

*Exemple 8-7 :*

```
// compile with: /clr
#include "stdafx.h"
#include "windows.h"
int main(){
    bool b = Beep(100, 100) ;
}
```

Le compilateur C++/CLI peut se passer de l'attribut `DllImport` car il connaît la définition de la méthode que l'on souhaite appeler grâce à l'inclusion du fichier d'entête adéquat. Dans notre exemple, la méthode `Beep()` est définie dans le fichier d'entête `windows.h`. La visualisation de l'assemblage produit par le compilateur C++/CLI avec l'outil `ildasm.exe` nous révèle que la méthode statique suivante a été fabriquée :

```
.method public static pinvokeimpl(lasterr stdcall)
 int32 modopt([mscorlib]System.Runtime.CompilerServices.CallConvStdcall)
 Beep(
  uint32 modopt([Microsoft.VisualC]Microsoft.VisualC.IsLongModifier) A_0,
  uint32 modopt([Microsoft.VisualC]Microsoft.VisualC.IsLongModifier) A_1)
      native unmanaged preservesig {
  .custom instance void
     [mscorlib]System.Security.SuppressUnmanagedCodeSecurityAttribute::
     .ctor() = ( 01 00 00 00 )
  // Embedded native code
  // Disassembly of native methods is not supported.
  //  Managed TargetRVA = 0x00001D92
} // end of method 'Global Functions'::Beep
```

La visualisation de l'assemblage produit par le compilateur C# pour l'Exemple 8-7 expose cette définition :

```
.method public hidebysig static pinvokeimpl("Kernel32.dll" winapi)
 bool Beep(uint32 iFreq,uint32 iDuration) cil managed preservesig{}
```

Du point de vue du CLR, les deux versions de la méthode `Beep()` sont à invoquer avec le mécanisme P/Invoke puisqu'elles ont toutes les deux le drapeau `pinvokeimpl`. Néanmoins, on remarque que le compilateur C++/CLI a généré du code natif pour localiser la méthode `Beep()` contenue dans la DLL `kernel32.dll` tandis que la version C# compte sur le CLR pour réaliser cet opération. D'autres différences sont à noter comme l'utilisation dans la version C++/CLI de l'attribut `SuppressUnmanagedCodeSecurity` (décrit page 201) et du modificateur `IsLongModifier` qui résout les problèmes dus au fait que les mots-clés C++/CLI `long` et `int` se référèrent tous deux au type `Int32`.

## *Types gérés et types non gérés*

Dans la version 2 du langage C++/CLI, un type marqué avec un des mots-clés `ref`, `value`, `interface` ou `enum` est un type géré tandis qu'un type non marqué par un de ces mots-clés est natif. En outre, une méthode définie après le `#pragma managed` sera compilée en langage IL tandis qu'une méthode définie après le `#pragma unmanaged` sera compilée en code natif. Le langage C++/CLI permet :

- La définition de types non gérés avec des méthodes dont le corps contient du code natif.
- La définition de types non gérés avec des méthodes dont le corps contient du code IL.
- La définition de types gérés avec des méthode dont le corps contient du code IL.

Les `#pragma managed` et `unmanaged` doivent être définis hors d'un type pour prendre effet. Un type ne peut donc avoir à la fois des méthodes natives et gérées. En outre, le langage C++/CLI ne permet pas la définition de types gérés avec des méthodes dont le corps contient du code natif. Tout cela est illustré par le programme suivant :

*Exemple 8-8 :*

```
// compile with: /clr
#include "stdafx.h"
#using <mscorlib.dll>
#pragma unmanaged
class TypeNatifCodeNatif{
public:
    TypeNatifCodeNatif(int state){ m_State = state ; }
    int GetEtat(){ return m_State ; }
private:
    int m_State ;
} ;
#pragma managed
class TypeNatifCodeIL {
public:
    TypeNatifCodeIL(int state){ m_State = state ; }
    int GetEtat(){ return m_State ; }
private:
    int m_State ;
} ;
ref class TypeGereCodeIL {
public:
    TypeGereCodeIL(int state){ m_State = state ; }
    int GetEtat(){ return m_State ; }
private:
    int m_State ;
} ;
int main(){
    TypeNatifCodeNatif* o1 = new TypeNatifCodeNatif(1) ;
    int i1 = o1->GetEtat() ;
    delete o1 ;
    TypeNatifCodeIL* o2 = new TypeNatifCodeIL(2) ;
    int i2 = o2->GetEtat() ;
    delete o2 ;
    TypeGereCodeIL^ o3 = gcnew TypeGereCodeIL(3) ;
    int i3 = o3->GetEtat() ;
    return 0 ;
}
```

Si vous compilez ce code et que vous analysez l'assemblage produit avec l'outil `ildasm.exe` vous vous apercevrez que le compilateur a prévu deux types valeurs gérés `TypeNatifCodeIL` et `TypeNatifCodeNatif` pour permettre d'utiliser les types natifs sous-jacent à partir du code géré. Ces deux types gérés ne présentent aucun membre. Leurs états sont stockés par les types natifs. Les types natifs sont stockés dans des sections binaires de l'assemblage qui ne sont pas visualisables à partir des outils `ildasm.exe` et `Reflector`.

On remarque la présence des quatre méthodes gérées publiques statiques `TypeNatifCodeIL.GetEtat(...)`, `TypeNatifCodeIL.{ctor}(...)`, `TypeNatifCodeNatif.GetEtat(...)` et `TypeNatifCodeNatifL.{ctor}(...)`. Il n'y a pas d'espace de noms `TypeNatifCodeNatif` ou `TypeNatifCodeIL`. Le langage IL accepte les noms de méthodes contenant un point. Il est intéressant de remarquer que les deux méthodes `.GetEtat()` prennent un paramètre représentant la référence `this`. En outre, les deux méthodes relatives à `TypeNatifCodeIL` contiennent du code IL alors que les deux méthodes relatives à `TypeNatifCodeNatif` ont le drapeaux `pinvokeimpl` indiquant qu'elle appelle une méthode native au moyen de P/Invoke.

Enfin, si vous visualisez le code IL de la méthode `main()`, vous verrez que le compilateur C++/CLI utilise les types de l'espace de noms `System.Runtime.CompilerServices` pour rendre possible la magie du code IL qui manipule des types natifs.

## *Utiliser des objets gérés à partir du code natif*

Le problème fondamental lors de l'utilisation d'un objet géré par du code natif est que vous ne pouvez pas utiliser de pointeur vers l'objet géré à moins de le fixer en mémoire. Cependant, le ramasse-miettes perd beaucoup en efficacité lorsqu'il y a des objets fixés. Il a donc fallu trouver une autre notion que celle de pointeur pour référencer les objets gérés à partir du code natif.

Le ramasse-miettes implémente un système de *pointeurs logiques*, aussi nommés *handles*. Un handle est un numéro sur quatre octets. Le ramasse-miettes fait en sorte que chaque objet géré ait son propre handle. Le code natif peut référencer un objet géré à partir de son handle.

Un handle vers une instance d'un type géré `T` est représente en C++/CLI avec l'expression `T^`. Le *framework* .NET présente la structure gérée `System.Runtime.InteropServices.GCHandle` qui permet de faire la passerelle entre les objets gérés et leurs handles. Cependant, le code natif n'a pas directement accès aux types gérés. Vous n'avez donc pas la possibilité d'utiliser la structure gérée `GCHandle()` ni à la syntaxe `T^` à partir du code natif.

Pour référencer un objet géré de type `T` à partir du code natif, vous devez utiliser une instance de `gcroot<T^>`. `gcroot<T>` est une structure native qui utilise le mécanisme de *template* de C++/CLI et dont les méthodes sont en IL. Il est instructif de regarder la définition de `gcroot<T>` dans le fichier `gcroot.h`. Vous y verrez :

- Un champ `_handle` de type `void*`. Ce champ ne peut être de type `GCHandle`. En effet, a structure `gcroot<T>` est native et ne peut avoir de champs de type géré tels que `GCHandle`.
- Les méthodes de `gcroot<T>` peuvent utiliser des instances de `GCHandle` de puisqu'elles sont compilées en IL. La passerelle entre une instance de `GCHandle` et le champ `_handle` est assurée par des opérateurs statiques de conversion de la structure `GCHandle`. Pour simplifier ces opérations, le fichier `gcroot<T>` utilise les deux macros suivantes :

```
__GCHANDLE_TO_VOIDPTR(x)
    ((GCHandle::operator System::IntPtr(x)).ToPointer())
__VOIDPTR_TO_GCHANDLE(x)
    (GCHandle::operator GCHandle(System::IntPtr(x)))
```

- Le destructeur de la structure `gcroot<T>` libère le handle. Lorsque le code natif appelle ce destructeur il signifie donc au ramasse-miettes que l'objet géré sous-jacent pourra être détruit si il n'y a plus d'autres références vers celui-ci.

Dans l'exemple suivant, la classe `TypeNatifCodeNatif` garde une référence vers une chaîne de caractère gérée. Comme vous pouvez le voir, le recours à `gcroot` est réduit au strict nécessaire et vous ne voyez pas apparaître la structure gérée `GCHandle` :

*Exemple 8-9 :*

```
// compile with: /clr
#include "stdafx.h"
#include <vcclr.h>
using namespace System ;
#pragma unmanaged
class TypeNatifCodeNatif {
public:
   TypeNatifCodeNatif( gcroot<String^> s ) {m_s = s;}
   gcroot<String^> m_s ;
} ;
#pragma managed
int main() {
   TypeNatifCodeNatif* obj = new TypeNatifCodeNatif("Bonjour") ;
   Console::WriteLine( obj->m_s ) ;
   delete obj ;
}
```

Pour exploiter un objet géré à partir du code natif vous devez développer vous-même un type natif avec du code géré qui fait la passerelle entre les deux mondes. L'exemple suivant montre comment du code natif qui détient une référence (en fait un handle) sur une instance gérée de type `string` peut invoquer la méthode `get_Length()` sur cette instance :

*Exemple 8-10 :*

```
// compile with: /clr
#include "stdafx.h"
#include <vcclr.h>
using namespace System ;
#pragma managed
class TypeNatifCodeIL {
public:
   static int Lentgh(gcroot<String^> s){
       return s->Length;
   }
};
#pragma unmanaged
class TypeNatifCodeNatif {
public:
   TypeNatifCodeNatif(gcroot<String^> s) {
       m_Length = TypeNatifCodeIL::Lentgh(s) ;
   }
```

```
   int m_Length ;
} ;
#pragma managed
int main() {
   TypeNatifCodeNatif* obj = new TypeNatifCodeNatif("Bonjour") ;
   Console::WriteLine( obj->m_Length ) ;
   delete obj ;
}
```

# .NET et les Handles win32

Les handles du ramasse-miettes présentés dans la section précédente sont conceptuellement proches des handles win32 que nous allons voir. Cependant, ces deux sortes de handles appartiennent à des domaines complètement disjoints.

## Introduction

Une application exécutée sous un systèmes *Windows* a la possibilité d'exploiter des ressources systèmes telles que les fichiers, le registre, les pilotes, les processus, les threads, les mutex, les pipes nommés, les sokets, les fenêtres etc. Une même ressource système peut être exploitée simultanément par plusieurs processus (on peut citer par exemple les mutex nommés). Une ressource système ne peut donc pas être référencée par un pointeur puisqu'elle n'appartient pas à l'espace d'adressage d'un processus. En conséquence, les processus *Windows* accèdent aux ressources systèmes par l'intermédiaire de pointeurs logiques nommés *handles*.

Toutes les fonctions win32 permettant la manipulation d'une ressource système acceptent un paramètre en entré de type `HANDLE`. Un handle est un numéro codé sur quatre octets. Chaque processus *Windows* maintient en interne une table d'association entre les handles et les ressources systèmes utilisées. Deux ressources systèmes ne peuvent donc pas être référencée par le même handle au sein d'un processus. Deux handles référençant une même ressource à partir de deux processus distincts peuvent être deux entiers différents.

Une ressource système est créée lors de l'appel de certaines fonctions win32 telles que `CreateFile()`, `CreateMutex()` ou `CreateThread()`. Ces fonctions ont la particularité de retourner un handle. L'appel à une de ces fonctions ne crée pas nécessairement une ressource. Par exemple vous pouvez récupérer un handle vers un mutex existant en appelant la fonction `CreateMutex()` paramétrée avec le nom du mutex. La fonction win32 `CloseHandle()` permet de signifier à *Windows* que le processus appelant n'a plus besoin de la ressource système référencée. *Windows* détruit une ressource système lorsque plus aucun processus ne maintient de handle vers elle.

Le compteur de performance *Windows 'Processus/Nombre de handles'* vous permet de connaître le nombre de handles couramment détenus par un processus. La colonne *'Handles'* du gestionnaire des tâches vous permet aussi de visualiser ce nombre en temps réel.

## La classe HandleCollector

Une instance de la classe `System.Runtime.InteropServices.HandleCollector` permet de fournir au CLR une estimation du nombre de handles actuellement détenu par le processus. Vous

pouvez préciser un seuil initial et un seuil maximum à partir duquel le ramasse-miettes lancera une collecte. En effet, le ramasse-miettes n'a aucune connaissance de la quantité de mémoire non gérée maintenue par des ressources systèmes et cette classe permet de pallier cette faiblesse. Une instance de `HandleCollector` peut être nommée de façon à n'être concernée que par les handles vers un certain type de ressource.

### *Les classes SafeHandle et CriticalHandle*

Avant la version 2.0 de .NET, un handle était référencé par une instance du type `IntPtr`. Cela posait plusieurs problèmes :

- Il n'y avait pas de vérification de type à la compilation. Par exemple, rien ne vous empêchez de communiquer un handle sur une fenêtre (de type win32 `HWND`) à une fonction win32 qui a besoin d'un handle sur un fichier (de type win32 `HFILE`).
- Vous n'aviez pas de garanties quant à la libération d'un handle. Une exception de type `ThreadAbortException` ou `OutOfMemory` pouvait compromettre cette opération.
- Vous étiez exposé à une *situation de compétition* (*race condition*) quant à la fermeture du handle. Rien n'empêchait un thread d'utiliser un handle pendant qu'un autre thread était en train de le fermer. Cela pouvait même mener à un problème de sécurité connu sous le nom de *handle-recycling attack* où du code malveillant exploitait une ressource en train d'être fermée.

Le *framework* .NET 2.0 offre un moyen de pallier ces problèmes avec les classes abstraites `System.InteropServices.CriticalHandle` et `System.InteropServices.SafeHandle`. L'idée est de prévoir une classe non abstraite dérivée d'une de ces classes pour gérer le cycle de vie d'un type de handle. Cette classe doit être dérivée de `SafeHandle` pour implémenter un type de handle qui supporte un compteur de références. Sinon cette classe doit dériver de `CriticalHandle`. Les deux classes `SafeHandle` et `CriticalHandle` dérivent de la classe `System.Runtime.ConstrainedExecution.CriticalFinalizer`. Et ont donc un finaliseur critique. Les finaliseurs critiques sont décrits en page 129. De ce fait, on obtient certaine garantie de fiabilité sur l'opération de fermeture d'un handle.

Vous pouvez aussi dériver des classes abstraites `CriticalHandleMinusOneIsInvalid`, `SafeHandleMinusOneIsInvalid` et `SafeHandleZeroOrMinusOneIsInvalid` dont les noms sont éloquents quant aux services de durée de vie offerts. Ces classes sont dans l'espace de noms `Microsoft.Win32.SafeHandles`.

## *Utilisation de COM à partir de .NET*

Avec le temps, la technologie COM est devenue de plus en plus compliquée à utiliser et à comprendre, à tel point que certaines entreprises se sont spécialisées dans le développement de composants COM. Au début du projet .NET, *Microsoft* avait le souhait de construire une nouvelle technologie en s'affranchissant des contraintes de compatibilité ascendante avec les technologies existantes. Cependant, *Microsoft* ayant préconisé l'utilisation de la technologie COM depuis plusieurs années, il existe maintenant des millions de composants COM à travers le monde. Il n'était pas envisageable d'obliger les entreprises à re-développer tous leurs composants sous .NET, et il était obligatoire de ne pas construire .NET au-dessus de COM. La seule solution était de construire un ensemble de classes et d'outils qui permettraient d'utiliser les composants COM sans les recompiler, à partir du code développé sous .NET. C'est ce que nous allons exposer.

## *Métadonnées de types et bibliothèque de types*

Bien que répondant au même besoin, la création de liens précoces avec les classes COM et la création de liens précoces avec les classes .NET sont techniquement parlant, très différents. **La différence principale est que le code géré .NET a besoin de consulter les métadonnées de types avant de créer un lien avec une méthode d'une classe .NET** (que le lien soit précoce ou tardif). Cela vient du fait que le code IL utilise la notion de jeton de métadonnées à la place de la notion d'adresse. Voici quelques autres différences :

- Les classes COM ne se manipulent qu'au travers d'interfaces COM.
- Le passage des arguments ne se fait pas de la même manière.
- Les classes COM n'admettent pas de constructeurs avec paramètres.

Les *composants COM* (i.e les équivalents dans la technologie COM des assemblages, des DLLs en général mais aussi des exécutables) présentent optionnellement des métadonnées, contenues dans ce que l'on appelle une *bibliothèque de types* (*type library* en anglais). Une bibliothèque de types peut être contenue directement dans le composant COM ou dans un fichier à part, d'extension `tlb`. En plus du fait qu'un composant COM n'a pas obligatoirement une bibliothèque de types, le formatage binaire des métadonnées au sein des bibliothèques de types est totalement différent du formatage binaire des métadonnées dans les assemblages .NET.

## *Assemblages inter-opérables et classes Runtime Callable Wrapper*

*Microsoft* a créé l'outil *type library importer* (`tlbimp.exe`) pour construire un assemblage .NET directement à partir d'un composant COM dont la bibliothèque de types est disponible. Un assemblage créé à partir de `tlbimp.exe` est qualifié d'*assemblage interopérable* (*interop assembly* en anglais). Nous verrons comment utiliser à partir de .NET des classes COM d'un composant COM dont la bibliothèque de types n'est pas accessible, dans la prochaine section. À la fin de la présente section, nous verrons que l'utilisation de *Visual Studio* peut éviter d'avoir à manipuler l'outil `tlbimp.exe`.

Voici un fichier *IDL* (*Interface Definition Langage*) qui décrit un composant COM nommé `COMComposant`. Il est clair que ce composant COM admet une bibliothèque de types qui contient la classe COM `CClassCOM`. Cette classe admet une interface propriétaire qui s'appelle `IClasseCOM`. Cette interface présente la fonction `CalcSomme()` qui calcule la somme de ses deux premiers paramètres et la retourne dans le troisième paramètre, qui est un paramètre de sortie.

*Exemple :* *`COMComposant.idl`*

```
[
   object,
   uuid(947469B1-61EB-4010-AE29-8380C2D577E9),
   dual,
   helpstring("IClasseCOM Interface"),
   pointer_default(unique)
]
interface IClasseCOM : IDispatch{
   HRESULT  CalcSomme([in]int a, [in]int b, [out,retval] int *pResult) ;
} ;
```

```
[ version(1.0),
  uuid(8A4F713C-910F-4EE0-8A26-B870FBE58596),
  custom(a817e7a1-43fa-11d0-9e44-00aa00b6770a,
  "{23391CA9-529E-43CF-9BF6-C636BF96BF26}"),
  helpstring("COMComposant 1.0 Type Library") ]
library COMComposant {
   importlib("stdole2.tlb") ;
   importlib("olepro32.dll") ;
   [
      version(1.0),
      uuid(1E3B6413-7E63-42B5-874D-E0A27A42190C),
      helpstring("CClasseCOM Class")
   ]
   coclass CClasseCOM{
      interface IClasseCOM;
   };
}
```

Voici la ligne de commande à utiliser pour obtenir un assemblage interopérable à partir de ce composant COM. `tlbimp.exe` nous offre la possibilité d'ajouter un espace de noms avec la directive `/namespace`. Cet outil présente de nombreuses autres directives décrites dans les **MSDN** à l'article **Type Library Importer (tlbimp.exe)**.

```
>tlbimp.exe COMComposant.dll /out:AsmCOMComposant.dll
    /namespace:TestInterop
```

Si on analyse l'assemblage `AsmCOMComposant.dll` avec `ildasm.exe`, on s'aperçoit qu'il contient entre autres une classe nommée `CClasseCOMClass` et une interface nommée `IClassCOM`. L'interface `IClassCOM` présente la méthode `CalcSomme()` et la classe `CClasseCOMClass` implémente l'interface `IClassCOM`.

L'implémentation de la méthode `CalcSomme()` dans la classe `CClasseCOMClass` ne contient que le code pour transformer l'appel d'une méthode d'une type .NET, en un appel vers la méthode COM `IClasseCOM.CalcSomme()`. Dans le jargon .NET on dit que la classe .NET `CClasseCOMClass` est une classe *Runtime Callable Wrapper* (*RCW*). En terminologie *design patterns* (Gof), on dit d'une telle classe qu'elle est un *adaptateur*, car elle adapte l'interface de l'objet COM à une interface que le client peut appeler. On peut aussi dire que `CClasseCOMClass` est une classe *proxy*, dans le sens ou tous les appels sont d'abord interceptés, pour effectuer un traitement sur les arguments. Après interception, les appels sont effectués directement sur la classe COM sous-jacente.

Voici le code d'un programme C# qui utilise l'assemblage interopérable `AsmCOMComposant.dll`. Naturellement il faut qu'à la compilation de ce programme, on précise une référence vers `AsmCOMComposant.dll` :

*Exemple 8-11 :* *DotNETClient.cs*

```
using TestInterop ;
class Program {
   static void Main() {
      IClasseCOM foo = new CClasseCOMClass() ;
      int result = foo.CalcSomme(2, 3) ;
```

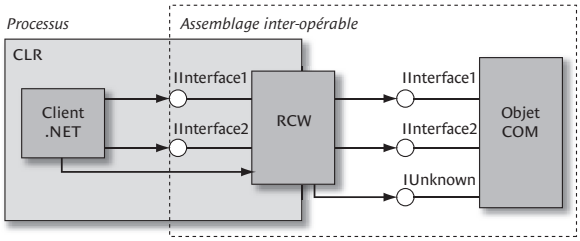


*Figure 8-1 : Utilisation d'une classe COM à partir d'un client .NET*

```
System.Console.WriteLine("Résultat :{0}", result) ;
   }
}
```

Notez qu'à la différence d'une classe COM, une classe RCW peut être directement appelée, sans passer par aucune interface. On aurait pu écrire :

```
...
      CClasseCOMClass  foo = new CClasseCOMClass() ;
      int Result = foo.CalcSomme(2,3) ;
...
```

Si le classe COM supporte plusieurs interfaces propriétaires, `tlbimp.exe` génèrera autant d'interfaces .NET. En outre, toutes les méthodes de ces interfaces seront déclarées et accessibles publiquement dans la classe RCW.

L'utilisation de *Visual Studio* peut éviter d'avoir à manipuler l'outil `tlbimp.exe`. Il suffit d'ajouter une référence au projet directement vers le composant COM, exactement comme si vous rajoutiez une référence vers un autre assemblage (voir la Figure 8-2). L'assemblage interopérable contenant les classes RCW sera automatiquement construit et placé dans le même répertoire que l'assemblage concerné.

Les assemblages interopérables et les composants COM d'une application .NET font bien évidemment partie des fichiers à déployer avec votre application .NET.

## *Accéder aux classes d'un composant COM sans utiliser de bibliothèque de types*

Le *framework* .NET présente la possibilité de créer ses propres classes et interfaces RCW pour accéder aux instances d'une classe COM sans utiliser la bibliothèque de types et l'outil `tlbimp.exe`. Cette fonctionnalité est essentiellement utilisée lorsqu'un composant COM n'a pas de bibliothèque de types, ou lorsqu'un composant COM a une bibliothèque de types non disponible au moment du développement.

Cette fonctionnalité est assez fastidieuse à utiliser. Il faut redéfinir entièrement chaque interface COM et chaque classe COM que vous utilisez. Pour chaque argument et chaque valeur de retour de chaque méthode il faut utiliser l'attribut `System.Runtime.InteropServices.MarshalAs`. Pour le composant COM de la page 279, il faudrait écrire :

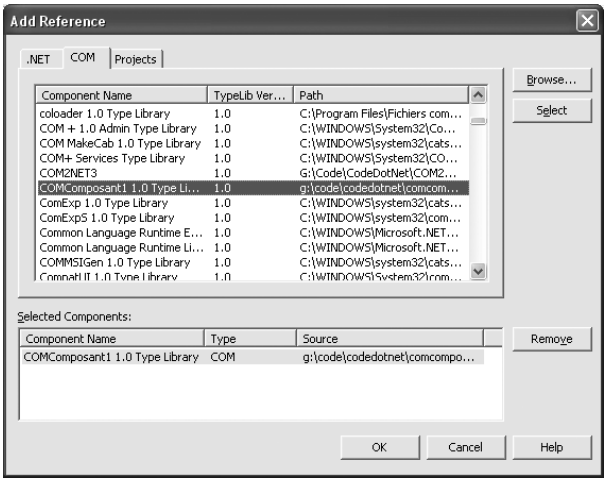


*Figure 8-2 : Ajout d'une référence vers un composant COM à partir de Visual Studio .NET*

*Exemple 8-12 :*

```
using System ;
using System.Runtime.InteropServices ;
[
   ComImport,
   Guid("947469B1-61EB-4010-AE29-8380C2D577E9"),
   InterfaceType(ComInterfaceType.InterfaceIsDual)
]
public interface IClasseCOM {
   [return : MarshalAs(UnmanagedType.I4)]
   int CalcSomme(
      [In,MarshalAs(UnmanagedType.I4)] int a,
      [In,MarshalAs(UnmanagedType.I4)] int b,
      [Out,MarshalAs(UnmanagedType.I4)]out int c) ;
}
[
   ComImport,
   Guid("1E3B6413-7E63-42B5-874D-E0A27A42190C")
]
public class CClasseCOM {}
class Program {
   static void Main() {
      IClasseCOM foo = new CClasseCOM () as IClasseCOM ;
      int result ;
      foo.CalcSomme(2, 3, out result) ;
      System.Console.WriteLine("Résultat :{0}", result) ;
   }
}
```

# *Import d'un ActiveX avec Visual Studio .NET*

Un *ActiveX* est le terme employé pour décrire une classe COM graphique. On peut aussi utiliser le terme *contrôle ActiveX* ou le terme *OCX*. Les contrôles ActiveX sont stockés dans des composants COM d'extension `.dll` ou d'extension `.ocx`. Il existe un très grand nombre de contrôles ActiveX disponibles gratuitement ou non sur internet. Il existe des OCX pour visionner des images, des animations, des OCX pour parcourir les répertoires, pour éditer des numéros de téléphones etc. De plus chaque technologie graphique (*Flash* de *Macromedia*, *MapPoint* de *Microsoft* etc) est en général fournie avec un contrôle ActiveX permettant de l'utiliser. Comme vous le voyez il existe un très vaste choix de contrôles ActiveX. La bonne nouvelle est que *Visual Studio* permet de les utiliser dans vos programmes .NET aussi simplement que s'ils étaient des contrôles graphiques .NET. Dans le chapitre consacré à la technologie *Windows Form* nous exposons comment utiliser des contrôles graphiques .NET mais intéressons-nous ici à l'import de contrôles ActiveX dans vos formulaires .NET grâce à *Visual Studio*.

Il est préférable de préparer le terrain, et d'insérer un onglet spécial pour nos imports de contrôles ActiveX dans la boîte à outils (*toolbox*) des composants graphiques de *Visual Studio*. Pour cela cliquez droit sur cette boite à outils et choisissez le menu « *Ajouter un onglet* » (*Add Tab*) et donnez un nom à votre nouvel onglet (par exemple « *ActiveX* »). Votre boîte à outils doit ressembler maintenant à celle de la Figure 8-3.

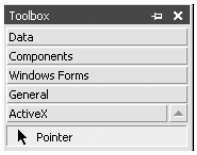


*Figure 8-3 : Préparation de la boîte à outils pour importer des OCX*

Vous pouvez maintenant ajouter un contrôle ActiveX en cliquant droit sur l'onglet « *ActiveX* » et en choisissant le menu « *Personnaliser la boîte à outils* » (*Customize Toolbox*...). Vous arrivez sur la fenêtre de la Figure 8-4 qui vous permet d'ajouter soit un contrôle ActiveX soit un contrôle graphique .NET dans votre onglet.

Ici nous avons choisi le contrôle ActiveX *Microsoft Rich Textbox Control* qui permet d'afficher un document au format RTF. Lorsque nous ajoutons un tel contrôle dans une *WinForm*, nous avons accès aux fenêtres de propriétés de ce contrôle. En interne, un assemblage interopérable a été créé, contenant les classes RCW adéquates.

# *Spécificités de COM à prendre en compte en manipulant une classe RCW*

## *Gestion du cycle de vie d'un objet COM à partir de .NET*

Le cycle de vie d'un objet COM suit des règles très précises qui sont complètement encapsulées dans les classes RCW. Notamment les appels à la fonction `CoCreateInstance()`, qui permet de créer un objet COM, sont encapsulés. Les appels aux méthodes de l'interface `IUnknow` (`QueryInterface()` qui permet de naviguer entre les interfaces, `AddRef()` et `Release()` qui permettent de gérer le compteur interne de références) sont aussi encapsulés.

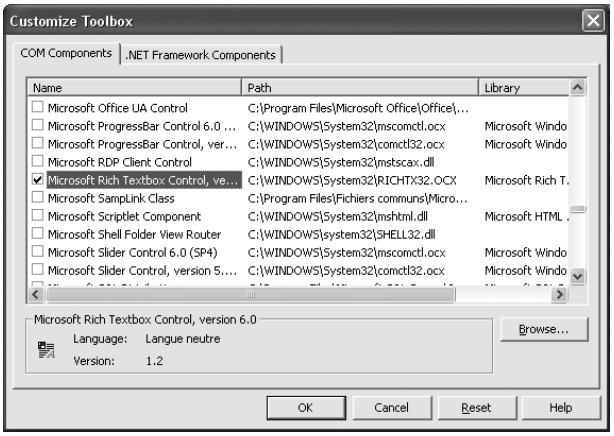


*Figure 8-4 : Ajout d'un OCX dans la boîte à outils*

Une classe RCW est une classe gérée. Vous ne pouvez décider exactement quand elle sera détruite, puisque cette responsabilité incombe au ramasse-miettes. Or, une classe RCW ne détruit son objet COM sous-jacent que lorsqu'elle est elle-même détruite. *A priori*, vous ne pouvez donc pas décider du moment de la destruction d'un objet COM dans vos programmes .NET. Cette situation est gênante car en général, l'architecture d'une classe COM prend en compte le fait que le client peut choisir le moment de la destruction des instances de cette classe COM.

La classe `System.Marshal.InteropServices.Marshal` présente la méthode `int ReleaseComObject(object)` pour pallier ce problème. Cette méthode prend en paramètre un `object` qui doit être l'objet RCW concerné. En interne, un appel à cette méthode provoque l'appel à la méthode `Release()` de l'objet COM et ne fait donc que décrémenter d'une unité le compteur interne de référence de l'objet COM. La nouvelle valeur de ce compteur interne est retournée, et vous pouvez donc tester si l'objet COM est effectivement détruit en vérifiant que cette valeur vaut bien 0. Une fois qu'un objet COM est détruit, nous vous conseillons de mettre immédiatement à nul les références vers l'objet RCW concerné, car toute utilisation de cet objet provoque l'envoi de l'exception `InvalidComObjectException`. Par exemple :

```
...
   IClasseCOM  foo = new CClasseCOMClass() ;
   int result = foo.CalcSomme(2,3) ;
   System.Runtime.InteropServices.Marshal.ReleaseComObject(foo);
   foo = null;
...
```

## *Gestion des types de données COM à partir de .NET*

La conversion entre les types de données COM et les types de données .NET ne pose pas de problème. Il est néanmoins utile de préciser les conversions suivantes :

- Les classes RCW convertissent les *Basic String* (*BSTR*) de COM en des instances de la classe `System.String`.

- Les classes RCW convertissent les *VARIANT* de COM en des instances d'une classe dérivée de la classe `object`. La classe .NET de l'objet sous-jacent dépend naturellement du type sous-jacent du VARIANT et vous pouvez transtyper cet objet .NET avec l'opérateur `as` de C#.
- Les classes RCW convertissent les *SAFEARRAY* de COM en des tableaux gérés du type adéquat. Par exemple un argument de type SAFEARRAY(BSTR) dans une méthode COM devient un argument de type `System.String[]` dans la méthode de la classe RCW.

## *Gestion des erreurs COM à partir de .NET*

La gestion des erreurs est radicalement différente entre .NET et COM :

- .NET gère les erreurs au moyen d'exceptions, gérées par le CLR.
- COM gère les erreurs au moyen de la valeur de retour de chaque méthode de chaque interface COM. Cette valeur de retour est toujours de type *`HRESULT`*. Un `HRESULT` est une valeur codée sur quatre octets qui précise si une erreur s'est produite lors de l'appel d'une méthode sur un objet COM. Le cas échéant, le `HRESULT` contient le type de l'erreur et éventuellement des informations sur la couche logicielle qui a généré l'erreur.

Si l'appel à une méthode d'un objet COM renvoie un `HRESULT` d'erreur, une exception de type `COMException` est automatiquement levée par le CLR. La propriété `ErrorCode` de cette exception contient la valeur du `HRESULT`.

## *Gestion des apartments COM à partir de .NET*

COM permet de gérer les affinités entre les objets COM et les threads au moyen d'*apartments COM*. Concrètement, vous pouvez vous assurer que les méthodes des instances d'une classe COM ne peuvent être exécutées que par certains threads. Un processus peut contenir plusieurs apartments COM. Chaque objet COM est contenu dans un processus et appartient à un apartment COM. Chaque thread du processus appartient à un apartment COM. Si un thread d'un apartment COM doit effectuer l'appel d'une méthode d'un objet COM qui n'est pas dans le même apartment COM (voire dans un autre processus), l'interface COM concernée doit être « marshallée » de façon à pouvoir être utilisée par le thread client. Cette opération de « marshalling » est nécessaire car le code des méthodes de la classe d'un objet COM est toujours exécuté par le ou les threads de l'apartment COM de l'objet.

Il existe deux sortes d'apartment COM les *Single Thread Apartment* (*STA*) et les *Multi Thread Apartment* (*MTA*) qui contiennent respectivement un thread ou plusieurs threads. Un processus peut contenir plusieurs STA mais ne peut contenir plus d'un MTA. Durant l'évolution de la technologie COM, la notion de STA est apparue avant la notion de MTA. Le but de STA était de décharger les développeurs du souci des accès concurrents aux ressources dans une classe COM. En effet, dans le mode STA, c'est le même thread qui gère les différents appels aux différentes instances d'une même classe COM. La gestion STA s'est révélée insuffisante pour certaines applications et il a fallu introduire la notion de MTA, qui a redonnée au développeur la responsabilité de gérer les accès concurrents aux ressources.

Les threads gérés de .NET connaissent cette notion d'apartment COM. Plus exactement, un thread géré sait si son thread *Windows* sous-jacent est STA ou MTA. Vous pouvez positionner la propriété `ApartmentState` d'une instance de la classe `Thread` avec une des valeurs de l'énumération `System.Threading.ApartmentState`, à savoir `STA`, `MTA` ou `Unknown` (qui est la valeur prise par défaut). Cette propriété ne peut être modifiée qu'une seule fois dans l'existence d'un thread géré.

Vous pouvez aussi spécifier l'apartment COM d'un thread en utilisant l'un des attributs `System.STAThread` ou `System.MTAThread` sur la méthode qui constitue le point d'entrée du thread. Par exemple :

```
...
[MTAThread]
public static void Main(){
}
...
```

## *Lien tardif explicite sur les classes COM*

Bien avant la venue de .NET et de son mécanisme de réflexion, la technique de lien tardif explicite était disponible sur les classes COM dont les interfaces étendaient l'interface `IDispatch`. `IDispatch` présente notamment la méthode `GetDispID()` qui permet d'avoir la position de la déclaration d'une méthode dans une interface, en fonction du nom de la méthode. Cette position s'appelle `DISPID`. `IDispatch` présente aussi la méthode `Invoke()` qui permet d'appeler une méthode en connaissant son `DISPID`. L'utilisation de ces deux méthodes permet respectivement de créer des liens tardifs explicites avec des classes COM et d'invoquer la méthode avec un tel lien.

Vous pouvez exploiter le mécanisme de liens tardifs explicites `IDisptach` de COM, à partir du mécanisme de liens tardifs explicites de .NET. Pour utiliser cette possibilité avec la classe du composant COM de la page 279 il faudrait écrire :

*Exemple 8-13 :*

```
using System ;
using System.Reflection ;
class Program {
   static void Main() {
      Type type = Type.GetTypeFromProgID("COMComposant1.CClasseCOM");
      // On peut aussi écrire :
      // Type type = Type.GetTypeFromCLSID(
      //    new System.Guid("1E3B6413-7E63-42B5-874D-E0A27A42190C")) ;
      Object obj = Activator.CreateInstance(type) ;
      Object[] args = new Object[] { 3, 4 } ;
      int retVal = (int) type.InvokeMember(
         "CalcSomme",
         BindingFlags.InvokeMethod,
         null,
         obj,
         args) ;
      Console.WriteLine(retVal) ;
   }
}
```

Notez l'obtention du type à partir de son PROGID ou de son CLSID, grâce à l'une des méthodes statiques de la classe `Type`, `GetTypeFromProgID()` ou `GetTypeFromCLSID()`.

## *Utiliser une classes COM sans l'enregistrer dans le registre*

*Windows XP* ainsi que les versions postérieures présentent la possibilité d'utiliser une classe COM sans l'enregistrer dans la base de registres. L'idée est de fournir un fichier qui contient l'association entre les CLSIDs et les DLLs COM des classes COM utilisées pour chaque application. Lors du lancement de l'application, *Windows XP* charge les données de cette table. Durant l'exécution, lorsqu'une classe COM doit être chargée, *Windows XP* consulte au préalable cette table. Si la classe COM n'est pas trouvée par cette technique, le processus classique de localisation du composant COM contenant la classe par consultation de la base des registre est alors enclenché.

Vous pouvez exploiter cette technique nommée *registration free COM* (ou *reg free COM*) à partir d'un assemblage .NET. Pour cela, il suffit de fournir un fichier ayant le même nom que l'assemblage avec l'extension `.manifest` contenant les données d'association entre les CLSIDs et les DLLs COM au format XML comme ceci (`MyAsm.exe` est le nom de l'assemblage en l'occurrence, donc le nom du fichier est `MyAsm.exe.manifest`) :

*Exemple :* *`MyAsm.exe.manifest`*

```
<?xml version="1.0" encoding="utf-8"?>
<assembly xsi:schemaLocation="urn:schemas-microsoft-com:asm.v1
                              assembly.adaptive.xsd"
          manifestVersion="1.0"
          xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ... >
  <assemblyIdentity name="MyAsm.exe" version="1.0.0.0" type="win32" />
  <file name="[Chemin relatif ou absolu]/MyCOMComponent.dll"
        asmv2:size="20480">
    <typelib  tlbid="{ea995c49-e5d0-4f1f-8489-31239fc9d9d0}"
              version="2.0" helpdir=""
              resourceid="0" flags="HASDISKIMAGE" />
    <comClass clsid="{97b5534f-3b96-40a4-88b8-19a3bf4eeb2e}"
              threadingModel="Apartment"
              tlbid="{ea995c49-e5d0-4f1f-8489-31239fc9d9d0}"
              progid="MyCOMComponent.MyClass" />
    <comClass ... />
  </file>
  <file ... />
  ...
```

- *Visual Studio 2005* permet d'exploiter simplement cette technique. Pour cela, il faut que vous positionniez l'attribut `Isolated` d'une référence COM à `true`. Cela fonctionne aussi si la référence COM est un OCX. La compilation du projet fait alors en sorte de créer le fichier d'extension `.manifest` dans le répertoire de sortie. Elle copie aussi les DLLs composants COM dans ce répertoire. *Reg free COM* est particulièrement utile si vous désirez avoir recours à des classes COM dans un projet déployé à la *XCopy*, par exemple avec la technologie *ClickOnce*.
- Précisons que pour exploiter cet attribut il faut que les classes COM soient enregistrées dans la base des registres de la machine qui réalise la compilation. En outre, il vaut mieux tester ce genre d'application sur une machine vierge. En effet, une utilisation défectueuse de *reg free COM* ne serait pas détectée sur une machine sur laquelle les composants COM utilisés sont enregistrés.

Sachez que vous pouvez vous passer de *reg free COM* en effectuant le travail de localisation, de chargement et d'instanciation de la classe COM vous-même. Cette technique est utile si votre déploiement s'effectue potentiellement sur des versions antérieures à *Windows XP*. Pour chaque classe et composant COM il faut :

- Localiser la DLL composant COM.
- La charger en mémoire avec la fonction *win32* `LoadLibrary()`.
- Localiser la fonction `DllGetClassObject()` exportée par la DLL composant COM. Pour cela, il faut avoir recours à la fonction *win32* `GetProcAddress()`.
- Appeler cette fonction `DllGetClassObject()` pour obtenir un objet COM implémentant l'interface COM `IClassFactory`.
- Appeler la méthode `IClassFactory.CreateInstance()` pour créer un objet COM.

# *Encapsuler une classe .NET dans une classe COM*

## *Notion de CCW*

Dans un processus de migration d'une application vers .NET, il peut arriver que certains composants soient migrés avant d'autres. Il se peut que du code non géré doive instancier et utiliser une classe .NET. À cette fin, le *framework* .NET nous permet d'encapsuler une classe .NET dans un *COM Callable Wrapper* (*CCW*). Un CCW est un objet COM créé et géré automatiquement par le CLR. Un tel objet COM encapsule un objet .NET, pour faire en sorte que ce dernier puissent être accessible comme un objet COM. Un CCW est créé durant l'exécution par le CLR, à partir des métadonnées de la classe .NET (et non à partir d'une quelconque bibliothèque de types COM). Il existe au plus un CCW par objet .NET, quel que soit le nombre de clients souhaitant l'utiliser comme un objet COM. Un objet .NET ayant un CCW peut aussi être utilisé par des clients .NET. Dans ce cas, le fait que l'objet .NET ait un CCW est complètement transparent pour les clients .NET. La Figure 8-5 résume l'interaction entre le client non géré et l'objet .NET au moyen d'un CCW.

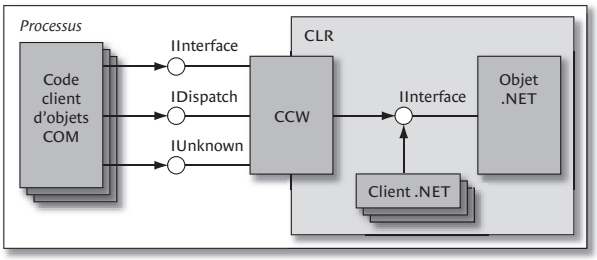


*Figure 8-5 : Utilisation d'un objet .NET par l'intermédiaire d'un CCW*

Il est important de noter que le modèle objet proposé par COM est assez restrictif par rapport au modèle objet proposé par .NET. Pour être utilisée comme une classe COM normale par l'intermédiaire d'un CCW, une classe .NET doit satisfaire les contraintes suivantes :

- La classe .NET doit avoir un constructeur sans arguments, appelé aussi constructeur par défaut. En effet, COM ne supporte pas la notion de constructeur avec arguments. Parmi les constructeurs de la classe .NET, seul ce constructeur sans argument pourra être appelé par l'intermédiaire du CCW.
- Seules les classes publiques d'un assemblage peuvent être encapsulées dans un CCW.
- La classe .NET peut avoir des membres statiques, mais ils ne seront pas utilisables par l'intermédiaire d'un CCW puisque COM ne supporte pas cette notion de membres statiques.
- Les surcharges d'une méthode de la classe .NET seront renommées, car COM ne supporte pas cette notion. Concrètement, un blanc souligné est mis avant le nom de chaque méthode surchargée, et un numéro est mis à la fin du nom de la méthode.

## *Produire une bibliothèque de types COM décrivant les classes CCW à partir d'un assemblage .NET*

L'outil *Type Library Exporter* `tlbexp.exe` permet de fabriquer une bibliothèque de types COM décrivant les classes et interfaces .NET publiques contenues dans un assemblage. Montrons comment utiliser `tlbexp.exe` au moyen d'un exemple simple. Voici du code C# :

*Exemple 8-14 :* *dotNET2COM.cs*

```
namespace Test {
   public interface ICalc {
      int CalcSomme(int a, int b) ;
   }
   public class CCalc : ICalc {
      public int CalcSomme(int a, int b) {
         return a + b ;
      }
   }
}
```

Pour produire l'assemblage `dotNET2COM.dll` à partir de ce fichier C#, il suffit de taper la ligne de commande suivante :

```
>csc.exe -t:library dotNET2COM.cs
```

Pour produire une bibliothèque de types COM décrivant le CCW qui encapsule la classe `CCalc` et l'interface `ICalc`, il suffit de taper la ligne de commande suivante :

```
>tlbexp.exe dotNET2COM.dll /out:dotNET2COM.tlb
```

Comprenez bien que le CCW est produit à l'exécution par le CLR. Durant cette opération le CLR n'a pas besoin d'une bibliothèque de types. La bibliothèque de type n'est utile que pour les clients qui souhaitent effectuer un lien précoce COM avec le CCW.

Nous pouvons visualiser la bibliothèque de types `dotNET2COM.tlb` au moyen de l'outil *OLE Viewer* (`oleview.exe`). Cet outil est accessible par le menu *Outils* de Visual Studio .NET. Il est aussi accessible en ligne de commande. *OLE Viewer* comporte le menu *Fichier* ► *Visualiser une bibliothèque de types...* . Rappelons que les bibliothèques de types sont décrites dans un format binaire, et qu'il est donc nécessaire de disposer d'un tel outil pour les visualiser :

```
[
     uuid(824A9F5F-39AE-35BB-8741-7C4C8F7DB26C),
     version(1.0),
     custom(90883F05-3D28-11D2-8F17-00A0C9A6186D,
   dotNET2COM,
   Version=0.0.0.0,
   Culture=neutral,
   PublicKeyToken=null)
]
library dotNET2COM{
    importlib("mscorlib.tlb") ;
    importlib("stdole2.tlb") ;

    // Forward declare all types defined in this typelib
    interface ICalc ;
    interface _CCalc ;

    [
      odl,
      uuid(72E3095D-B243-37F0-A95A-41ABBE13E029),
      version(1.0),
      dual,
      oleautomation,
      custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, Test.ICalc)
    ]
    interface ICalc : IDispatch {
        [id(0x60020000)]
        HRESULT CalcSomme(
                        [in] long a,
                        [in] long b,
                        [out, retval] long* pRetVal);
    };

    [
      uuid(A51C81A3-4892-39EC-981A-AF77FB4CFD36),
      version(1.0),
      custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, Test.CCalc)
    ]
    coclass CCalc {
        [default] interface _CCalc;
        interface _Object;
        interface ICalc;
    };

    [
      odl,
      uuid(84181003-CCB9-3219-B373-629AF4E3B246),
      hidden,
      dual,
```

```
      oleautomation,
      custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, Test.CCalc)
    ]
    interface _CCalc : IDispatch {
    } ;
} ;
```

Plusieurs remarques peuvent être faites :

- La classe COM `CCalc` implémente les interfaces COM suivantes :
  - `ICalc` : Cette interface COM représente l'interface .NET `ICalc`.
  - `_Object` : Cette interface représente les méthodes de la classe `System.Object` dont dérive toutes les classes .NET. L'interface `_Object` est définie dans la bibliothèque de types `mscorlib.tlb`.
  - `_CCalc` : Cette interface COM est l'*interface de classe* (*class interface* en anglais) de la classe .NET `CCalc`. Une interface de classe est sensée présenter tous les membres publiques non statiques d'une classe .NET (y compris les membres des types dont la classe est dérivée). Cependant, dans notre exemple cette interface de classe ne contient aucun membre. L'utilisation d'interfaces de classes étant déconseillée, le comportement par défaut de `tlbexp.exe` est de ne pas insérer de membres dans une interface de classe qu'il produit. L'utilisation d'interfaces de classes est déconseillée car elle couple les clients non gérés avec des classes gérées et non avec des interfaces. Vous pouvez néanmoins préciser à `tlbexp.exe` que vous souhaitez que l'interface de classe soit correctement construite en utilisant dans votre code C# l'attribut `System.Runtime.InteropServices.ClassInterface` avec la valeur `AutoDual`. Par exemple :

```
...
   [ClassInterface(ClassInterfaceType.AutoDual)]
   public class CCalc : ICalc{
      ...
   }
...
```

  - Les champs publics d'une classe admettant une interface de classe, sont matérialisés dans l'interface de classe sous forme de deux accesseurs. De même, les accesseurs des propriétés des interfaces publiques, sont présentés comme des méthodes dans les interfaces COM générées par `tlbexp.exe`.
- Une classe ID a été produite automatiquement pour la classe COM `CCalc`. On aurait pu utiliser l'attribut `System.RunTime.InteropServices.Guid` pour spécifier notre propre attribut dans le code C# :

```
...
   [Guid("A51C81A3-4892-39EC-981A-AF77FB4CFD36")]
   public class CCalc : ICalc{
      ...
   }
...
```

- Une interface ID a été produite automatiquement pour l'interface COM `ICalc`. Là aussi, on aurait pu spécifier notre propre interface ID avec l'attribut `Guid`.
- L'interface COM `ICalc` étend l'interface COM `IDispatch` qui permet de créer des liens tardifs explicites sur les classes COM.

Vous avez la possibilité d'utiliser dans votre code .NET l'attribut .NET `System.RunTime.InteropServices.COMVisible` sur une interface, une classe, une méthode ou un événement afin d'indiquer à `tlbexp.exe` que l'entité concernée ne soit pas visible dans la bibliothèque de types construite. Par exemple :

```
...
   [ComVisible(false)]
   public class CCalc : ICalc{
      ...
   }
...
```

Les structures publiques sont prises en compte par `tlbexp.exe`. Elles pourront ainsi être passées comme arguments des méthodes des classes et des interfaces.

## *Enregistrer les CCW sur un système d'exploitation Microsoft*

Nous avons vu dans la section précédente comment fabriquer une bibliothèque de types COM à partir d'un assemblage .NET. Nous allons voir maintenant comment enregistrer cette bibliothèque de types dans la base des registres d'un système d'exploitation *Microsoft*. Rappelons que pour être utilisable, une classe COM doit impérativement être enregistrée dans la base des registres. C'est principalement l'information d'association entre le classe ID de la classe COM et le fichier (DLL ou exécutable) contenant effectivement l'implémentation de la classe, qui est sauvegardée dans la base des registres. En effet, une application qui instancie une classe COM ne connaît la classe COM que par son classe ID et n'a aucune information quant au fichier qui contient l'implémentation de la classe. Il faut donc aller consulter la base des registres pour localiser l'implémentation d'une classe COM, au moins la première fois que celle-ci est instanciée dans une application.

Pour enregistrer (ou « dés-enregistrer ») les classes COM d'un composant COM classique, il suffit d'utiliser l'outil `regsvr32.exe`. Pour enregistrer (ou « désenregistrer ») les classes .NET d'un assemblage comme des classes COM, il faut utiliser l'outil *Assembly Registration Tool* `regasm.exe` spécialement conçu pour cette tâche. Par exemple :

```
>regasm.exe dotNET2COM.dll
```

Cet outil est paramétrable, avec plusieurs options décrites dans les **MSDN** à l'article **Assembly Registration Tool (Regasm.exe)**. L'option `/regfile` permet d'indiquer à `regasm.exe` qu'il faut produire le fichier de mise à jour de la base des registres, d'extension `.reg`. L'utilisation de l'option `/regfile` entraîne que la base des registres n'est pas modifiée. L'exécution d'un fichier `.reg` met à jour la base des registres. Dans un fichier `.reg`, les informations de mise à jour sont encodées dans un format texte. Il est ainsi aisé de les consulter et de les modifier. Par exemple :

```
>regasm.exe dotNET2COM.dll /regfile:dotNET2COM.reg
```

Voici le fichier produit :

*Exemple :* *dotNET2COM.reg*

```
[HKEY_CLASSES_ROOT\Test.CCalc]
@="Test.CCalc"

[HKEY_CLASSES_ROOT\Test.CCalc\CLSID]
@="{A51C81A3-4892-39EC-981A-AF77FB4CFD36}"

[HKEY_CLASSES_ROOT\CLSID\{A51C81A3-4892-39EC-981A-AF77FB4CFD36}]
@="Test.CCalc"

[HKEY_CLASSES_ROOT\CLSID\{A51C81A3-4892-39EC-981A-AF77FB4CFD36}
    \InprocServer32]
@="C:\WINDOWS\System32\mscoree.dll"
"ThreadingModel"="Both"
"Class"="Test.CCalc"
"Assembly"="dotNET2COM, Version=0.0.0.0, Culture=neutral,
    PublicKeyToken=null" "RuntimeVersion"="v1.0.3705"

[HKEY_CLASSES_ROOT\CLSID\{A51C81A3-4892-39EC-981A-AF77FB4CFD36}\ProgId]
@="Test.CCalc"

[HKEY_CLASSES_ROOT\CLSID\{A51C81A3-4892-39EC-981A-AF77FB4CFD36}
    \Implemented Categories\{62C8FE65-4EBB-45E7-B440-6E39B2CDBF29}]
```

La clé `InprocServer32`, qui est sensée contenir le chemin et le nom du fichier qui contient l'implémentation de la classe COM, est égale à `mscoree.dll`. Cette DLL, appelée DLL cale, permet le chargement du CLR dans un processus. Les quatre composantes du nom fort de l'assemblage qui contient l'implémentation de la classe .NET sont spécifiées dans la sous-clé `Assembly`. Lorsqu'une application aura besoin de la classe COM qui a pour ProgID `Test.Calc`, elle chargera d'abord le CLR (si ce n'est déjà fait pour ce processus) puis elle utilisera le mécanisme de localisation d'assemblage, pour localiser l'assemblage `dotNET2COM`.

La sous clé `ThreadingModel` vaut `both`, c'est-à-dire que les instances de la classe COM CCW supportent indifféremment les modes STA et MTA. Ceci est la conséquence du fait que les modèles STA et MTA ne sont pas pris en compte dans .NET. Les objets .NET n'ont pas d'affinité avec les threads.

Par défaut le ProgID de la classe COM produite est *{espace de nom}.{nom de la classe}*. Vous pouvez toutefois spécifier un autre ProgID en utilisant l'attribut .NET `System.RunTime.InteropServices.ProgId` dans votre code .NET. Par exemple :

```
...
   [ProgID("dotNET.CCalc")]
   public class CCalc : ICalc{
      ...
```

L'outil `regasm.exe` présente l'option `/tlb` qui permet de fabriquer une bibliothèque de type, exactement comme avec l'outil `tlbexp.exe`.

```
>regasm.exe dotNET2COM.dll /tlb:dotNET2COM.tlb
```

Les outils regasm.exe et tlbexp.exe peuvent être utilisés indépendamment sur le même assemblage. On pourrait craindre que les identifiants uniques des classes et des interfaces ne soient pas les mêmes dans les fichiers produits par regasm.exe et dans les fichiers produits par tlbexp.exe. Heureusement il n'en est rien, ce qui nous amène à penser que les valeurs de ces identifiants uniques sont calculées d'une manière déterministe à partir de l'assemblage. Empiriquement, les identifiants uniques ne changent pas même si les membres des classes ou des interfaces concernées varient. En revanche, les identifiants uniques changent pour les classes ou les interfaces dont le nom change.

## *Utiliser un assemblage .NET comme un composant COM*

Nous présentons ici un exemple de code C++ non géré qui utilise la classe .NET CCalc comme une classe COM. Grâce à l'import de la bibliothèque de types dotNET2COM.tlb, produite à partir de l'assemblage dotNET2COM.dll avec tlbexp.exe ou regasm.exe, le code C++ peut effectuer un lien précoce avec la classe COM CCalc. Voici le code :

*Exemple 8-15 :*

```
#include "stdafx.h"
#import "dotNET2COM.tlb"
int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
   int nRetCode = 0 ;
   // Initialise le thread courant pour qu'il puisse utiliser COM.
   HRESULT hr = ::CoInitialize(NULL);
   dotNET2COM::ICalcPtr pI;
   // Crée l'objet COM qui contient notre objet .NET.
   hr = pI.CreateInstance( __uuidof(dotNET2COM::CCalc));
   // Appel d'une méthode (grâce au lien précoce produit par l'import
   // de la bibliothèque de type).
   int result = pI->CalcSomme(2,3);
   // Libère l'objet COM.
   pI = NULL;
   // Désinitialise le thread courant pour qu'il ne puisse
   // plus utiliser COM.
   ::CoUninitialize();
   return nRetCode ;
}
```

Comme l'assemblage dotNET2COM.dll que nous avons produit ne peut pas être placé dans le GAC (car il n'a pas de nom fort puisqu'il n'est pas signé), il faut obligatoirement qu'il soit dans le répertoire de l'application. N'oubliez pas d'enregistrer la classe CCalc avec regasm.exe (comme décrit dans la section précédente) pour exécuter cet exemple.

## *Exception .NET et CCW*

Supposons qu'une l'exception soit lancée à partir du code géré d'un objet .NET utilisé par du code natif comme un objet COM. Dans ce cas, le CLR a la responsabilité de gérer la transition entre le code géré et le code non géré. Pour accomplir cette tâche, il convertit au mieux les informations de l'exception en un code d'erreur dans un HRESULT. Le CLR utilise des règles

de conversion très précises pour convertir les différentes classes d'exception et les différents `HRESULT`. Par exemple `HRESULT COR_E_DIVIDEBYZERO` est converti en une exception de type `DivideByZeroException`. Tout ceci fait l'objet de l'article « **HRESULTs and Exceptions** » des **MSDN**.

## *Gérer le cycle de vie*

Nous précisons ici une faiblesse du mécanisme d'utilisation d'une classe .NET comme une classe COM. Lorsqu'un tel objet n'est plus utilisé, c'est-à-dire que lorsque le compteur interne de références vers un objet COM CCW arrive à 0, l'objet COM CCW est détruit mais pas l'objet .NET sous-jacent. Cet objet .NET sera détruit par le ramasse-miettes ultérieurement, à un moment indéterminé.

Contrairement au mécanisme d'utilisation d'un objet COM à partir de .NET, il n'existe pas de mécanisme spécial pour forcer la destruction d'un objet .NET encapsulé dans un objet COM CCW. Ce comportement peut ne pas être souhaitable, car l'objet .NET peut posséder de précieuses ressources qu'il faudrait libérer dès qu'il n'est plus utilisé.

Un problème encore plus gênant peut survenir si l'objet .NET contient des références vers des objets COM, et que le développeur du code non géré suppose qu'il n'y a plus d'objet COM dans le processus. Après la destruction de l'objet COM CCW le développeur du code non géré peut appeler la fonction `CoUninitialize()` pour indiquer qu'il n'a plus besoin de la bibliothèque COM dans le thread courant. Cependant, si ce thread est amené ultérieurement à détruire les objets COM possédés par l'objet .NET, il aura un comportement indéterminé qui provoquera sûrement un crash.

Pour éviter les problèmes de gaspillage de ressources et ce scénario catastrophe, il est conseillé de prévoir une méthode dans la classe .NET, spécialement prévue pour libérer les ressources détenues par l'objet .NET. Cette méthode devra être explicitement appelée par le code non géré.

# *Introduction à COM+*

## *Qu'est ce que COM+ ?*

COM+ est le terme désignant les *services d'entreprise* dans le contexte des applications destinées à être exécutées sous les systèmes d'exploitation *Microsoft*. COM+ est donc la technologie *Microsoft* pour construire des *serveurs d'applications*. Un service d'entreprise de COM+ est une fonctionnalité évoluée qui peut être ajoutée à une classe COM. Ces fonctionnalités sont axées autour du développement d'applications distribuées transactionnelles. On verra la liste de ces fonctionnalités dans la prochaine section mais on peut déjà citer le pooling d'objets ou la passerelle avec des milieux transactionnels non *Microsoft*.

COM+ 1.0 est apparu avec *Windows 2000*. COM+ n'était pas une nouvelle version de COM mais une nouvelle version de la technologie *MTS* (*Microsoft Transaction Server*) qui permettait, entre autres, de réaliser des transactions distribuées. Par rapport à la technologie MTS, on obtenait de bien meilleures performances avec COM+ 1.0. COM+ 1.5 est apparu avec *Windows XP* et ajoute quelques services d'entreprise à COM+ 1.0.

## *COM+ via .NET : les services d'entreprises*

Il était nécessaire que le *framework* .NET ait une technologie de serveur d'applications. Pendant les dernières années, *Microsoft* a mobilisé beaucoup de ressources pour le développement du *framework* .NET. Parallèlement, la technologie COM+ offrait satisfaction, et beaucoup de ressources avaient déjà étés utilisées pour mettre au point cette technologie. *Microsoft* a donc préféré capitaliser sur COM+ plutôt que de re-développer une nouvelle technologie de serveurs d'application.

Chaque service d'entreprise COM+ est exploitable via une classe .NET. Le *framework* .NET présente un ensemble d'attributs, de classes et d'outils permettant à vos classes .NET d'utiliser les services d'entreprise COM+. Ces attributs et ses classes sont sous l'espace de noms `System.EnterpriseServices`. Ils sont implémentés dans l'assemblage `System.EnterpriseServices.dll` qui se trouve dans le GAC. L'idée principale est que le développeur n'a pas à connaître COM+ dans les détails pour pouvoir utiliser les services d'entreprise COM+ dans ses classes .NET.

# *Présentation des services d'entreprise COM+*

## *La liste complète des services d'entreprise COM+*

La présentation détaillée de tous les services d'entreprise COM+ dépasse le cadre de cet ouvrage. Voici la liste des services d'entreprise de COM+ que vous pouvez assigner à vos classes .NET :

- **Pooling d'objets** ;
  L'idée du pooling d'objet est de recycler les objets pour qu'ils puissent chacun servir plusieurs clients consécutivement. Les coûts de la construction et de la destruction d'un objet sont alors divisés par le nombre de clients servis par l'objet. Le pool représente le conteneur des objets qui ne sont pas en train de servir un client.
- **Activation juste à temps (JITA Just In Time Activation)** ;
  L'idée de ce mécanisme est de désactiver un objet après les appels de certaines méthodes, afin de le rendre utilisable par d'autres clients. Rappelons que désactiver un objet signifie que COM+ appelle la méthode `Deactivate()` sur l'objet, puis le met dans le pool d'instances du composant servi concerné. Si un tel pool d'objets n'existe pas, la méthode `Dispose()` est appelée sur l'objet et il sera détruit ultérieurement par le ramasse-miettes. Pour indiquer que les instances d'un composant servi supportent le mécanisme JITA, il suffit d'utiliser l'attribut `System.EnterpriseServices.JustInTimeActivation` dans la déclaration du composant servi concerné.
- **Transactions distribuées** ;
  COM+ permet d'exploiter le serveur transactionnel de *Windows* nommé MS DTC (*Distributed Transaction Coordinator*) afin de réaliser des transactions distribuées.
- ***BYOT* (Bring Your Own Transaction)**
  Ce mécanisme permet d'associer une transaction existante à un composant qui *a priori* ne fait pas partie de cette transaction. La transaction peut être gérée par MS DTC (le gestionnaire de *Windows* de transactions distribuées), mais aussi par un autre milieu transactionnel comme *TIP* (*Transaction Internet Protocol*).

- **COM Transaction Integrator (*COMTI*)**
  Ce mécanisme vous permet d'intégrer les mainframes développés avec les technologies *CICS* (*Customer Information Control System*) et *IMS* (*Information Management System*) de *IBM*, dans des objets COM classiques. Sous le terme COMTI sont regroupés les outils et les classes permettant cette intégration.
- **Compensating Resource Manager (CRMs)**
  La théorie des transactions s'applique, entre autres, aux bases de données relationnelles. Cependant, il ne faut pas oublier qu'elle constitue un cadre théorique général à la protection de l'intégrité de ressources quelconques. Un composant servi CRM est un composant servi propriétaire qui gère des ressources (par exemple des fichiers). Un CRM peut participer au sein d'une transaction distribuée gérée par MS DTC (le gestionnaire *Microsoft* de transactions distribuées). Lorsqu'un CRM participe à une transaction, il doit pouvoir stocker temporairement et de manière persistante l'état de ses ressources avant leur modification de manière à pouvoir revenir en arrière si la transaction échoue. Le CRM doit être capable de valider les changements (*commit*) ou de les annuler (*rollback*) même si le processus qui le gère est détruit durant la transaction. Si ce processus est effectivement détruit durant la transaction, les opérations de *commit* ou de *rollback* doivent pouvoir être effectuées lorsque le processus est relancé. Contrairement à ce que l'on pourrait penser, écrire un CRM est une tâche relativement aisée. La plupart de la complexité est encapsulée dans les classes fournies par le *framework* .NET.
- **Interopérabilité XA**
  Ce mécanisme permet d'encapsuler, au sein d'une transaction gérée par le modèle de transaction *Microsoft* (*OLE Transaction*), les accès à une base de données qui supporte le modèle transactionnel *XA* (*X/Open Distributed Transaction Processing* (*DTP*).
- **Événements couplés (Loosely Coupled Events)**
  Ce mécanisme permet à un client d'une COM+ application d'être averti par le serveur lorsqu'un événement se produit. Cela évite au client d'aller demander régulièrement au serveur si l'événement s'est produit.
- **Passage d'une chaîne de caractères à la construction d'un objet**
  Ce mécanisme permet de pallier partiellement à l'absence d'arguments des constructeurs des classes COM. Cependant la chaîne de caractères est la même pour toutes les instances d'un composant servi. Vous avez la possibilité de modifier cette chaîne par l'intermédiaire de l'outil « Services de composants ». La principale utilité de cette chaîne est de fournir une chaîne de connexion vers une base de données.
- **Composants privés**
  Tous les composants servis d'une COM+ application ne sont pas nécessairement accédés par les clients de la COM+ application. Il existe souvent des composants servis qui ne doivent être utilisés que par d'autres composants servis, existant dans le même processus et appartenant à la même COM+ application. Pour empêcher un client d'utiliser un tel composant servi, il faut le déclarer comme un composant servi privé.
- **Appels asynchrones déconnectés (Queued Components)**
  Ce mécanisme d'*appel asynchrone* va plus loin que le mécanisme d'appel asynchrone de .NET. Contrairement à ce mécanisme, les requêtes des clients ne sont pas traitées directement. Elles sont stockées dans une file d'attente sous forme de messages. Durant cette opération, aucun objet serveur n'est créé. Le client et le serveur peuvent même être physiquement déconnectés. Tout ceci est transparent pour le client qui dialogue avec un objet proxy.

Tout ceci est transparent pour le serveur qui ultérieurement récupérera ces requêtes et les traitera lorsqu'il sera connecté avec la machine cliente. Notez que le mécanisme interne de gestion de file d'attente est *Microsoft Message Queue* (*MSMQ*). MSMQ doit être installée à la fois sur la machine client et la machine serveur.

- **Sécurité Role-Based**
  COM+ présente son propre mécanisme de sécurité, basé sur le rôle que joue l'utilisateur appelant. Ce mécanisme est différent du mécanisme de sécurité .NET basé sur les rôles, et du mécanisme *Windows* basé sur les rôles.
- **Services SOAP**
  Ce mécanisme permet de publier un composant servi comme un service web. On peut toujours continuer à y accéder comme un composant COM+.
- **Synchronisation / activité COM+**
  Une *activité COM+* est un arbre logique d'appels entre instances de composants servis. Ce mécanisme permet de synchroniser l'accès à des instances de composants servis entre plusieurs activités COM+. Le mécanisme d'interception d'appels de COM+ fait en sorte qu'il ne puisse y avoir plus d'une activité COM+ qui utilise une instance d'un composant servi à la fois.

## *Avez-vous besoin des services d'entreprise COM+ ?*

Parce que COM+ n'est pas intégré à .NET, il vaut mieux éviter de l'utiliser quand cela est possible. En effet, certaines fonctionnalités sont redondantes entre les briques logicielles du *framework* .NET et les services d'entreprise COM+. Voici quelques-unes de ces fonctionnalités redondantes.

.NET 2.0 présente un *framework* de développement transactionnel qui est toujours préférable à la gestion des transactions avec COM+. Ce *framework* fait l'objet du chapitre sur les transactions.

Bien souvent, le pooling d'objets COM+ est utilisé pour éviter d'avoir à construire et à détruire une connexion avec une base de données, pour chaque client. Soyez conscient que la fonctionnalité de pool de connexions est en général prévue par les fournisseurs de données ADO.NET (voir page 718).

La fonctionnalité COM+ d'activation JITA d'un objet (« Just In Time Activation » ou « activation juste à temps ») permet de repousser la création (ou l'activation) d'un objet jusqu'au moment de sa première utilisation réelle par un client. Or, le mode WKO « simple appel » de .NET Remoting fonctionne sur ce modèle d'activation (voir page 795).

La notion d'appels asynchrones de COM+ est un peu plus générale que celle de .NET, présentée page 171. Elle est plus générale dans le sens où elle permet au client d'effectuer l'appel même si le serveur n'est pas joignable au moment de l'appel. Dans ce cas, l'appel est stocké dans une file d'attente côté client, et sera effectué automatiquement lorsque le serveur redeviendra joignable. Si vous n'avez pas besoin de cette fonctionnalité, mieux vaut utiliser les appels asynchrones .NET.

La notion de rôle dans le domaine de la sécurité est à la fois présente en COM+, *Windows* et .NET. Vous pouvez en savoir plus à ce sujet page 216.

La possibilité de n'autoriser au plus qu'un client à utiliser un composant à un instant donné, est possible aussi bien avec COM+ qu'avec .NET (voir page 160).

# *Utiliser les services COM+ dans des classes .NET*

## *Notion de composant servi (serviced component)*

Un *composant servi* (*serviced component* en anglais) est une classe .NET utilisant un ou plusieurs services d'entreprise COM+. Pour utiliser au moins un service d'entreprise COM+, une classe .NET doit impérativement dériver de la classe `System.EnterpriseServices.ServicedComponent`. Un composant servi doit impérativement avoir un constructeur sans argument, aussi appelé constructeur par défaut.

La classe `ServicedComponent` implémente l'interface `IDisposable` et sa méthode `Dispose()`. La classe `ServicedComponent` présente aussi la méthode virtuelle protégée `Dispose(bool)`. Si le booléen est positionné à `true`, cela signifie qu'il faut libérer les ressources gérées et non gérées. Si le booléen est positionné à `false`, cela signifie qu'il ne faut libérer que les ressources non gérées.

Voici donc à quoi ressemble le squelette d'un composant servi :

*Exemple 8-16 :*

```
using System ;
using System.EnterpriseServices ;

public class SystemeBancaire : ServicedComponent {
   public SystemeBancaire() {

   }
   new public void Dispose() {
      Dispose(true) ;
      GC.SuppressFinalize(this) ;
   }
   protected override void Dispose(bool bDisposing){
      // Libère les ressources non gérées.
      if( bDisposing ){
         // Libère les ressources gérées.
      }
      base.Dispose(bDisposing) ;
   }
}
```

## *Déclarer les services d'entreprise utilisés dans un composant servi*

Pour signaler au compilateur qu'une classe dérivant de `ServicedComponent` utilise tel ou tel service d'entreprise, il faut que la classe soit déclarée avec les attributs .NET correspondant aux services d'entreprise souhaités. Ces attributs sont tous dans l'espace de noms `System.EnterpriseServices`. Par exemple l'attribut `Transaction` permet de paramétrer le mode transactionnel utilisé dans une classe. L'attribut `ObjectPooling` permet d'indiquer que les instances de la classe font partie d'un pool. Pour utiliser conjointement ces deux services d'entreprise et les paramétrer dans notre classe `SystemeBancaire`, il faudrait la déclarer comme ceci :

```
...
[Transaction(TransactionOption.Required)]
[ObjectPooling(MinPoolSize=5,MaxPoolSize=30)]
public class SystemeBancaire : ServicedComponent {
   ...
```

## *Contexte COM+ et utilisation des services d'entreprise dans un composant servi*

Un *contexte COM+* est un conteneur logique dans un processus, qui héberge les objets qui utilisent les mêmes services d'entreprise COM+.

> Attention, bien que conceptuellement proche de la notion de contexte .NET présentée page 842, la notion de contexte COM+ est différente. Certains documents parlent de contextes gérés (.NET) et de contextes non gérés (COM+).

La plomberie interne pour utiliser un service d'entreprise peut être assez lourde. La notion de contexte permet de partager cette plomberie entre plusieurs objets afin d'améliorer les performances. La gestion des contextes est transparente pour le développeur. Lors de la création d'une instance d'un composant servi, COM+ s'occupe de vérifier s'il existe un contexte dans le processus qui utilise les mêmes services d'entreprise. Si tel est le cas, l'instance du composant servi utilisera ce contexte, sinon un nouveau contexte avec les services d'entreprise adéquats est créé.

Sachez que si le client d'une instance d'un composant servi n'est pas dans le même contexte COM+, COM+ crée un objet *proxy* pour le client. L'intérêt est que chaque appel au composant servi est d'abord intercepté en interne par COM+ avant d'être exécuté. Chaque retour d'appel est aussi intercepté par COM+. Ce que fait COM+ durant cette interception dépend complètement des services d'entreprise utilisés. Le mécanisme JITA, décrit un peu plus loin, constitue un exemple intéressant de ce que peut faire COM+ durant ces interceptions. Notez que ce mécanisme d'objets proxy et d'interception est transparent pour l'utilisateur.

Du point de vue du développeur d'un composant servi .NET, le contrôle de l'utilisation des services d'entreprise se fait par l'intermédiaire de la classe `System.EnterpriseServices.ContextUtil`. Cette classe ne contient que des membres statiques. Supposons qu'un composant servi utilise le service d'entreprise de gestion des transactions et que vous souhaitiez avorter la transaction courante dans le code d'une méthode. Il suffit d'appeler la méthode `ContextUtil.SetAbort()`. On ne peut pas dire qu'un composant servi .NET appartient à un contexte COM+, car le composant servi est géré mais pas le contexte. On dit plutôt qu'un composant servi utilise un contexte pour avoir accès à ses services d'entreprise. La Figure 8-6 résume tout ceci :

## *Notion de COM+ application*

Une *COM+ application* est une collection de composants servis. Les composants servis d'une COM+ application peuvent être dans différents assemblages. En revanche, tous les composants

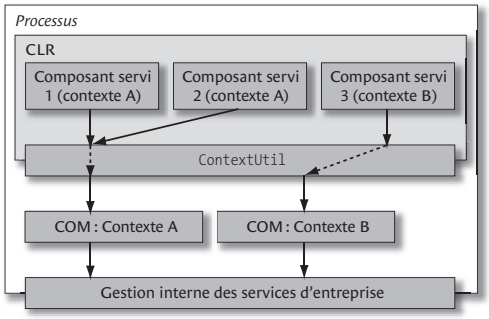


*Figure 8-6 : Contextes COM+ et composants servis .NET*

servis d'un même assemblage appartiennent à la même COM+ application. Les COM+ applications représentent ce que l'on appelle communément les *serveurs d'entreprises*. Les clients utilisent les instances des composants servis d'une COM+ application, et les composants servis utilisent les services d'entreprise COM+ pour effectuer leurs tâches.

Plusieurs attributs d'assemblage ont été conçus pour que vous puissiez configurer les paramètres de la COM+ application qui contiendra l'assemblage que vous développez. Voici les principaux :

- `ApplicationName` : précise le nom de la COM+ application. Par exemple :

```
[assembly : ApplicationName("Serveur bancaire.")]
```

- `ApplicationID` : précise l'identificateur unique de la COM+ application.

```
[assembly : ApplicationID("3O1E31A3-EO11-432b-9D7E-5643253EEE89")]
```

- `Description` : donne une description de la COM+ application.

```
[assembly : Description("Permet d'accéder aux informations bancaires.")]
```

- `ApplicationAccessControl` : permet de configurer des paramètres de sécurité de la COM+ application.
- `ApplicationQueuing` : Permet de configurer l'utilisation de files de messages dans la COM+ application.
- `ApplicationActivation` : Il existe deux modes d'activation d'une COM+ application, le mode d'activation librairie et le mode d'activation serveur. La description de ces modes fait l'objet de la prochaine section.

Si plusieurs assemblages configurent la même COM+ application, un paramètre donné de la COM+ application prendra la valeur spécifiée dans le dernier assemblage installé. Si un paramètre de la COM+ application n'est positionné par aucun assemblage, il sera positionné à une valeur par défaut, lors de la création de la COM+ application.

## Le catalogue COM+

Le *catalogue COM+* est une base de données présente dans tous les systèmes d'exploitation *Microsoft* depuis Windows 2000. Le catalogue COM+ contient les informations de configuration des COM+ applications qui résident sur la machine. Le catalogue COM+ est physiquement stocké sur deux supports :

- Une partie des informations du catalogue COM+ est stockée dans la base des registres, directement avec les informations des classes COM.
- L'autre partie des informations du catalogue COM+ est stockée dans une série de fichiers bibliothèques de composants (extension `.clb`). Ces fichiers sont stockés dans le répertoire `<<%systemroot%\Registration>>`.

Le format des fichiers `.clb` n'est pas documenté. Il est fortement déconseillé d'essayer de modifier ou de visualiser les données du catalogue COM+ par l'intermédiaire d'un éditeur tel que `regedt32.exe` ou en modifiant les fichiers `.clb`. Nous présenterons un peu plus loin, un outil spécialement conçu pour visualiser et modifier les données du catalogue COM+, de façon à pouvoir créer et paramétrer des applications COM+ sur une machine. La Figure 8-7 expose les différentes relations existant entre le catalogue COM+, les COM+ applications et les composants servis, installés sur une machine.

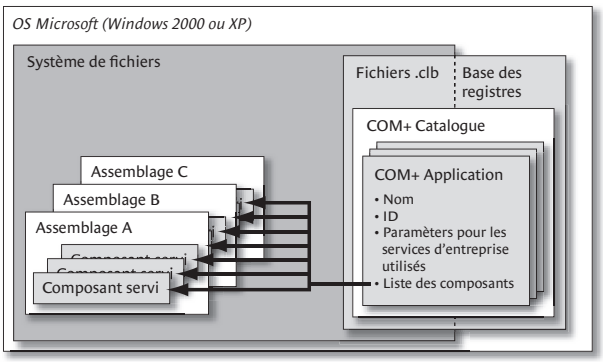


*Figure 8-7 : Catalogue COM+ et COM+ applications*

## Les modes d'activation d'une COM+ application

Chaque COM+ application a un *mode d'activation* qui est soit « librairie » soit « serveur ».

### Mode d'activation librairie

Le mode d'activation « librairie » indique que les assemblages contenant les composants servis d'une COM+ application doivent être chargés dans le processus du client de la COM+ application. La Figure 8-8 expose l'architecture générale mise en œuvre dans le mode d'activation « librairie ».

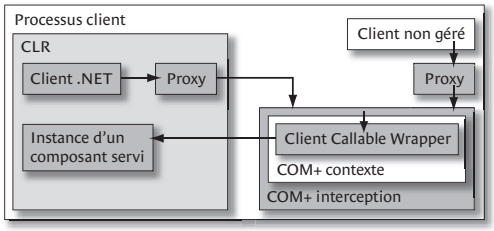


*Figure 8-8 : Mode d'activation librairie d'une application COM+*

### *Mode d'activation serveur*

Le mode d'activation « serveur » indique que les assemblages contenant les composants servis d'une COM+ application doivent être chargés dans un processus prévu pour héberger les COM+ applications. Ce processus est lancé automatiquement par le système lors de la première utilisation d'une COM+ application. L'exécutable `dllhost.exe` est utilisé pour lancer ce processus. On dit d'un tel processus qu'il est *subrogé* (*surrogate* en anglais). Pour qu'une COM+ application puisse être utilisée par des clients distants (i.e qui ne sont pas sur la même machine) il faut naturellement qu'elle ait le mode d'activation « serveur ». La Figure 8-9 expose l'architecture générale mise en œuvre dans le mode d'application « serveur ».

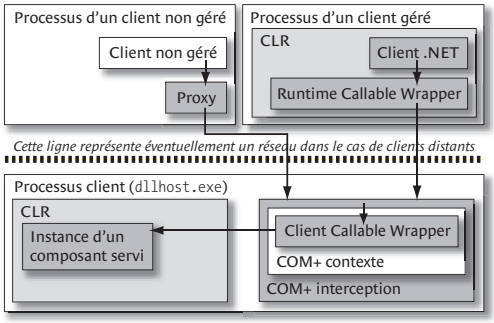


*Figure 8-9 : Mode d'activation serveur d'une application COM+*

## *Installation des composants servis dans une COM+ application*

L'installation des composants servis contenus dans un assemblage dans une COM+ application a besoin que l'assemblage ait un nom fort. C'est-à-dire que l'assemblage doit être signé numériquement avec l'outil `sn.exe`. Au moment de l'exécution, le CLR trouvera cet assemblage soit dans le GAC, soit dans le répertoire courant de l'application.

### *Les étapes de l'installation*

L'installation des composants servis contenus dans un assemblage, dans une COM+ application, contient les quatre étapes suivantes :

- Enregistrer les composants servis dans la base des registres, pour qu'ils puissent être accédés comme des objets COM. La description de cette étape fait l'objet de la section précédente.
- Construire une bibliothèque de types décrivant les composants servis contenus dans l'assemblage. Cette bibliothèque de types contient les informations spécifiées par les attributs COM+ contenus dans l'assemblage.
- Trouver la COM+ application qui doit contenir ces composants servis. Si la COM+ application n'est pas trouvée alors créer une nouvelle COM+ application.
- Configurer la COM+ application à partir des informations de la bibliothèque de types, issues des attributs COM+ de l'assemblage. Les paramètres non positionnés par ces valeurs prennent des valeurs par défaut. De plus, si la COM+ application existait déjà, il se peut que les valeurs de ses paramètres changent avec l'installation de ces nouveaux composants servis.

### *Les différentes façons de réaliser l'installation*

La classe `System.EnterpriseServices.RegistrationHelper` permet d'installer (ou de désinstaller) les composants servis d'un assemblage. Elle permet d'exécuter les quatre étapes vues ci-dessus dans une même transaction. Après avoir utilisé la classe `RegistrationHelper` vous pouvez donc être certain que soit toutes les étapes ont été effectuées avec succès, soit rien n'a été modifié car une des étapes a échoué.

Concrètement, il existe trois façons d'installer les composants servis d'un assemblage. Chacune de ces façons utilise la classe `RegistrationHelper` d'une manière masquée ou non. Quelle que soit la façon utilisée, seul un utilisateur ayant les droits d'administrateur peut réaliser cette opération.

- La façon « manuelle » :
  Il vous suffit d'utiliser l'outil *Services Installation Utility* (`regsvcs.exe`) en ligne de commande, avec pour argument l'assemblage qui contient les composants servis. Par exemple :

  ```
  >regsvcs.exe SystemeBancaire.dll
  ```

  Les options de l'outil `regsvcs.exe` sont décrites dans les **MSDN** à l'article **.NET Services Installation Tool (Regsvcs.exe)**. Si le nom de la COM+ application n'est pas spécifié dans l'assemblage ou à partir de l'option `/appname` de `regsvcs.exe`, le nom de la COM+ application sera égal au nom de l'assemblage (sans extension de fichier).
- La façon « programmée » :
  Il vous suffit d'utiliser la classe `RegistrationHelper` au sein d'une méthode d'une classe d'un assemblage. La classe `RegistrationHelper` présente la méthode `InstallAssembly()` prévue à cet effet. Par exemple :

  *Exemple 8-17 :*

  ```
  using System.EnterpriseServices ;
  public class Program {
     static void Main() {
        RegistrationHelper rh = new RegistrationHelper() ;
        string sCOMPlusAppName = "Serveur bancaire." ;
        string sTypeLibName = "SystemeBancaire.tlb" ;
        rh.InstallAssembly(
  ```

```
            "SystemeBancaire.dll", // Nom de l'assemblage.
            ref sCOMPlusAppName,
            ref sTypeLibName,
            InstallationFlags.CreateTargetApplication) ;
    }
}
```

L'argument `CreateTargetApplication` indique qu'une nouvelle COM+ application doit être créée. Si ce n'est pas possible l'exception `RegistrationException` est lancée.

- La façon « automatique » :
  Quand un client a besoin qu'un composant servi soit instancié, le CLR vérifie si le composant servi est effectivement présent dans une COM+ application. Si tel n'est pas le cas, le CLR installe automatiquement l'assemblage dans une nouvelle COM+ application. Bien que séduisante, cette façon de faire doit être évitée car vous pouvez difficilement être certain que le premier client a les droits d'administrateur qui sont nécessaires pour réaliser cette opération.

## *Visualisation et manipulation du catalogue COM+*

Vous pouvez visualiser et manipuler le catalogue COM+, et donc les COM+ applications présentes sur une machine, en utilisant le *composant logiciel enfichable* (*snap-in* en anglais) « *Services de composants* » (*component services* en anglais). La Figure 8-10 présente une copie d'écran de cet outil :

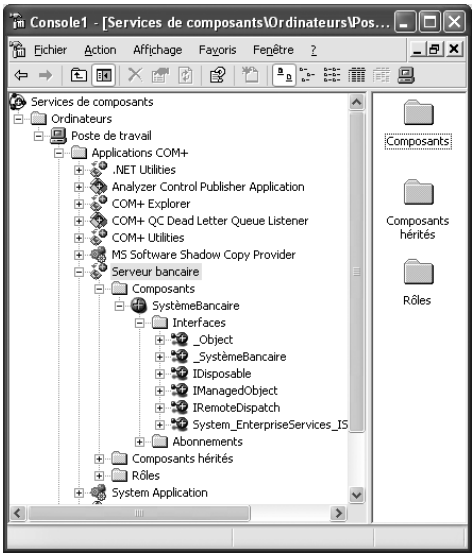


*Figure 8-10 : L'outil Services de composants*

L'outil « services de composants » permet à un administrateur de modifier les paramètres d'une COM+ application. Cependant, il est conseillé de ne modifier que les paramètres relatifs à l'installation d'une COM+ application, comme les paramètres de sécurité.

Si une COM+ application est en cours d'utilisation, l'outil « services de composants » vous l'indique avec une petite animation. Il vous permet aussi de stopper une COM+ application utilisée. Si vous changez les paramètres d'une COM+ application, il faudra stopper la COM+ application pour que les changements soient pris en compte lors de la prochaine utilisation.

## *Concevoir un client d'un composant servi*

Du point de vue d'un client géré, il n'y a aucune différence entre l'utilisation d'une instance d'une classe normale et l'utilisation d'une instance d'un composant servi. Il est conseillé d'utiliser la syntaxe C# avec le mot-clé `using` pour appeler la méthode `Dispose()` d'une manière implicite. Par exemple :

```
public class Program {
   static void Main(){
      using( SystemeBancaire sys = new SystemeBancaire() ){
         //... utilise sys
      } //appelle sys.Dispose() d'une manière implicite
   }
}
```

Du point de vue d'un client non géré, l'utilisation d'une classe composant servi passe par l'utilisation de la classe COM définie par le *Com Callable Wrapper*.

# Le langage C♯ 2.0 et la comparaison C♯ 2.0/C++

# 9

# *Les concepts fondamentaux du langage*

## *Organisation du code source*

C++ → C♯ Malgré une syntaxe relativement proche de celle du C++, les espaces de noms jouent un rôle plus important dans C♯.

En C++ on les utilisait pour empêcher les collisions entre identificateurs.

En C♯, les espaces de noms remplacent la directive préprocesseur `#include`. En effet, en C♯ il n'y a plus de fichiers d'en-tête (.h). Pour utiliser une ressource contenue dans un espace de noms et déclarée dans un autre fichier source C♯, il suffit de déclarer qu'on utilise l'espace de noms concerné. Ce fichier source peut éventuellement faire partie d'un autre assemblage que celui dans lequel on utilise la ressource.

Cette possibilité d'utilisation de ressources déclarées dans un autre assemblage est aussi exploitée dans l'utilisation de ressources définies dans des bibliothèques. Par exemple toutes les classes de base du *framework* .NET sont accessibles au travers de l'espace de noms `System`.

Une autre grosse différence est que l'on ne peut séparer la déclaration et la définition (le corps) d'une méthode ou d'une classe, comme on pouvait le faire en C++ grâce à l'opérateur de résolution de portée.

Autre différence importante : il n'existe plus de fonctions globales. Il ne peut y avoir que des méthodes, c'est-à-dire des fonctions déclarées à l'intérieur d'un type (classe ou structure).

En C♯, l'utilisation des espaces de noms pour empêcher les collisions entre identificateurs est toujours d'actualité. De plus pour étendre le nom d'un identificateur en le faisant précéder du nom de l'espace de noms, on n'utilise plus l'opérateur de résolution de portée du C++ (l'opérateur '::' qui a une autre application en C♯2) mais un point.

Enfin les possibilités de définir des alias de noms d'espace de noms, et d'imbriquer des espaces de noms, sont présentes en C#.

## *Les espaces de noms*

Une ressource dans le code source peut être déclarée et définie à l'intérieur d'un espace de noms. Si une ressource n'est déclarée dans aucun espace de noms elle fait partie d'un espace de noms global et anonyme. Un *espace de noms* nommé `Foo` par exemple, se déclare comme suit :

```
namespace Foo{
  // Ici, la définition d'éléments.
}
```

On a la possibilité d'imbriquer les espaces de noms :

```
namespace Foo1{
   // Ici, la définition d'éléments.
   namespace Foo2{
      // Ici, la définition d'éléments.
      // La qualification complète de Foo2 est Foo1.Foo2.
   }
   // Ici, la définition d'éléments.
}
```

On a la possibilité de découper un espace de noms. Par exemple, pour étaler des ressources qui doivent être dans le même espace de noms sur plusieurs fichiers sources. Pour cela il suffit de déclarer un ou plusieurs espaces de noms avec le même nom pour chaque fichier source concerné. Notez qu'avec les types partiels de C#2, une ressource a maintenant la possibilité d'être déclarée sur plusieurs fichiers sources.

La technique de découpage d'un espace de noms est d'autant plus puissante que le découpage peut se faire sur les fichiers sources de différents modules (du même assemblage ou non).

Les espaces de noms constituent un moyen de partitionner le code source dans des blocs distincts et organisés, selon une hiérarchie que les architectes du projet ont établi.

Enfin, mentionnons que les documentations anglo saxonnes utilisent souvent les expressions `Foo` et `Bar` pour nommer les entités quelconques de leurs exemples (un peu comme nous utilisons parfois `toto`). Ces expressions sont issues de l'acronyme FUBAR (*Fucked Up Beyond All Repairs*) qui peut se traduire par *c'est irréparable*, *c'est foutu*. Cet acronyme a vraisemblablement été inventé lors de la seconde guerre mondiale pour désigner une situation désespérée.

## *Utilisation des ressources contenue dans un espace de noms*

Les types de ressources déclarés à l'intérieur d'un espace de noms sont :

| Type de ressource | Mot-clé C# |
|---|---|
| espace de noms | `namespace` |
| classe | `class` |
| interface | `interface` |
| structure | `struct` |
| énumération | `enum` |
| délégué | `delegate` |

Dans une moindre mesure on peut considérer que les commentaires constituent une sorte de ressources dont la caractéristique principale est de ne pas être pris en compte par le compilateur.

Pour utiliser une ressource définie dans l'espace de noms de nom B, il faut que celui-ci soit déclaré en tête du fichier source ou dans l'espace de noms où l'on souhaite utiliser la ressource, avec le mot-clé `using`. Par exemple :

*Exemple 9-1 :*

```
using B;
namespace A{
   class Program{
      static void Main() {}
      ClasseFoo f ; // la classe ClasseFoo peut être utilisée.
   }
}
namespace B{
   using A;
   class ClasseFoo{Program p;} // la classe Program peut être utilisée.
}
```

Toutes les ressources déclarées dans un même espace de noms sont accessibles à partir du code contenu dans cet espace de noms. Y compris celles qui sont déclarées dans le même espace de noms dans d'autres fichiers sources, d'autres modules ou assemblages référencés à la compilation.

Comprenez bien que le mot-clé `using` n'est qu'une commodité à l'usage du compilateur C#2 pour lui permettre de faire le lien entre les ressources et les noms des ressources qualifiés sans leurs espaces de noms. L'équivalent en Java et en VB.NET est le mot clé `Imports`/`imports`. Dans aucun de ces trois langages ce mot-clé n'importe quoi que ce soit au sens où quelque chose est déplacé.

On peut utiliser le mot-clé `using` pour définir un *alias d'espace de noms*. En général, cela permet d'éviter d'avoir à nommer les espaces de noms imbriqués. Par exemple :

```
using SysWinForm = System.Windows.Forms ;
```

Il faut être conscient que dans ce cas on est obligé d'utiliser l'alias devant les identificateurs de cet espace. Ce n'est pas le cas lorsqu'il n'y a pas d'alias et qu'il n'y a pas de collisions d'identificateurs.

Enfin, il n'existe pas en C# la syntaxe Java suivante :

```
using Foo.* ;
```

## *Structure d'un projet C#*

Le code source d'un projet écrit en C# est répartit sur un ou plusieurs fichiers texte dont l'extension est `.cs`. Précisons qu'un projet C# est l'ensemble des fichiers sources C# concernés dans la fabrication d'un assemblage par le compilateur C#. L'organisation d'un projet C# peut être résumée par la Figure 10-1 :

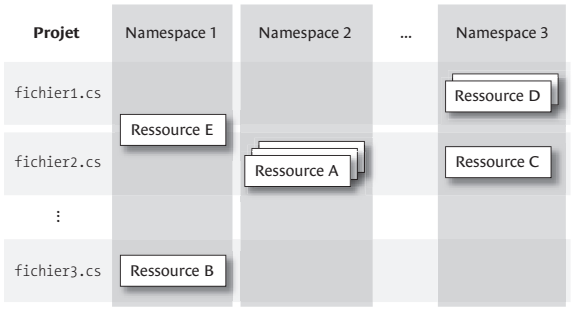


*Figure 9-1 : Les éléments structurels d'un projet C#*

En tenant compte des remarques suivantes :

- Le même espace de noms peut se trouver sur plusieurs fichiers.
- Les ressources hors de tout espace de noms sont dans l'espace de noms anonyme.
- Comme exposé par la ressource E, depuis la version 2.0 de C# une classe une structure ou une interface a la possibilité d'être déclarée sur plusieurs fichiers sources d'un même projet, mais dans le même espace de noms.

## *Organisation assemblages / espaces de noms*

Une bonne pratique consiste à nommer un assemblage avec le nom de l'espace de noms racine qu'il contient. *Microsoft* applique cette règle pour nommer ses assemblages standard. Ainsi, l'organisation des classes de bases standard apparaît immédiatement à la lecture des noms des assemblages : `System.dll`, `System.Drawing.dll`, `System.Drawing.Design.dll`, `System.Runtime.Remoting.dll`, `System.Security.dll` etc.

# *Les étapes de la compilation*

C++ → C# Attention, l'étape de la création de liens (*link*) du C++ n'existe plus en C#. En effet le compilateur C# ne se sert pas de fichiers intermédiaires d'extension `.obj`.

En revanche les étapes du préprocessing et la compilation elle-même sont toujours là. En C# beaucoup de directives préprocesseur du C++ disparaissent, et quelques nouvelles voient le jour.

C# La compilation consiste à fabriquer un assemblage à partir de fichiers sources écrits en C#. La notion d'assemblage fait l'objet du chapitre 2 « Assemblages, modules, langage IL ». La

compilation est prise en charge par le compilateur `csc.exe` décrit un peu plus loin. Ce compilateur peut être appelé soit directement en ligne de commande, soit par l'environnement de développement *Visual Studio*. Les options de compilation sont définies soit dans la ligne de commande, soit par l'intermédiaire de l'environnement de développement.

Il y a principalement deux étapes à la compilation :

- *Préprocessing* des fichiers sources : cette étape génère de nouveaux fichiers sources, modifiés selon des directives préprocessing qui peuvent être définies soit dans les fichiers sources eux même, soit directement en ligne de commande du compilateur. Ces directives sont décrites un peu plus loin.
- *Compilation* des fichiers sources résultant du préprocessing : le produit de cette étape est soit :
  - un assemblage, contenu dans un fichier dont l'extension est `.exe` ou `.dll`,
  - un module contenu dans un fichier d'extension `.netmodule`. Rappelons que l'environnement *Visual Studio* ne sait pas gérer les modules.

# *Le préprocesseur*

C++ → C# En C# la directive `#define` ne peut plus définir une constante à remplacer. On ne l'utilise plus que pour changer simplement le code source, avec l'aide des directives `#if #elif #else et #endif`.

Les constantes prédéfinies ( `__LINE__ ,__FILE__, __DATE__`...) n'existent plus.

Les macros avec paramètres n'existent plus.

L'environnement de développement *Visual Studio* est capable de mettre en gris les lignes qui ne seront pas prises en compte lors de la compilation.

La très célèbre directive `#include` n'existe plus, puisque l'organisation des fichiers sources est très différente en C#.

Les opérateurs # et ## de manipulation de chaînes de caractères par le préprocesseur n'existent plus.

La directive `#warning` a été ajoutée en C# et fonctionne sur le même principe que la directive `#error`.

Les directives `#line #region` et `#endregion` apparaissent et sont spécifiques à C#.

C# Toute compilation d'un fichier source est précédée d'une phase de mise en forme du fichier. Celle-ci est effectuée par un préprocesseur. Il n'effectue que des traitements simples de manipulation textuelle. En aucun cas le préprocesseur n'est chargé de la compilation. Toutes les *directives préprocesseur* sont précédées par le caractère #.

Le préprocesseur reconnaît les directives suivantes :

```
#define    #undef    #if      #elif    #else   #endif
#error     #warning  #line    #region  #endregion
#pragma warning disable   #pragma warning restore
```

## Constantes symboliques et compilation conditionnelle

Vous avez la possibilité de définir une entité avec la directive `#define`. Selon la présence ou non de l'entité, le code source peut être modifié avec l'utilisation des directives `#if #elif #else` et `#endif` :

*Exemple 9-2 :*

```
#define MACRO1
public class Program {
   public static void Main() {
#if (MACRO1)
      System.Console.WriteLine("MACRO1 définie.") ;
#elif (MACRO2)
      System.Console.WriteLine("MACRO2 définie et MACRO1 non définie") ;
#else
      System.Console.WriteLine("MACRO2 et MACRO1 non définies") ;
#endif
   }
}
```

L'environnement de développement *Visual Studio* met en gris les lignes qui ne seront pas prises en compte lors de la compilation. En outre, les constantes symboliques doivent être définies avant toute utilisation de la directive C# `using`.

On a aussi la possibilité de définir une constante symbolique lors de la compilation en ligne. Par exemple :

```
>csc.exe prepross.cs /define:MACRO2
```

La directive `#undef` annule la définition d'une constante symbolique, par exemple :

*Exemple 9-3 :*

```
#define MACRO1
#undef MACRO1
public class Program {
   public static void Main() {
#if (MACRO1)
      System.Console.WriteLine("MACRO1 définie.") ;
#elif (MACRO2)
      System.Console.WriteLine("MACRO2 définie et MACRO1 non définie") ;
#else
      System.Console.WriteLine("MACRO2 et MACRO1 non définies") ;
#endif
   }
}
```

## Constantes symboliques et l'attribut ConditionalAttribute

Il existe une alternative élégante (mais restrictive) à l'utilisation de `#if #elif #else` et `#endif`. Cette alternative est l'attribut `ConditionalAttribute` qui permet de définir une méthode en

fonction de la définition d'une constante symbolique. L'avantage par rapport à l'utilisation des directives préprocesseur `#if #elif #else` et `#endif`, est qu'il n'est pas nécessaire d'aller commenter les appels à la méthode lorsque celle-ci n'est plus définie. L'inconvénient est qu'un certain nombre de contraintes doivent s'appliquer à la méthode. Par exemple la méthode doit retourner le type `void`. La liste exhaustive de ces contraintes se trouve à l'article **The Conditional Attribute** dans les **MSDN**. Voici un exemple d'utilisation de l'attribut `ConditionalAttribute` :

*Exemple 9-4 :*

```
//#define __TRACE__
class Program {
   [System.Diagnostics.Conditional("__TRACE__")]
   public static void Trace(string s) {
      System.Console.WriteLine(s) ;
   }
   static void Main() {
      Trace("Hello") ;
      System.Console.WriteLine("Bye") ;
   }
}
```

Ce code affiche ceci si la constante symbolique `__TRACE__` n'est pas définie.

```
Bye
```

Ce code affiche cela si la constante symbolique `__TRACE__` est définie.

```
Hello
Bye
```

## *Les directives #error et #warning*

La directive `#error` peut prévenir des conflits de définitions de constantes symboliques à la compilation.

Le programme suivant ne compilera pas, et le compilateur affichera l'erreur suivante : `MACRO1 et MACRO2 ne peuvent être définies en même temps`

*Exemple 9-5 :*

```
#define MACRO1
#define MACRO1
#define MACRO2
#if MACRO1 && MACRO2
#error MACRO1 et MACRO2 ne peuvent être définies en même temps
#endif
public class Program {
   public static void Main() {
#if (MACRO1)
      System.Console.WriteLine("MACRO1 définie. ") ;
#elif (MACRO2)
      System.Console.WriteLine("MACRO2 définie et MACRO1 non définie") ;
```

```
#else
      System.Console.WriteLine("MACRO2 et MACRO1 non définies") ;
#endif
   }
}
```

La directive #warning fonctionne sur le même principe, mis à part qu'elle n'arrête pas la compilation mais génère un avertissement.

## *Les directives #pragma warning disable et restore*

Les directives #pragma warning disable et #pragma warning restore permettent de désactiver la production d'avertissements du compilateur C#. Leur syntaxe d'utilisation est illustrée par l'exemple suivant :

*Exemple 9-6 :*

```
class Program {
   static void Main() {
#pragma warning disable 105
   // Désactive l'avertissement CS0105 dans ce bloc.
#pragma warning restore 105

#pragma warning disable
   // Tous les avertissements sont désactivés dans ce bloc.
#pragma warning restore

#pragma warning disable 105, 251
   // Les avertissements CS0105 et CS0251 sont désactivés dans ce bloc.
#pragma warning restore 105
   // Seul l'avertissement CS0251 est désactivé dans ce bloc.
#pragma warning restore
   // Tous les avertissements sont activés dans ce bloc.

#pragma warning disable
   // Désactive tous les avertissements jusqu'à la fin du fichier.
   }
}
```

## *La directive #line*

La directive #line permet au développeur de modifier la ligne et éventuellement le fichier où une erreur est déclarée par le compilateur. L'exemple suivant affiche que les erreurs de compilations trouvées sont à la ligne 1 dans le fichier Méthode Main() :

*Exemple 9-7 :*

```
class Program {
   public static void Main() {
      #line 1 "Méthode Main()"
```

```
        int i == 0 ;  // <- erreur ici : pas le droit d'utiliser '=='
    }
}
```

Attention à l'utilisation de cette directive car elle peut mettre en défaut certaines directives de l'environnement de développement. Notamment, l'environnement n'est plus capable de retrouver l'erreur puisqu'il ne dispose plus ni du fichier, ni de la ligne valide.

## *Les directives #region et #endregion*

Les directives `#region` et `#endregion` sont très pratiques car elles permettent de plier/déplier par un simple clic, une région de code. Cette possibilité est notamment utilisable dans les IDE *Visual Studio* et *SharpDevelop*. La Figure 9-2 illustre ces effets.

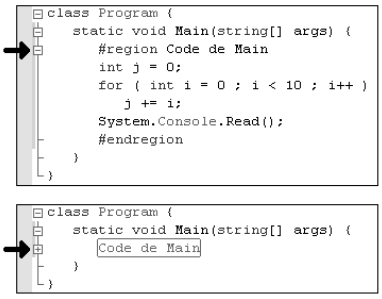


*Figure 9-2 : Effet des directives préprocesseur #region et #endregion*

`#endregion` peut optionnellement rappeler le même texte que `#region`. Il est également possible d'imbriquer ces directives.

# *Le compilateur csc.exe*

Le compilateur `csc.exe` peut être appelé soit directement en ligne de commande, soit par l'environnement de développement *Visual Studio*, soit par des scripts de compilation type MSBuild ou NAnt. Les options de compilation sont définies soit dans la ligne de commande, soit par l'intermédiaire de l'environnement soit au sein des scripts.

La Figure 9-3 illustre les entrées possibles et les types de fichiers de sortie possibles, de `csc.exe`. Un seul fichier est produit par compilation :

Présentons les options de compilation les plus utilisées à l'aide de quelques exemples :

- Compile `fichier.cs` et produit `fichier.exe` (notez que pour produire un exécutable il faut qu'au moins une méthode statique `Main()` existe dans au moins une classe).

```
>csc.exe fichier.cs
```

- Compile `fichier.cs` et produit `fichier.dll` (une méthode statique `Main()` n'est pas requise ici).

```
>csc.exe /target:library fichier.cs
```

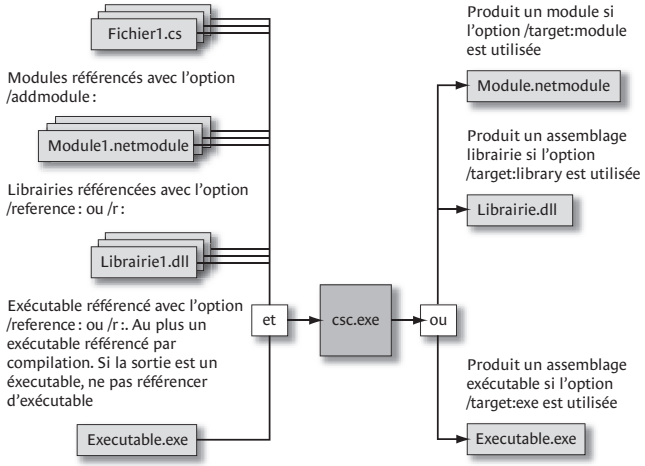


*Figure 9-3 : Entrées et sorties du compilateur C♯ csc.exe*

- Compile `fichier.cs` et produit `prog.exe`. La méthode `Main()` qui sert de point d'entrée est celle de la classe `Prog` qui est dans l'espace de noms `Namespace001`.

```
>csc.exe /out:prog.exe /main:Namespace001.Prog fichier.cs
```

- Compile avec optimisation tous les fichiers d'extension `.cs` dans le répertoire courant, définit la constante symbolique `MACRO1` pour chacun de ces fichiers et produit `prog.exe`.

```
>csc.exe /out:prog.exe /define:MACRO1 /optimize *.cs
```

- Compile en mode `debug` tous les fichiers d'extension `.cs` dans le répertoire courant, ne génère pas d'avertissement et produit le fichier `mod.netmodule`.

```
>csc.exe /target:module /out:mod.netmodule /warn:0 /debug *.cs
```

- Compile en tenant compte des références aux assemblages dont les modules avec le manifeste sont les fichiers `lib1.dll` et `lib2.dll`.

  Le module `mod.netmodule` sera un module de l'assemblage dont le module contenant le manifeste est `prog.exe`.

  Pour la compilation, les fichiers `lib1.dll` et `lib2.dll` peuvent se trouver dans le répertoire courant ou dans le répertoire de chemin `C:\`.

```
>csc.exe /lib:c:\ /r:lib1.dll;lib2.dll /addmodule:mod.netmodule
    /out:prog.exe fichier.cs
```

Nous avons illustré ici les options suivantes :

| Option | Description |
|---|---|
| `/target /t` | Spécifie le format du produit de la compilation. Le format peut être :`/target:library` Une librairie (extension `.dll`)`/target:module` Un module (extension `.netmodule`)`/target:exe` Un exécutable en mode console (extension `.exe`). C'est le format pris par défaut si l'option `target` n'est pas précisée.`/target:winexe` Un exécutable présenté dans une fenêtre (de type *Windows Form*). L'extension du fichier sera aussi `.exe`. |
| `/out` | Spécifie le nom du fichier produit par la compilation. |
| `/main /m` | Spécifie la classe qui contient la fonction `Main()` qui servira de point d'entrée. Attention, le nom de la classe doit comprendre les espaces de nom et doit respecter la casse. Cette option ne peut être utilisée que si l'on fabrique un exécutable. |
| `/define /d` | Définie une constante symbolique. |
| `/optimise /o` | Enclenche la compilation optimisée (compatible avec l'option `debug`). |
| `/warn /w` | Règle le niveau d'avertissement. Ce niveau varie entre 0 (pas d'avertissement) et 4 (tous les avertissements sont affichés). |
| `/debug` | Génère des informations nécessaires au déboguage, dans un fichier qui porte le même nom que le fichier produit, avec l'extension `pdb` (*programme database*). (Compatible avec l'option `optimise`). |
| `/addmodule` | Référence les modules utilisés par le produit de la compilation (un assemblage ou un module). Rappelons qu'à l'exécution, tous les modules d'un assemblage doivent se trouver dans le même répertoire que l'assemblage. |
| `/reference/r` | Référence les assemblages (exécutable ou librairies) utilisés par le produit de la compilation. On peut utiliser les ressources d'un assemblage référencé, dans le code de l'assemblage référençant. Dans ce cas, l'assemblage référencé sera chargé par le CLR lors de la première utilisation de l'une de ses ressources.Si le produit est un assemblage exécutable, il ne faut pas référencer un assemblage exécutable. Si le produit n'est pas un exécutable, on peut référencer au plus un assemblage exécutable. |
| `/lib` | Spécifie les répertoires où le compilateur peut chercher les fichiers référencés par l'option `/reference` ou `/r`. Ces fichiers sont cherchés dans l'ordre suivant :<br>• Dans le répertoire courant ;<br>• dans le répertoire du Common Langage Runtime ;<br>• dans les répertoires spécifiés par `/lib` ;<br>• dans les répertoires spécifiés par la variable d'environnement `LIB`. |

| | |
|---|---|
| `/resource/ linkresource` | Ajoute des ressources à l'assemblage. L'utilisation de ces options est détaillée page 37. |
| `/unsafe` | Indique que votre code peut contenir des zones de code non vérifiables par le CLR (voir page 501). |
| `/doc` | Cette option permet de produire un fichier XML contenant les informations présentes dans les commentaires /// du code source. Un exemple de l'utilisation de cette option est présenté un peu plus loin dans ce chapitre. |
| `/help /?` | Affiche l'aide. |

Une trentaine d'options sont disponibles. Nous vous avons présenté ici les plus courantes. L'article **C# Compiler Options Listed by Category** des **MSDN** fournit la liste exhaustive des options.

Toutes ces options sont aussi disponibles dans les propriétés d'un projet dans l'environnement de développement *Visual Studio*.

# *Les alias*

## *Alias sur les espaces de noms et sur les types*

Le mot clé `using` peut être utilisé pour définir un *alias* vers un espace de nom ou vers un type. La portée d'un tel alias est le fichier courant si il est défini hors de tout espace de noms. Dans le cas contraire, la portée d'un alias est le fichier courant intersection l'espace de nom dans lequel il est définit.

*Exemple 9-8 :*

```
//Définition de l'alias 'C' vers le type 'System.Console'.
using C = System.Console;
class Program{
   static void Main(){
      C.WriteLine("Hello 1");
   }
}
```

## *Qualificateur d'alias d'espace de noms*

Il peut y avoir un conflit entre un espace de noms et un alias. Les trois assemblages suivants illustrent ce cas :

*Exemple 9-9 :* *Code de `Asm1.dll`*

```
namespace Foo.IO{ public class Stream{} }
```

*Exemple 9-10 :* *Code de `Asm2.sll`*

```
namespace Custom.IO{ public class Stream{} }
```

*Exemple 9-11 : Code de Program.exe qui référence Asm1.dll et Asm2.dll*

```
using FooIO = Foo.IO;
using CusIO = Custom.IO;
class Program{
   static void Main(){
      FooIO.Stream stream1 = new FooIO.Stream();
      CusIO.Stream stream2 = new CusIO.Stream();
   }
}
```

Grâce à l'utilisation des alias `FooIO` et `CusIO` vous pouvez utiliser les deux types `Stream` dans `Program.exe` sans avoir à réécrire les espaces de noms en entier. Cependant ce programme ne compile plus si vous modifiez le code de `Asm2.dll` comme ceci :

*Exemple 9-12 : Code de Asm2.dll*

```
namespace Custom.IO{ public class Stream{} }
namespace FooIO{ public class Stream{} }
```

En effet, le compilateur sera incapable de déterminer si `FooIO.Stream` désigne le type `FooIO.Stream` défini dans `Asm2.dll` ou le type `Foo.IO.Stream` de `Asm1.dll` aliasé `FooIO.Stream`..

Pour éviter que l'évolution d'une bibliothèque empêche votre code de compiler vous pouvez avoir recours au *qualificateurs d'alias d'espace de noms* ::. Le compilateur infère que l'identificateur à gauche du qualificateur d'alias d'espace de noms est un alias. Dans notre exemple, il faut donc réécrire `Program.cs` comme ceci :

*Exemple 9-13 : Code de Program.exe qui référence Asm1.dll et Asm2.dll*

```
using FooIO = Foo.IO;
using CusIO = Custom.IO ;
class Program {
   static void Main(){
      FooIO::Stream stream1 = new FooIO::Stream();
      CusIO.Stream stream2 = new CusIO.Stream() ;
   }
}
```

Cette technique du qualificateur d'alias d'espaces de noms est efficace car la portée d'un alias est au plus le fichier dans lequel il est définit. Une évolution dans un assemblage externe ne peut donc pas introduire subrepticement une erreur de compilation.

## *Le qualificateur global*

Dans certains projets volumineux il se peut que vous ayez un conflit entre le nom d'un espace de noms et le nom d'une ressource. Le programme suivant ne compile pas :

*Exemple 9-14 :*

```
using System ;
class Program {
   class System { }
```

```
   const int Console = 691 ;
   static void Main() {
      // KO : Le compilateur essaye d'accéder à Program.Console.
      Console.WriteLine("Hello 1") ;
      // KO : Le compilateur essaye d'accéder Program.System.
      System.Console.WriteLine("Hello 2") ;
   }
}
```

C#2 introduit le qualificateur `global` qui, placé devant un qualificateur d'alias d'espaces de noms, indique au compilateur que l'on souhaite utiliser un espace de noms. L'exemple précédent doit donc être réécrit comme suit :

*Exemple 9-15 :*

```
using System ;
class Program{
   class System { }
   const int Console = 691 ;
   static void Main(){
      global::System.Console.WriteLine("Hello 1");
      global::System.Console.WriteLine("Hello 2");
   }
}
```

## *Les alias externes*

Les *alias externes* servent à utiliser simultanément deux types déclarés dans deux assemblages différents mais qui ont le même nom et qui sont dans le même espace de noms. Cette situation peut par exemple survenir si vous devez utiliser dans le même programme deux versions d'un même assemblage. La syntaxe des alias externes est illustrée par cet exemple :

*Exemple 9-16 : Code de `Asm1.dll`*

```
namespace FooIO{ public class Stream{} }
```

*Exemple 9-17 : Code de `Asm2.sll`*

```
namespace FooIO{ public class Stream{} }
```

*Exemple 9-18 : Code de `Program.exe` qui référence `Asm1.dll` et `Asm2.dll`*

```
extern alias AliasAsm1;
extern alias AliasAsm2;
class Program{
   static void Main(){
      AliasAsm1::FooIO.Stream stream1 = new AliasAsm1::FooIO.Stream();
      AliasAsm2::FooIO.Stream stream2 = new AliasAsm2::FooIO.Stream();
   }
}
```

Il faut alors compiler `Program.exe` comme ceci :

```
>csc.exe /r:AliasAsm1=Asm1.dll /r:AliasAsm2=Asm2.dll Program.cs
```

Vous pouvez aussi préciser vos alias externes dans *Visual Studio* grâce à la propriété `Aliases` dans les propriétés des références vers les assemblages bibliothèques. Vous y remarquerez qu'une référence vers un assemblage peut supporter plusieurs alias.

Enfin, précisons que le meilleur moyen d'utiliser ces qualificateurs et autres alias est de ne pas en avoir besoin en essayant d'éviter les conflits de noms des entités de vos programmes. En tant qu'artefacts avancés de C♯2, la majorité des développeurs ne les maîtrisent pas et auront ainsi des difficultés à relire du code qui les utilise.

# *Commentaires et documentation automatique*

C++ → C♯ Les commentaires en C♯ et C++ sont déclarés de la même façon. Cependant C♯ accepte un nouveau type de commentaires destiné à la documentation du code. Ces nouveaux commentaires sont déclarés avec la balise `///`.

La nouveauté dans les identificateurs (i.e les noms de variables, les noms des méthodes, les noms des classes...) est que l'on peut utiliser des lettres accentuées.

## *Les commentaires*

C♯ Il existe trois façons de commenter du texte dans un fichier source C♯ :

- Le texte placé entre les balises `/*` suivie de `*/` est commenté. Ces balises peuvent éventuellement se trouver sur deux lignes différentes.
- Si une ligne contient la balise `//` alors le texte de cette ligne qui suit cette balise est commenté.
- Si une ligne contient la balise `///` alors le texte de cette ligne qui suit cette balise est commenté. De plus ce texte fera partie de la documentation automatique du code source, présentée plus loin.

Un commentaire de type `/*...*/` ne peut être imbriqué dans un autre commentaire `/*...*/`. En revanche, un commentaire de type `//` ou `///` peut être imbriqué dans un commentaire de type `/*...*/`.

En conséquence, une bonne ligne de conduite est d'utiliser les commentaires de type `//` ou `///` pour commenter le code, et d'utiliser les commentaires de type `/*...*/` pour désactiver temporairement une région du code.

## *Utiliser la liste des tâches de Visual Studio*

Bien souvent, en tant que développeur nous sommes obligés de repousser l'écriture d'une portion de code. Sous *Visual Studio*, si un commentaire de type `//` ou `/* */` commence avec l'un des mots `TODO`, `HACK` ou `UNDONE` il est automatiquement ajouté dans la liste des tâches. Pour avoir accès à cette fonctionnalité, il faut sélectionner le menu *Afficher* ▶ *liste des tâches* puis sélectionner *Commentaires* dans la combo box de la liste des tâches qui apparaît.

## *La documentation automatique*

C# offre la possibilité de produire un document à partir des commentaires d'un code source marqués avec la balise ///. Cette possibilité est très intéressante car :

- Dès qu'un projet atteint une certaine taille, les seuls commentaires dans le code ne suffisent pas à donner une vue d'ensemble du projet. On est obligé d'avoir de la documentation associée au projet.
- Documenter du code est une tâche longue et fastidieuse. L'expérience montre qu'avec le temps les développeurs négligent la maintenance de la documentation technique d'un projet. Seules les entreprises qui peuvent se permettre d'avoir un département dédié à la documentation technique parviennent à maintenir correctement la documentation d'un projet.
- Les développeurs sont bien souvent les personnes les mieux placées pour commenter leur code.

Le fait que la documentation technique d'un projet soit un processus parallèle au développement du projet, implique la désynchronisation inéluctable de la documentation. La possibilité de générer automatiquement la documentation à partir du code source résout ce problème puisque la documentation technique est intégrée au développement du projet. Concrètement, dès que le développeur déclare une classe ou une méthode, il crée la documentation technique associée au même endroit (et au même moment).

La production automatique de la documentation technique se fait en deux étapes :

- Il faut d'abord extraire et hiérarchiser les informations précisées par les commentaires /// du code source. Ces informations sont alors stockées dans un document XML. Ces commentaires contiennent eux-mêmes des balises XML qui se retrouveront directement dans le document XML généré. Notez que *Visual Studio 2005* vous aide dans la production de ces balises avec l'intellisense. Précisons que les commentaires issus de la documentation automatique se retrouvent aussi dans les *tooltips* de *Visual Studio* concernant les entités commentées.
- Appliquer une feuille de style au fichier XML afin d'obtenir une présentation adaptée à la lecture. Cette feuille de style est en général une transformation XSLT. La présentation finale est en général une arborescence de fichiers HTML.

La Figure 9-4 résume ceci :

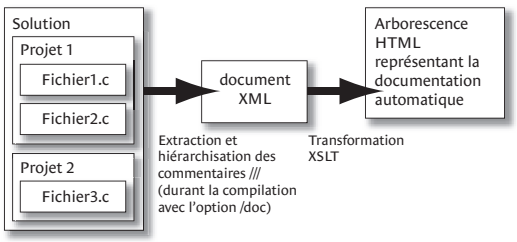


*Figure 9-4 : Production automatique de la documentation technique*

Illustrons ceci avec un exemple. Le fichier de code source C# suivant...

*Exemple 9-19 :*

```
namespace MonEspaceDeNoms {
   /// <summary>
   /// MaClass illustre la production automatique de
   /// la documentation technique
   /// </summary>
   class MaClass {
      /// <summary>
      /// Le point d'entrée de l'application
      /// </summary>
      static void Main() {}
      /// <summary>
      /// La fonction f(int)
      /// </summary>
      /// <param name="i">Un entier</param>
      static void f(int i){}
   }
}
```

...produit le fichier XML suivant :

```
<?xml version="1.0"?>
<doc>
   <assembly>
      <name>AutomaticDocTest</name>
   </assembly>
   <members>
      <member name="T:MonEspaceDeNoms.MaClass">
         <summary>
         MaClass illustre la production_automatique de
         la documentation technique
         </summary>
      </member>
      <member name="M:MonEspaceDeNoms.MaClass.Main(System.String[])">
         <summary>
         Le point d'entrée de l'application
         </summary>
      </member>
      <member name="M:MonEspaceDeNoms.MaClass.f">
         <summary>
         La fonction f(int)
         </summary>
      <param name="i">Un entier</param>
      </member>
   </members>
</doc>
```

Notez la présence des balises <summary> et <param> à la fois dans le code source C# et dans le fichier XML. Vous avez deux manières de produire ce fichier XML :

- Soit vous utilisez l'option `/doc` du compilateur C# `csc.exe` :

```
>csc.exe MonFichier.cs /doc:MonFichierDoc.XML
```

- Soit vous précisez dans les propriétés du projet dans *Visual Studio* que vous souhaitez produire un fichier XML de documentation lors de la compilation. Pour cela il faut préciser le nom du fichier XML dans l'option *Fichier de documentation XML* de la fenêtre *Générer* des propriétés du projet.

À partir du fichier XML, vous pouvez appliquer une feuille de style pour produire la documentation technique au format que vous souhaitez. *Visual Studio 2003* présentait un outil de création de pages HTML à partir de tels documents XML Avec *Visual Studio 2005*, nous vous recommandons d'utiliser des outils spécialisés effectuer cette tâche. Citons notamment l'excellent outil *NDoc* qui est open source et téléchargeable gratuitement.

# *Les identificateurs*

Les identificateurs sont des noms choisis par le développeur qui nomment des ressources telles que des espaces de noms, des classes, des méthodes de classes, des champs de classes et en fait, tout ce qui peut être nommé dans le code source.

Un identificateur doit obéir aux règles suivantes :

- Le premier caractère est soit une lettre (A à Z ou a à z ou une lettre accentuée UNICODE) soit le caractère de soulignement _ soit le caractère @. Le premier caractère ne peut être un chiffre.
- Les autres caractères sont dans l'ensemble des caractères cités (à part @), unis avec l'ensemble des chiffres (0 à 9).
- Un identificateur ne peut contenir plus de 255 caractères.
- Un identificateur ne peut être un mot-clé C#.

Le compilateur C# tient compte de la casse (i.e la différence majuscule/minuscule).

## *Convention de nommage CamelCase et PascalCase*

La convention de nommage *PascalCase* encourage le développeur à nommer ses identificateurs avec des lettres minuscules sauf pour les premières lettres des mots. Par exemple : `MaVariable`, `UneFonction`, `UnIdentificateurPascalCase`.

La convention de nommage *CamelCase* est similaire à PascalCase mis à part que le premier caractère est en minuscule. Par exemple : `maVariable`, `uneFonction`, `unIdentificateurPascalCase`. Le nom CamelCase provient des « bosses » provoquée par les caractères majuscules qui rappèlent celles d'un chameau (*camel* veut dire chameau en anglais). Il n'y a pas de relation avec le langage de programmation *Camel*.

En C#, il est conseillé par *Microsoft* d'utiliser la convention de nommage PascalCase pour les identificateurs représentant des noms de méthodes, d'événements, de champs, de propriétés, de constantes, d'espaces de noms, de classes, de structures, de délégations, d'énumérations, d'interfaces et d'attributs.

La convention CamelCase doit alors s'appliquer aux noms de variables et de paramètres de méthodes.

Il est aussi conseillé de faire commencer le nom d'un champ privé par `m_` et de faire commencer le nom d'une interface par un `I` majuscule.

# *Les structures de contrôle*

C++ → C# En ce qui concerne les structures de contrôle, il y a peu de différences entre C++ et C#.

L'utilisation de l'instruction `switch` a un peu changé. Le type de boucle, `foreach`, spécialement adapté au parcours d'éléments d'un tableau, fait son apparition.

C# Une structure de contrôle est un élément du programme qui change le comportement par défaut de l'unité d'exécution (du thread). Rappelons que ce comportement par défaut est d'exécuter les instructions les unes à la suite des autres. En programmation, les structures de contrôle se déclinent généralement en trois familles :

- Les *conditions*, qui exécutent (ou pas) un bloc de code qu'à une certaine condition, portant généralement sur les états de variables et d'objets.
- Les *boucles*, qui exécutent en boucle un bloc d'instructions. Le programmeur a le choix entre terminer de boucler après un certain nombre d'itérations, ou terminer de boucler à une certaine condition, voire ne jamais terminer (boucle infinie).
- Les *branchements* ou *sauts*, qui permettent de rediriger directement vers une instruction particulière l'unité d'exécution. Cependant ce type de structures de contrôle est à proscrire car il complexifie grandement la maintenance du code. De plus il est démontré qu'on peut toujours se passer de branchements dans du code source C#.

Les appels de méthodes modifient eux aussi le comportement par défaut de l'unité d'exécution, d'exécuter les instructions les unes à la suite des autres. Cela peut s'assimiler à un saut, à la différence fondamentale qu'à la fin de la méthode, l'unité d'exécution est capable de retourner à l'instruction située juste après l'appel de méthode. Ce comportement fait qu'en général on ne classe pas un appel de méthode dans les structures de contrôles.

## *Les conditions (if/else, ?:, switch)*

C++ → C# En C# il n'est pas possible d'écrire `if(i=1)` au lieu de `if(i==1)`. L'opérateur d'affectation ne renvoie pas un booléen. Cette erreur est particulièrement dangereuse en C++, car elle est non détectable par le compilateur.

L'instruction `switch` subit aussi quelques différences expliquées plus loin.

Aucun autre changement n'est à noter sur les tests et conditions de type `if/else` entre C/C++ et C#.

### *Utilisation de if/else*

C# Une condition se présente sous cette forme :

```
if ( expression retournant un booléen )
   Bloc d'instructions à exécuter si l'expression retourne true
else
   Bloc d'instructions à exécuter si l'expression retourne false
```

L'ensemble « *else et son bloc d'instructions* » est optionnel. Un bloc d'instructions peut être une seule instruction ou plusieurs instructions, auquel cas il faut placer les accolades qui définissent le bloc d'instructions :

```
if ( expression retournant un booléen )
   i = j*5 ;
else{
   // Commencement du bloc d'instructions à exécuter
   // si la condition est fausse.
   i = j*2 ;
   j++ ;
} // Fin du bloc d'instructions à exécuter si la condition est fausse.
```

Pour les lecteurs non habitués, il faudra faire attention lors de la lecture du code, au cas où il n'y aurait qu'une instruction. Vous pouvez utiliser les accolades, même dans le cas où il n'y a qu'une instruction. En fait, il est conseillé de toujours utiliser les accolades pour améliorer la lisibilité du code.

Une condition est considérée comme une instruction. Il est donc tout à fait possible d'imbriquer les conditions :

```
if( expression1 retournant un booléen )
   if( expression2 retournant un booléen )
      Bloc à exécuter si expression1 et expression2 sont true
   else  // se rapportant à l'expression2
      Bloc à exécuter si expression1 true et expression2 false
else // se rapportant à l'expression1
   if( expression3 retournant un booléen ) // pas de bloc else pour ce if
     Bloc d'instructions à exécuter si expression1 false et expression3 true
```

Ceci nuit gravement à la qualité du code. De plus les conditions imbriquées sont, en général, issues d'une mauvaise conception.

## *Expressions qui retournent un booléen*

Voici quelques exemples d'expressions, qui retournent un booléen :

```
bool b = true ; int  i = 5 ; int  j = 8 ;
if( b )                 // si b vaut true alors...
if( !b )                // si b vaut false alors...
if( b == true )         // si b vaut true alors...
if( b == false)         // si b vaut false alors...
if( i )                 // si i différent de 0  alors...
if( !i )                // si i égal à 0 alors...
if( i == 4)             // si i égal 4 alors...
if( i != 4 )            // si i différent de 4 alors...
if( i < 4)              // si i strictement inférieur à 4 alors...
if( i <= 4)             // si i inférieur ou égal à 4 alors...
if( i < 4 && j > 6)     // si i strictement inférieur à 4 et j strictement
                        // supérieur à 6 alors...
if( i >= 4 && i<= 6)    // si i dans l'intervalle fermé [4,6] alors...
if( i < 4 || j > 6)     // si i strictement inférieur à 4 ou j strictement
```

```
                        // supérieur à 6 alors...
if( i < 4 || i > 6)     // si i hors de l'intervalle fermé [4,6] alors...
if( i != 4 || b )       // si i différent de 4 ou b est true alors...
```

### *La facilité d'écriture « ?: »*

Une facilité d'écriture est proposée :

```
condition ? Val retournée si condition true : Val retournée si condition false ;
```

On parle d'opérateur ternaire `?:`. En effet, ces le seul opérateur pour lequel trois opérandes sont prises en compte.

Voici quelques exemples d'utilisation :

```
bool b = true ;
int  i = 5 ;
int  j = 8 ;

// k1 = i si b true, sinon k = j
int k1 = b  ?  i  : j ;

// k2 = 6 si i différent de j, sinon k2 = 7
int k2 = (i!=j) ?  6 : 7 ;

// s="bonjour" si i strictement inférieur à j, sinon s="hello"
string s = i<j ? "bonjour" : "hello" ;

// k3 = i*2 si i supérieur ou égal à j, sinon k3 = i*3
int k3 = i>=j ? i*2 : i*3 ;
```

### *L'instruction switch*

C++ → C# Programmeur C/C++ ATTENTION ! Il y a de subtiles modifications en ce qui concerne l'utilisation du mot-clé `switch` :

- Vous pouvez toujours « switcher » sur une variable de type entier, booléen, énumération. La nouveauté C# est que vous pouvez « switcher » sur une chaîne de caractères.
- La continuation vers le mot-clé `case` suivant, ne se fait pas automatiquement, donc le mot-clé `break` est obligatoire. La continuation vers le mot-clé `case` suivant se fait automatiquement lorsqu'il n'y a pas d'instructions pour le cas présent.
- Vous pouvez déclarer des variables à l'intérieur d'un bloc d'instructions, dans un bloc d'instructions `case`, même si celui-ci n'est pas entre accolades.

C# Tout comme les mots-clés `if/else`, le mot-clé `switch` permet de modifier le cours du programme en fonction de la valeur d'une variable. Cependant, l'utilisation de `switch` est particulièrement adaptée aux types à valeurs discrètes (entiers, énumérations, string) et sa syntaxe permet de traiter plusieurs cas de valeurs, plus facilement que les mots-clés `if/else`. Voici un exemple :

*Exemple 9-20 :*

```
class Program {
   static void Main() {
      int i = 6 ;
      switch (i){
         case 1:
            System.Console.WriteLine("i vaut 1") ;
            break ;
         case 6:
            System.Console.WriteLine("i vaut 6") ;
            break ;
         default:
            System.Console.WriteLine("i ne vaut ni 1 ni 6") ;
            break ;
      }
   }
}
```

Deux mots-clés apparaissent dans cet exemple, en plus des mots-clés `switch` et `case` :

- `break` : lorsque l'unité d'exécution rencontre l'instruction `break`, elle continue son cours directement à la fin du `switch`. Notez que si un bloc de code `switch` contient au moins une instruction, il doit se terminer soit par l'instruction `break` soit par une instruction `goto` soit par une instruction `return`. Dans le cas contraire, le compilateur signale une erreur.
- `default` : l'unité d'exécution se branche sur le bloc d'instructions `default` si la valeur de la variable est différente de toutes les valeurs spécifiées dans les blocs `case`. Notez que le bloc `default` n'est pas nécessairement en dernière position, bien que la plupart des développeurs l'utilisent comme cela.

Vous avez la possibilité d'exécuter le même bloc d'instructions pour plusieurs valeurs. Par exemple :

*Exemple 9-21 :*

```
class Program {
   static void Main() {
      int i = 6 ;
      switch (i){
         case 1:
         case 3:
         case 6:
            System.Console.WriteLine("i vaut 1 ou 3 ou 6") ;
            break ;
         default:
            System.Console.WriteLine("i ne vaut ni 1 ni 3 ni 6") ;
            break ;
      }
   }
}
```

Dans ce cas, il est impératif qu'aucune instruction n'apparaisse après `case 1:` ou `case 3:`.

Enfin on peut utiliser aussi l'instruction de branchement `goto` (décrite un peu plus loin) mais ceci est complètement à proscrire. Le cas suivant montre que l'on peut faire du code difficilement compréhensible en quelques lignes :

*Exemple 9-22 :*

```
class Program {
   static void Main() {
      int i = 6 ; int j = 7 ;
      switch (i){
         case 1:
            System.Console.WriteLine("passage par case 1") ;
            goto default ;
         case 6:
            System.Console.WriteLine("passage par case 6") ;
            if (j > 2) goto case 1 ;
            break ;
         default:
            System.Console.WriteLine("passage par default") ;
            break ;
      }
   }
}
```

Ce programme affiche :

```
passage par case 6
passage par case 1
passage par default
```

Enfin, sachez que les types possibles de la variable sur laquelle agit un `switch` sont :

- Les types entiers `sbyte byte short ushort int uint long ulong`.
- Les booléens, le type `bool`.
- Les énumérations.
- Les chaînes de caractères, le type `string`. Notez que dans ce cas, si une instruction `switch` a plus de 6 blocs `case` le compilateur C#2 utilise une table de hachage pour éviter de trop nombreuses comparaisons de chaînes de caractères.

## *Les boucles (do,while,for,foreach)*

C++ → C# Un nouveau type de boucle apparaît, les boucles `foreach`, spécialement adaptées pour parcourir les éléments d'un tableau. Cependant, toutes les possibilités séduisantes de l'utilisation de `foreach` ne sont vraiment détaillées qu'à la section page 563.

Les trois autres types de boucles en C/C++, `do/while` , `while` et `for` ont une utilisation similaires en C#, de même que pour les mots-clés `continue` et `break`.

C# On parle de *boucles* (*loop* en anglais), lorsqu'un bloc d'instruction (entre accolades) ou une instruction (pas forcément entre accolades) est exécuté plusieurs fois consécutivement.

## *Les boucles de type while et do/while*

Dans ces deux types de boucles, le fait que le programme sorte de la boucle dépend d'une condition (similaire à celles vues dans la section précédente). Par exemple :

*Exemple : Boucle while*

```
int i=0;int j=8 ;
while( i < 6 && j > 9) {
   i++;j-- ;
}
```

*Boucle do/while*

```
int i=0;int j=8 ;
do {
    i++;j-- ;
}
while( i < 6 && j > 9)
```

La seule différence entre ces types de boucle est mise en évidence dans cet exemple. Les boucles `do/while` exécutent au moins une fois le bloc d'instructions. Si la condition est fausse en entrant dans le `while`, les boucles de type `while` n'exécutent pas le bloc d'instructions.

## *Les boucles for*

Les boucles `for` se présentent ainsi :

```
for(Instruction(s) d'initialisation (séparées par des virgules) ;
    Condition de sortie de boucle (sort si false) vérifiée à chaque
        début de boucle ;
    Instruction(s) effectuée(s) à chaque fin de boucle(séparée par
        des virgules))
  Bloc d'instructions à exécuter à chaque boucle
```

La condition est similaire à celles vues dans la section précédente. Voici quelques exemples :

```
for(int i = 1 ; i<=6 ; i++) ...
//-------------------------------------------------------------------
int i = 3;int j=5 ;
for( ; i<7&& j>1 ;i++ , j--)...
//-------------------------------------------------------------------
for( int i =6 ; i<9 ; ) { ... i++;}
```

Les variables déclarées dans les instructions d'initialisation de la boucle `for` doivent avoir des noms différents des variables extérieures et intérieures à la boucle. Ces variables ne seront plus visibles dès la sortie de la boucle.

## *Les boucles foreach*

Les boucles de type `foreach` sont spécialement adaptées au parcours des éléments d'un tableau ou d'une collection. Par conséquent nous détaillons ceci lors de la présentation des tableaux, dans la section page 563.

## *Les instructions break et continue*

Deux instructions existent pour modifier le cours d'une boucle (de type `while do/while for` ou `foreach`).

- L'instruction `continue` fait passer directement à l'itération suivante de la boucle.

- L'instruction `break` fait quitter la boucle. Cette instruction fait aussi quitter un bloc `switch` comme on l'a vue précédemment.

Dans le cas de boucles imbriquées, ces deux instructions s'appliquent à la boucle la plus proche d'eux.

*Exemple 9-23 :*

```
class Program {
   static void Main() {
      for (int i = 0 ; i < 10 ; i++){
         System.Console.Write(i) ;
         if (i == 2) continue ;
         System.Console.Write("C") ;
         if (i == 3) break ;
         System.Console.Write("B") ;
      }
   }
}
```

Ce programme affiche :

```
0CB1CB23C
```

Les instructions `break` et `continue` ont tendance à compliquer la lisibilité du code. Le résultat de l'exemple précédent, n'est évident pour personne. Il faut donc les utiliser le moins souvent possible.

L'instruction `break` peut aussi être utilisée pour interrompre ce qu'on appelle des boucles infinies, c'est-à-dire des boucles de type `for`, `do/while` et `while`, dont la condition de sortie est toujours vraie. Voici des exemples de boucles infinies :

```
for(;;) {...}
for(;true;) {...}
while(true) {...}
do{...}
while(true) {...}
```

## *Boucles et optimisations*

Du fait que le code contenu dans une boucle est potentiellement sujet à un grand nombre d'exécutions, il peut être efficace d'essayer de l'optimiser. Voici quelques conseils :

- Si vous détectez des appels à des méthodes qui nécessitent beaucoup de passages d'arguments, il peut être efficace de copier le corps de la méthode dans la boucle (on parle d'*inlining*). Remarquez qu'en page 112, nous expliquons que parfois le compilateur JIT du CLR est capable d'effectuer une telle optimisation.
- Si vous accédez à une propriété d'un objet dont vous savez que la valeur retournée restera constante durant toute la boucle, il est efficace de stocker au préalable ces valeurs constantes dans des variables locales.
- Penser à utiliser la classe `StringBuilder` plutôt que la classe `String` pour fabriquer une chaîne de caractères dans une boucle.

- Si vous avez le choix, ayez recours à des boucles plutôt qu'à la récursivité.
- Si vous avez à tester plusieurs conditions de sortie d'une boucle, il est efficace de tester en premier la condition de sortie la plus probable.
- Bien qu'en général moins pratique d'utilisation, les boucles `for` sont légèrement plus efficaces que les boucles `foreach`.

Lorsque vous réalisez des optimisations, essayez surtout de quantifier le gain de performance apporté. En effet, dans un environnement géré par le CLR, certaines de vos optimisations peuvent gêner le CLR et son compilateur JIT pour finalement s'avérer contre productives.

## *Les branchements (goto)*

L'instruction de branchement `goto` a eu son heure de gloire il y a bien longtemps. L'utilisation de l'instruction `goto` ne doit se faire que dans certains cas très particuliers. Par exemple, l'instruction `goto` peut se révéler fort utile dans le code d'un compilateur ou dans du code généré. Certains prônent l'utilisation de `goto` pour coder une *machine à état*. Dans ce cas il vaut mieux ne pas l'utiliser et faire une machine à état à partir d'une utilisation de `switch` dans une boucle infinie.

L'instruction `goto` force l'unité d'exécution à continuer sur les instructions après une étiquette. Cette étiquette et le `goto` doivent être absolument dans la même méthode. De plus l'étiquette doit être visible du `goto`, ce qui ne signifie pas qu'elle doit être dans le même bloc d'instructions. Par exemple, le code suivant compile :

*Exemple 9-24 :*

```
class Program {
   static void Main() {
      int i = 0 ;
      goto label2;
   label1:
      i++ ;
      goto label3;
   label2:
      i-- ;
      goto label1;
   label3:
      System.Console.WriteLine(i) ;
   }
}
```

Il se peut qu'une mauvaise utilisation de `goto` amène à un cas de variable non initialisée. Le compilateur C# détecte ceci et produit une erreur.

# *La méthode Main()*

C++ → C# Comme en C/C++, en C# le point d'entrée d'un assemblage exécutable est une méthode appelée `Main()`.

Comme en C/C++, en C#, la méthode `Main()` peut retourner `int` ou `void` et accepte éventuellement un tableau de chaînes de caractères représentant les arguments en ligne de commande de l'exécutable. En C#, il n'est plus besoin de préciser la taille du tableau.

À la différence de C/C++ le tableau de chaînes de caractères ne contient pas le nom de l'exécutable en première occurrence, mais directement le premier argument.

À la différence de C/C++ le « m » de `Main` est une majuscule.

À la différence de C/C++ `Main` est une méthode statique d'une classe et non une fonction globale.

C# Chaque assemblage directement exécutable (i.e dont le module principal a une extension `.exe`) possède au moins une méthode statique `Main()` dans une de ses classes. Cette méthode représente le *point d'entrée du programme*, c'est-à-dire que juste après le lancement et l'initialisation d'une application .NET, le thread principal va commencer par exécuter le code de cette méthode. Lorsque cette méthode retourne, le processus est détruit à la condition qu'il n'y ait pas de threads de premier plan (thread foreground) qui soient toujours en cours d'exécution. Si ces notions de thread ou de thread foreground vous sont étrangères, sachez qu'elles sont présentées au début du chapitre 5.

Un même assemblage peut avoir éventuellement plusieurs méthodes `Main()` (chacune dans une classe différente). Le cas échéant, il faut préciser au compilateur quelle méthode `Main()` constitue le point d'entrée du programme. Ce qui peut se faire soit avec l'option `/main` en ligne de commande du compilateur `csc.exe`, soit dans les propriétés du projet sous *Visual Studio*, *Application* ▶ *startup object*. Cette facilité est extrêmement utile pour déboguer une classe particulière.

Une méthode `Main()` est statique et sa signature suit les règles suivantes :

- Elle retourne le type `void` ou `int`.
- Elle accepte un tableau de chaînes de caractères facultatif en argument. Le cas échéant, les chaînes contiennent les arguments en ligne de commande de l'exécutable. La première chaîne de caractères représente le premier argument, la deuxième chaîne de caractères représente le deuxième argument ...

Voici différentes définitions possibles pour la méthode `Main()` :

```
static void Main() {/*...*/}
static int Main() {/*...*/}
static void Main(string[] args) {/*...*/}
static int Main(string[] args) {/*...*/}
```

Par exemple le programme suivant ajoute les nombres passés en arguments en ligne de commande, et affiche le résultat. S'il n'y a pas d'argument, le programme le signale :

*Exemple 9-25 :*

```
class Program {
   static void Main(string[] args) {
      if (args.Length == 0)
         System.Console.WriteLine("Entrez des nombres à ajouter.") ;
      else{
         long result = 0 ;
         foreach (string s in args)
            result += System.Int64.Parse(s) ;
```

```
            System.Console.WriteLine("Somme de ces nombres :{0}", result) ;
        }
    }
}
```

Les informations communiquées en ligne de commande (ainsi que les valeurs des variables d'environnement) peuvent être aussi récupérées grâce aux méthodes `string[] GetCommandLineArgs()` et `IDictionary GetEnvironmentVariables()` de la classe `System.Environment`.

# 10

# *Le système de types*

C++ → C♯ *Ce chapitre contient la plupart des grosses différences entre C/C++ et C♯.*

C♯ C♯ est un langage typé, c'est-à-dire que chaque objet a un type et un seul. Ce type est complètement défini au moment de la création de l'objet, à l'exécution. En C♯ chaque variable doit être initialisée, sinon le compilateur produira une erreur lors de son utilisation.

## *Stockage des objets en mémoire*

Les concepts de threads (i.e unité d'exécutions) et de processus (i.e tâches ou espace d'adressage) sont requis pour comprendre cette section, et plus généralement ce chapitre. Ces concepts sont introduits au début du chapitre 5.

### *Allocation/désallocation*

Lors de l'exécution d'un programme, le fait de réserver une zone mémoire pour qu'elle contienne les données relatives à un objet est nommé *allocation*. L'opération inverse de restitution de la zone mémoire est appelée *désallocation*. La taille de cette zone mémoire, spécifiée en octets, doit être au moins égale au nombre d'octets nécessaires pour stocker l'état de l'objet. Ce nombre d'octets est fonction de l'implémentation de l'objet.

Un processus peut à un instant donné contenir un ou plusieurs threads. En tant qu'espace d'adressage, le processus contient toutes les zones mémoires allouées pour tous les objets du programme. En tant qu'unités d'exécution, seuls les threads peuvent utiliser les objets.

### *La pile*

Chaque thread *Windows* a une zone mémoire privée que l'on nomme la *pile* (*stack* en anglais). Cette zone de mémoire est privée dans le sens où elle ne devrait pas être accessible par les autres threads (bien que ceci soit possible sous certaines conditions spéciales). Cette zone mémoire est

contenue dans l'espace d'adressage du processus du thread. Le thread se sert de sa pile principalement pour :

- stocker les valeurs des arguments de la fonction couramment exécutée ;
- stocker l'adresse (dans le code machine natif) à laquelle il faudra se brancher lorsque la fonction retournera ;
- stocker certains objets (mais pas tous).

Chaque thread a un accès privilégié à sa pile. En effet, les jeux d'instructions machine contiennent des instructions optimisées pour accéder à la pile. En outre le langage IL contient de nombreuses instructions dédiées à la gestion de la pile. Notez qu'une pile est de taille variable et bornée en général par une grandeur de l'ordre du Mo. Cette limite peut être définie lors de la construction du thread.

## *Le tas*

Un processus a généralement un seul (mais parfois plusieurs) *tas* (*heap* en anglais). C'est une zone mémoire contenue dans l'espace d'adressage du processus. Cette zone mémoire est accessible par tous les threads du processus. Donc, contrairement aux piles des threads d'un processus, le tas n'est pas spécifique à un thread particulier. Le tas est principalement utilisé pour stocker des objets et, à l'instar des piles, sa taille peut varier au cours du temps. Cependant la taille maximale du tas est beaucoup plus grosse que le Mo. En fait, il est assez singulier que le bon déroulement d'une application soit limité par la taille maximale du tas.

## *Comparaison pile/tas*

Un objet peut donc être stocké soit dans une pile d'un thread soit dans un tas d'un processus. Les notions de pile et de tas doivent coexister car chacune a ses avantages :

- L'avantage du tas est qu'il peut être beaucoup plus gros qu'une pile. De plus il est accessible par tous les threads du processus mais ceci n'est pas toujours un avantage.
- L'avantage de la pile est que l'accès aux données est plus rapide qu'avec le tas. Ce gain est dû à des instructions IL spéciales. Ce gain est aussi dû au fait que l'accès à la pile n'a pas à être synchronisé.

Il serait donc judicieux d'utiliser le tas pour stocker les objets volumineux et d'utiliser la pile pour stocker les objets de petite taille. Nous allons voir que c'est exactement ce choix qui a été fait par les concepteurs de .NET.

## *Allocations statiques et allocations dynamiques*

C++ → C♯ Un point commun de C++ et de C♯ est que les objets peuvent être alloués soit dans la pile d'un thread (on parle d'*allocation statique*) soit dans le tas du processus (on parle d'*allocation dynamique*). La différence entre C++ et C♯ est la façon dont le mode d'allocation (statique ou dynamique) est choisi :

- En C++ le choix du mode d'allocation d'une variable (d'un objet) est laissé au développeur. L'allocation statique est utilisée lorsque l'objet est directement déclarée dans le code (par exemple `int i=0;`) L'allocation dynamique est utilisée lorsque l'objet est allouée avec l'opérateur `new` (par exemple `int * pi = new int(0);`).

- En C♯ le choix du mode d'allocation d'un objet est fonction de son implémentation.. En effet, nous allons voir qu'il existe deux sortes de types. Les types valeur dont les instances sont allouées statiquement, et les types référence, dont les instances sont allouées dynamiquement.

Une autre différence importante entre C++ et C♯ est la responsabilité de la désallocation des variables dynamiques. En C++ cette responsabilité incombe au développeur alors qu'en C♯ elle incombe à une couche logicielle fournie par l'environnement d'exécution .NET. Cette couche est nommée *ramasse-miettes* et elle fait l'objet de la section 116. Dans tous les cas, cette responsabilité est lourde car si les variables allouées dynamiquement, devenues inutiles, ne sont pas régulièrement désallouées, la taille du tas croît en permanence et finira sûrement par causer des problèmes. Ce type de problème est connu sous le nom de *fuite de mémoire* (*memory leak* en anglais).

La désallocation des variables allouées statiquement est dans tous les cas sous la responsabilité du thread qui possède la pile concernée. Retenez surtout qu'**en C♯ la responsabilité du développeur est allégée par rapport au C++** :

- Il n'a pas à choisir le type d'allocation de ses variables.
- Il n'a pas à se soucier de la désallocation de ses variables.

# *Type valeur et type référence*

C++ → C♯ Si vous êtes programmeur C++ vous risquez d'être interpellé par ces notions de types valeur/référence. Nous allons voir qu'en C♯ l'opérateur `new` peut être utilisé pour allouer dynamiquement, mais aussi statiquement une variable. Une conséquence est que l'on ne peut plus fournir les arguments du constructeur directement après la déclaration d'une variable. Rappelons qu'en C++ l'utilisation de l'opérateur `new` est réservée pour allouer dynamiquement une variable.

C♯ La notion de type valeur/référence est fondamentale, quelque soit le langage .NET que vous utilisez.

Chaque type en C♯ est soit un *type valeur* (*value type* en anglais) soit un *type référence* (*reference type* en anglais). Chaque objet est donc soit l'instance d'un type valeur, soit l'instance d'un type référence. Une instance d'un type valeur est allouée sur la pile du thread (allocation statique), une instance d'un type référence est allouée sur le tas du processus (allocation dynamique).

On ne manipule jamais directement des objets de type référence. On manipule ces objets par l'intermédiaire de références. En revanche, on manipule toujours directement un objet de type valeur. Voici un exemple pour illustrer ce fait (nous anticipons un peu la notion de structures, qui définissent toujours des types valeur, et la notion de classes, qui définissent toujours des types référence) :

*Exemple 10-1 :*

```
// TypeVal est un type valeur, car c'est une structure.
struct TypeVal {
   public int m_i ;
   public TypeVal( int i ) { m_i = i ; }
}
// TypeRef est un type référence, car c'est une classe.
class TypeRef {
   public int m_i ;
   public TypeRef( int i ) { m_i = i ; }
}
class Program {
   static void Main() {
      TypeVal v1 = new TypeVal(6);
      TypeVal v2 = v1; // Une nouvelle instance du type TypeVal est
      // créée et le champ v2.i est aussi égal à 6.
      // Néanmoins v1 et v2 sont deux instances différentes
      // de type TypeVal.
      v2.m_i = 9;
      // Ici v1.i vaut 6, il y a bien deux instances du type TypeVal.
      System.Diagnostics.Debug.Assert( v1.m_i == 6 && v2.m_i == 9 ) ;

      TypeRef r1 = new TypeRef(6);
      TypeRef r2 = r1; // Il n'y a pas de nouvelle instance de la
      // classe TypeRef. r2 et r1 sont deux références de la
      // même instance de la classe TypeRef.
      r2.m_i = 9;
      // Ici r1.i vaut 9, il n'y a qu'une seule instance de la
      // classe TypeRef.
      System.Diagnostics.Debug.Assert( r1.m_i == 9 && r2.m_i == 9 );
   }
}
```

On s'aperçoit dans l'exemple précédent que l'opérateur `new` peut être éventuellement utilisé pour les allocations d'objets de type valeur mais ne modifie en rien le caractère statique de l'allocation. Dans ce cas, cet opérateur sert cependant à communiquer des paramètres au constructeur.

Contrairement au C++, lors d'une allocation statique (i.e d'un type valeur) en C# on ne peut fournir des arguments au constructeur sans utiliser l'opérateur `new`. En C++ ceci était accepté par le compilateur :
`MyType v1(6) ;`
En C# il faut écrire :
`MyType v1 = new MyType(6) ;`

Dans le cas d'une allocation dynamique, donc de l'allocation d'un objet de type référence, l'utilisation de l'opérateur `new` est obligatoire (y compris si le constructeur ne prend pas de paramètres). Nous allons détailler par la suite quels sont les types valeur et quels sont les types

référence, mais nous pouvons déjà préciser que les types valeur sont les types primitifs de C# (déclarés avec les mots-clés `int`,`double`...) les structures (déclarées avec le mot-clé `struct`) et les énumérations (déclarées avec le mot-clé `enum`) alors que les types référence sont les classes (déclarées avec le mot-clé `class`) et les délégations (qui sont des classes particulières déclarées avec le mot-clé `delegate`).

Les instances des types valeur ne sont pas toujours stockées sur la pile. En effet, lorsqu'un champ d'une instance de classe est de type valeur, il est stocké au même endroit que l'instance de la classe, c'est-à-dire sur le tas. En revanche, les objets de types référence sont toujours stockés sur le tas. Lorsqu'un champ d'une instance de structure est de type référence, seule la référence est stockée au même endroit que l'instance de la structure (sur la pile ou sur le tas selon les cas).

## *Notion de référence sur une instance de classe*

C++ → C# La notion de référence en C# est à mi-chemin entre la notion de pointeur et de référence de C++. Comme une référence C++, une référence C# référence un objet, et les membres publics de l'objet sont accessibles avec l'opérateur « . ».

À l'inverse d'une référence C++, et comme un pointeur C++, une référence C# peut être nulle.

À l'inverse d'une référence C++, et comme un pointeur C++, une référence C# peut être modifiée. C'est-à-dire que pour une référence donnée, l'objet référencé (pointé) n'est pas nécessairement le même au cours du temps.

À l'inverse d'un pointeur C++, et comme une référence C++, une référence C# ne donne pas accès à l'adresse physique de l'objet et ne permet aucune manipulation d'adresse (comme l'incrémentation vers l'objet suivant) de type « arithmétique des pointeurs ».

C# Toute classe est un type référence et tout type référence est une classe ou une interface. Toutes les classes héritent de la classe `System.Object` que nous allons bientôt décrire.

Il peut aussi y avoir une certaine ambiguïté entre les termes *type*, *classe* et *implémentation*. La signification du terme *type* varie selon le contexte. On parle de *type d'une référence* pour désigner la classe ou l'interface qui type une référence. On parle de *type d'un objet* pour désigner son implémentation. L'implémentation d'un objet désigne la classe ou la structure dont il est instance. Le terme *type* est donc plus général et plus abstrait que le terme *implémentation* qui lui même est plus général que le terme *classe*.

La notion de classe est un vaste sujet, qui fait l'objet du chapitre suivant. Nous allons nous intéresser ici au fait qu'en tant que type référence, les instances des classes, sont EXCLUSIVEMENT manipulés au travers de références. Etudions l'exemple suivant :

*Exemple 10-2 :*

```
class Personne {
   public int m_Age ;
   public string m_Nom ;
   public Personne(int Age, string Nom) { m_Age = Age ; m_Nom = Nom ; }
}
class Program {
   static void Main() {
      Personne ref1 = null ; // ref1 ne référence personne.
      Personne ref2 = new Personne(50, "Raymond") ;
```

```
        Personne ref3 = new Personne(48, "Josiane") ;
        ref1 = ref2 ;          // ref1 référence Raymond.
        ref1.m_Age += 10 ;     // Raymond est vieilli de 10 ans !
        ref1 = ref3 ;          // ref1 référence Josiane.
        ref1.m_Age += 10 ;     // Josiane est vieillie de 10 ans !
        // ici ref2.m_Age == 60 , Raymond a 60 ans.
        // ici ref3.m_Age == 58 , Josiane a 58 ans.
    }
}
```

Une classe est déclarée avec le mot-clé `class`. Avant tout, comprenez bien que `ref1` `ref2` et `ref3` sont trois références vers des objets, instances de la classe `Personne`. Au départ `ref1` est initialisée avec le mot-clé `null`. Cela signifie qu'aucun objet n'est référencé par `ref1`. À ce stade, aucun membre de `Personne` ne peut être utilisé sur `ref1`.

Deux objets `Personne` sont alors alloués (donc sont alloués dynamiquement sur le tas du processus puisqu'ils sont de type référence). Appelons-les `Raymond` et `Josiane`. Contrairement à ce que l'on a vu pour les types valeur, dans le cas de type référence, il est obligatoire d'utiliser l'opérateur `new` pour créer `Raymond` et `Josiane`. À ce stade nous disposons des deux objets `Raymond` et `Josiane` qui instancient la classe `Personne`. Nous disposons aussi de trois références, `ref1` qui est nulle, `ref2` qui référence `Raymond` et `ref3` qui référence `Josiane`. `ref1` référence alors `Raymond`. `Raymond` est vieilli de 10 ans. `ref1` référence alors `Josiane`. `Josiane` est vieillie de 10 ans.

Au final, les objets `Raymond` et `Josiane` ont tous les deux été modifiés sans passer par les références `ref2` et `ref3`. De plus les objets `Raymond` et `Josiane` ne sont jamais manipulés directement. La syntaxe de C# ne permet de manipuler des instances de types référence que par l'intermédiaire de références.

L'accolade de fin de la méthode `Main()` implique que les objets `Raymond` et `Josiane` ne sont plus référencés (en effet les références `ref1`, `ref2` et `ref3` n'existent plus). `Raymond` et `Josiane` ne seront plus jamais utilisés puisqu'ils ne sont plus référencés. Ainsi ces deux objets seront marqués comme non actifs par le prochain déclenchement du ramasse-miettes. Ils seront désalloués ultérieurement.

# *Le CTS (Common Type System)*

## *Les types .NET sont indépendants du langage*

Un aspect très intéressant de .NET est que les types utilisés sont indépendants du langage dans lequel nous écrivons notre code source. Ceci est possible grâce à un système de types commun à tous les langages .NET nommé le *CTS* (*Common Type System*). Le CTS est une spécification qui décrit les différentes caractéristiques des différents types connus par le CLR.

L'existence du CTS résout de très nombreux problèmes bien connus de ceux qui ont déjà développé des modules devant communiquer entre eux dans des langages différents. Par exemple les chaînes de caractères en VB sont représentées par des instances du type `BSTR` et en C++ par des pointeurs vers un tableau de `char` (et encore, ne mentionnons pas le type `string` de la STL et le type `CString` des MFC). La conséquence est qu'un module écrit en C++, utilisé par un programme écrit en VB, doit manipuler des instances de `BSTR` et utiliser des fonctions de conversion. Cela rajoute de la complexité aux programmes et a donc un coût non négligeable (voire prohibitif si en plus on mélange des formats d'encodages tels que ASCII ou UNICODE).

## *Vue d'ensemble du CTS*

Le CTS se décline en un ensemble de types valeur et référence. Le CTS vous laisse l'opportunité de définir vos propres types :

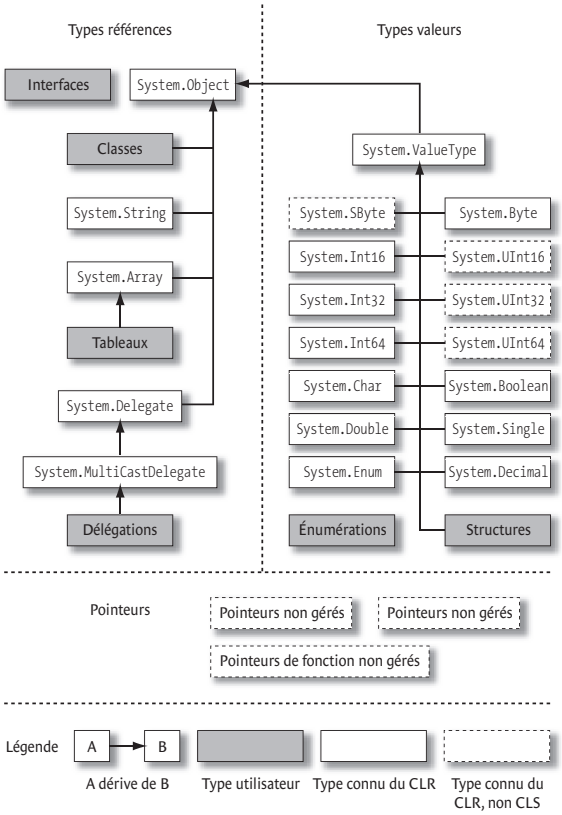


*Figure 10-1 : Vue d'ensemble du CTS*

Le reste de ce chapitre est dédié à l'étude des différents types du CTS que l'on peut classer comme ceci :

- Les *types primitifs* (appelés aussi *types élémentaires*) : Ces types représentent les entiers, les nombres à virgules, les caractères et les booléens. Ce sont des types valeur et en général les langages définissent des alias pour faciliter leur utilisation. Ainsi le type `System.Int16` correspond à l'alias `short` en C# et `Short` en VB.NET.
- Les *énumérations* : Ces types sont de type valeur et sont utilisés pour typer des ensembles de valeurs.
- Les *structures* : Ces types sont de type valeur. Les structures et les classes ont des similitudes et des différences.
- Les *classes* : Ces types sont de type référence. Notez que le type représentant les chaînes de caractères et le type représentant les tableaux sont respectivement les classes `System.String`

et System.Array (décrites respectivement page 370 et page 570). Notez que l'utilisation de la classe Array fait l'objet d'une syntaxe particulière en C# et en VB.NET.

- Les *délégations* : Ces types sont des classes particulières dont les instances sont utilisées pour référencer des méthodes.
- Les *pointeurs* : Ces types sont très spéciaux et utilisables seulement sous certaines conditions. Nous détaillons ce sujet dans la section en page 503.

# *La classe System.Object*

C++ → C# En C# toutes les classes et toutes les structures dérivent de la classe Object. Les méthodes de la classe Object, utilisables par toutes les classes et structures, ajoutent des fonctionnalités de hachage d'objet et de *RTTI* (*Run Time Type Information*). La fonctionnalité la plus utilisée est assurément la possibilité de redéfinir la méthode Object.ToString() qui est censée retourner une chaîne de caractères décrivant l'objet, un peu comme l'opérateur C++ « dans les flots de données.

C# Le mot clé object du langage C# est un alias vers System.Object. La vue d'ensemble du CTS montre que mis à part les interfaces et les types pointeurs, tous les types dérivent automatiquement et implicitement, directement ou indirectement de la classe Object. En ce qui concerne les interfaces on peut toujours convertir un objet référencé par une interface en une référence de type Object car une interface est toujours implémentée par un type référence ou valeur. Ainsi, la classe Object joue un rôle prépondérant dans l'architecture de la plateforme .NET. De nombreuses méthodes standard acceptent leurs arguments sous forme de références typées par la classe Object.

La classe Object présente plusieurs méthodes. Chacune de ces méthodes peut donc s'appliquer à tous les objets. Néanmoins il est logique de redéfinir certaines des méthodes virtuelles pour pouvoir les utiliser. Ces méthodes virtuelles peuvent être redéfinies dans le cadre d'une classe, mais aussi dans le cadre d'une structure. Dans le cas d'une énumération, seule la méthode virtuelle ToString() est automatiquement redéfinie par le compilateur. Les méthodes de la classe Object sont :

- `public Type GetType()`
  Renvoie le type d'un objet.
- `public virtual String ToString()`
  Renvoie une chaîne de caractères décrivant l'objet. Le comportement par défaut est de retourner le nom du type, ce qui n'est pas ce que souhaite le développeur la plupart du temps. Voici un exemple de redéfinition et d'utilisation de cette méthode :

  *Exemple 10-3 :*

```
class Personne { // On aurait pu mettre 'struct' à la place de 'class'.
   string m_Nom ;
   int m_Age ;
   public Personne(string Nom, int Age) { m_Nom = Nom ; m_Age = Age ; }
   public override string ToString() {
      return "Nom:" + m_Nom + " Age:" + m_Age ;
   }
}
```

```
class Program {
   static void Main() {
      Personne raymond = new Personne("Raymond", 50) ;
      // WriteLine() appelle automatiquement Raymond.ToString()
      System.Console.WriteLine(raymond) ;
   }
}
```

Ce programme affiche :

```
Nom:Raymond Age:50
```

`Console.WriteLine()` appelle automatiquement la méthode virtuelle `ToString()` des objets dont elle doit afficher l'état sur la console. Si cette méthode n'est pas redéfinie, c'est l'implémentation par défaut de la méthode `ToString()` de la classe `Object` qui est invoquée.

- `public virtual void Finalize()`
  Cette méthode est appelée lorsque l'objet est détruit par le ramasse-miettes. En C# il est impossible de redéfinir explicitement cette méthode. Pour la redéfinir, il faut utiliser une syntaxe particulière expliquée en page 423.
- `protected object MemberwiseClone()`
  Cette méthode est expliquée un peu plus loin lors de la présentation du clonage d'objet.
- `public static bool ReferenceEquals(object objA, object objB)`
- `public virtual bool Equals(object obj)`
- `public virtual int GetHashCode()`
  Ces trois méthodes sont expliquées dans la section suivante.

# *Comparer des objets*

## *Comparaison entre objets : identité vs. équivalence*

Par défaut il existe trois façons de comparer deux instances d'un même type référence : l'utilisation d'un des deux opérateurs « == » ou « =! », l'utilisation de la méthode d'instance `Object.Equals()` ou l'utilisation de la méthode statique `Object.ReferenceEquals()`. Dans ce dernier cas, la comparaison se fait sur les références et non sur l'état des objets. Deux références sont égales si elles référencent le même objet. C'est la comparaison selon l'*identité*.

Par défaut vous ne pouvez comparer deux instances d'un même type valeur qu'en utilisant la méthode virtuelle `Object.Equals()`. L'implémentation par défaut de cette méthode compare les instances champs à champs en ayant recours au mécanisme de réflexion. Deux instances d'un type valeur sont considérées comme égales si leurs champs sont égaux deux à deux. C'est la comparaison selon l'*équivalence*. L'exemple suivant illustre tout ceci :

*Exemple 10-4 :*

```
using System.Diagnostics ;
class  TypeRef { public int state;}
struct TypeVal { public int state;}
public class Program {
   public static void Main() {
```

```
        // Comparaison selon l'identité.
        TypeRef ref1 = new TypeRef() ; ref1.state = 3 ;
        TypeRef ref2 = new TypeRef() ; ref2.state = 3 ;
        Debug.Assert( ref1 != ref2 );
        Debug.Assert( ! ref1.Equals(ref2) );
        Debug.Assert( ! object.ReferenceEquals(ref1, ref2 ));
        ref2 = ref1;
        Debug.Assert( ref1 == ref2 );
        Debug.Assert( ref1.Equals(ref2) );
        Debug.Assert( object.ReferenceEquals(ref1, ref2) );

        // Comparaison selon l'équivalence.
        TypeVal val1 = new TypeVal() ; val1.state = 3 ;
        TypeVal val2 = new TypeVal() ; val2.state = 3 ;
        Debug.Assert(val1.Equals(val2));
        val1.state = 4;
        Debug.Assert(!val1.Equals(val2));
    }
}
```

Nous allons maintenant nous intéresser aux possibilités offertes par le *framework* .NET pour personnaliser la comparaison des instances de vos types.

## *Personnaliser le test d'égalité entre deux objets*

Si vous ne souhaitez que redéfinir le test d'égalité entre deux instances, il vaut mieux à la fois réécrire la méthode `Object.Equals()` et redéfinir les opérateurs « == » et « != ». L'implémentation de vos opérateur doit alors appeler seulement la méthode `Object.Equals()`. Plus d'information concernant la redéfinition des opérateurs sont disponibles en page 433.

Il est conseillé de redéfinir la méthode `Equals()` et les opérateurs d'égalité pour tous les types valeurs qui ont besoin d'avoir leurs instances comparées. Vous y gagnerez en performance car l'implémentation par défaut de la méthode `Equals()` pour les types valeur utilise la réflexion et n'est donc pas efficace.

Il est conseillé que l'implémentation de `Object.Equals()` définisse ce que l'on appelle une *relation d'équivalence* sur les instances du type concerné. Pour cela il faut que l'implémentation soit *réflexive* (i.e `x.Equals(x)` retourne `true` pour tout instance `x`) *symétrique* (i.e `x.Equals(y)` est égal à `y.Equals(x)` pour toutes instances `x` et `y`) et *transitive* (i.e si `x.Equals(y)` et `y.Equals(z)` sont égales à `true`, alors `x.Equals(z)` est égale à `true` pour toutes instances `x`, `y` et `z`).

Il est aussi conseillé que la réécriture de la méthode `Equals()` dans une classe `D` dérivée de la classe `B`, appelle la réécriture de la méthode `Equals()` définie dans la classe `B` si celle-ci existe.

En général, on décide de personnaliser le test d'égalité entre deux objets de type référence pour obtenir un comportement de comparaison par équivalence. Cette pratique est notamment illustrée par la façon dont les instances de la classe `System.String` sont comparées. En effet, nous verrons un peu plus loin que les chaînes de caractères se comparent selon l'équivalence.

### *Possibilité de stocker vos objets dans une table de hachage*

Si vous ne redéfinissez que la méthode `Object.Equals()` vous vous apercevrez que le compilateur C#2 émet un avertissement vous conseillant de réécrire la méthode `Object.GetHashCode()`. Cet avertissement est émis indépendamment du fait que vous réécrivez ou non les opérateurs d'égalité. En fait, vous ne devez suivre ce conseil que si les instances du type concerné peuvent potentiellement servir de clés dans une table de hachage. Plus d'information à ce sujet sont disponibles en page 584.

### *Personnaliser l'ordonnancement de vos objets*

En plus de pouvoir comparer l'égalité de vos instances, vous pouvez aussi désirer pouvoir les ordonner, par exemple pour les stocker dans une liste triée. Dans ce cas, vous avez le choix entre implémenter l'interface `System.IComparable<T>` sur le type des objets à comparer ou implémenter l'interface `System.Collections.Generic.IComparer<T>` dans une classe spécialisée. Si vous implémentez une de ces interfaces, il vaut mieux aussi surcharger les opérateurs « == », « != », « < », « > », « <= » et « >= » de façon à ce que les implémentations de ces surcharges appèlent l'implémentation que vous avez fourni pour l'interface.

## *Cloner des objets*

Nous avons souvent besoin d'obtenir une copie d'un objet existant, c'est-à-dire, de cloner un objet. Il est plus rigoureux de dire que l'on copie l'état de l'objet dans un autre objet. Pour certaines classes, cloner une instance n'a pas de sens. Par exemple il n'y a pas de sens à copier une instance de la classe `Thread`.

Pour les instances de types valeur, l'opérateur d'affectation « = » copie l'état de l'objet source dans l'objet destination. La copie est réalisée octet par octet. Par exemple, l'opération de boxing se sert de cette copie.

Pour les instances de types référence, l'opérateur d'affectation « = » copie la référence, et non l'objet. Il est donc nécessaire de prévoir une méthode pour copier l'état d'un objet de type référence. Vous pouvez implémenter la méthode `object Clone()` de l'interface `System.ICloneable`, prévue à cet effet. Cette interface ne présente que cette méthode. Voici un exemple qui illustre l'implémentation de cette interface :

*Exemple 10-5 :*

```
class Article {
   public string Description ;
   public int Prix ;
}
class Commande : System.ICloneable {
   public int Quantite ;
   public Article Article ;
   public override string ToString() {
      return "Commande : " + Quantite + " x " + Article.Description +
             "   Coût total : " + Article.Prix * Quantite ;
   }
   public object Clone() {
      Commande clone = new Commande();
```

```
      // Copie superficielle
      clone.Quantite = this.Quantite;
      clone.Article = this.Article;
      return clone;
   }
}
class Program {
   static void Main() {
      Commande commande = new Commande() ;
      commande.Quantite = 2 ;
      commande.Article = new Article() ;
      commande.Article.Description = "Chaussure" ;
      commande.Article.Prix = 80 ;
      System.Console.WriteLine(commande) ;
      Commande commandeClone = commande.Clone() as Commande ;
      commandeClone.Article.Description = "Veste" ;
      System.Console.WriteLine(commande) ;
   }
}
```

Cet exemple affiche ceci :

```
Commande : 2 x Chaussure   Coût total : 160
Commande : 2 x Veste   Coût total : 160
```

Clairement, la modification faite sur l'article de la commande clonée a été répercutée sur l'article de la commande originale. Cela est tout à fait normal puisque dans ce programme il n'existe qu'une seule instance de la classe `Article`. Elle est référencée par les deux instances de la classe `Commande`. On dit que la commande originale a subit une *copie superficielle* (*shallow copy* en anglais). La classe `Object` présente la méthode protégée `MemberwiseClone()` qui permet de réaliser une copie superficielle. Ainsi on peut réécrire l'exemple précédent comme ceci sans en modifier la sémantique :

*Exemple 10-6 :*

```
...
class Commande : System.ICloneable {
...
   public object Clone() {
      // Copie superficielle
      return this.MemberwiseClone();
   }
}
...
```

La notion de copie superficielle induit des bugs car en général il n'est pas considéré comme normal qu'une modification sur un graphe d'objets clonés soit répercutée sur le graphe original. On peut lui préférer la notion de *copie en profondeur* (*deep copy* en anglais) qui comme son nom l'indique, clone la totalité du graphe. Adaptons notre exemple à la copie en profondeur :

*Exemple 10-7 :*

```
...
class Article : System.ICloneable {
   ...
   public object Clone() {
      // Copie superficielle = Copie en profondeur.
      return this.MemberwiseClone() ;
   }
}
class Commande : System.ICloneable {
   ...
   public object Clone() {
      // Copie en profondeur.
      Commande clone = new Commande() ;
      clone.Quantite = this.Quantite ;
      clone.Article = this.Article.Clone() as Article;
      return clone ;
   }
}
...
```

L'affichage de ce programme est maintenant ceci :

```
Commande : 2 x Chaussure   Coût total : 160
Commande : 2 x Chaussure   Coût total : 160
```

Remarquez que dans l'exemple précédent nous précisons qu'en ce qui concerne la classe `Article` la copie superficielle est équivalente à la copie en profondeur. On pourrait être tenté d'affirmer que pour une classe donnée, la copie superficielle est équivalente à la copie en profondeur si et seulement si tous ses membres sont de type valeur. Or la classe `Article` admet un champ de type `string` qui est un type référence. Nous expliquons un peu plus loin dans ce chapitre (en page 370) que la classe `String` présente certaines propriétés dont l'immuabilité de ses instances, qui font que souvent, les instances de cette classe peuvent être considérées comme des instances d'un type valeur.

L'interface `ICloneable` est souvent critiquée principalement parce qu'elle ne permet pas à ses implémentations de communiquer clairement à leurs clients s'il s'agit d'une copie en profondeur ou d'une copie superficielle, voire d'une copie en profondeur incomplète. Les concepteurs du *framework* ont failli rendre cette interface obsolète lors du passage à .NET 2.0. La raison principale qui a « sauvée » cette interface de l'obsolescence est qu'elle est implémentée par de nombreuses classes standard ou propriétaires. Aussi, il reste déconseillé de l'implémenter à moins de fournir une documentation rigoureuse. Une alternative est par exemple un *constructeur de copie* qui prend en paramètre un booléen précisant qu'elle genre de copie l'on souhaite :

*Exemple 10-8 :*

```
...
class Commande {
   public int Quantite ;
   public Article Article ;
   public override string ToString() {
```

```
      return "Commande : " + Quantite + " x " + Article.Description +
             "   Coût total : " + Article.Prix * Quantite ;
   }
   // Constructeur par défaut.
   public Commande() { }
   // Constructeur de copie paramètrable.
   public Commande( Commande original , bool bDeepCopy) {
      this.Quantite = original.Quantite;
      if( bDeepCopy )
         this.Article = original.Article.Clone() as Article;
      else
         this.Article = original.Article;
   }
}
class Program {
   static void Main() {
      ...
      Commande commandeClone = new Commande( commande , true ) ;
      ...
   }
}
```

Un autre argument qui plaide en faveur de l'interface `ICloneable` est qu'elle permet d'implémenter naturellement le *design pattern prototype* qui permet de créer de nouveaux objets en clonant un objet prototype référencé par une référence de type `ICloneable`.

# *Boxing et UnBoxing*

C++ → C# Rien de semblable n'existe en C/C++. Tout ceci découle directement du fait que tous les types dérivent de la classe `Object`. Il n'existe pas de telle classe en C/C++.

C# Les instances de type valeur, locales à une méthode, sont directement stockées dans la pile du thread. Le thread n'a donc pas besoin de pointeurs ni de référence vers les instances de types valeur.

Beaucoup de méthodes ont besoin d'arguments sous forme d'une référence de type `Object`. Comme tous les types, les types valeur dérivent de la classe `Object`. Cependant les instances de types valeur n'ayant pas de références, il a fallu trouver une solution pour pouvoir obtenir une référence vers une instance d'un type valeur lorsque l'on en a besoin. Cette solution fait l'objet de la présente section : c'est le *boxing* (appelé *compartimentation* en français. Ce terme est peu usité, aussi nous conserverons le terme *boxing* dans cet ouvrage).

## *Opération de Boxing*

Voici un exemple concret pour exposer la problématique. La méthode `f()` accepte une référence de type `object`. *A priori*, on ne peut donc pas l'appeler avec un argument qui n'admet pas de référence, par exemple un entier de type `int` :

*Exemple 10-9 :*

```
class Program {
   static void f( object o ) { }
   public static void Main() {
      int i = 9 ;
      f( i ) ;
   }
}
```

Cependant le petit programme précédent compile et fonctionne. La magie du boxing a permis d'obtenir une référence vers une instance qui n'en avait pas ! L'opération de boxing s'effectue en interne en trois étapes :

- Une nouvelle instance du type valeur est créée et allouée sur le tas.
- Cette instance est initialisée avec l'état de l'instance allouée sur le tas. Dans le cas de notre entier, une copie de quatre octets est effectuée. On peut dire que notre instance initiale a été clonée.
- Une référence vers la nouvelle instance est utilisée à la place de l'instance allouée sur le tas.

Le code IL de cette méthode `Main()` est le suivant :

```
.locals init ([0] int32 i)
  IL_0000:  ldc.i4.s   9
  IL_0002:  stloc.0
  IL_0003:  ldloc.0
  IL_0004:  box        [mscorlib]System.Int32
  IL_0009:  call       void Program::f(object)
  IL_000e:  ret
```

On voit que l'instruction `box` du langage IL est prévue spécialement pour l'opération de boxing. Cette instruction place la référence vers la nouvelle instance sur le haut de la pile.

On profite de cet exemple pour souligner que le mot clé C# `int` est un alias vers le type `System.Int32`.

## *Une optimisation dangereuse de l'utilisation du boxing*

Si vous êtes concerné par les performances, il faut savoir que l'opération de boxing a un coût non nul et non négligeable. Le programme de l'Exemple 10-10 utilise deux fois le boxing alors que le programme de l'Exemple 10-11 n'utilise qu'une seule fois le boxing, ce qui le rend plus optimisé. Malgré les apparences, ces programmes n'ont pas le même comportement car le premier affiche « Références différentes » et le second affiche « Mêmes références ». Cette optimisation est donc assez dangereuse puisqu'elle modifie le comportement d'une manière non évidente. Nous vous conseillons de ne pas l'utiliser.

*Exemple 10-10 :*

```
class Program {
   static void f(object o1, object o2) {
      if (o1 == o2)
         System.Console.WriteLine("Mêmes références") ;
      else
         System.Console.WriteLine("Références différentes") ;
   }
   public static void Main() {
      int i = 9 ;
      f(i, i) ;
   }
}
```

*Exemple 10-11 :*

```
class Program {
   static void f(object o1, object o2) {
      if (o1 == o2)
         System.Console.WriteLine("Mêmes références") ;
      else
         System.Console.WriteLine("Références différentes") ;
   }
   public static void Main() {
      int i = 9 ;
      object o = i;
      f(o, o) ;
   }
}
```

## *Opération de UnBoxing*

L'opération inverse du boxing existe et s'appelle le *UnBoxing* (le terme *décompartimentation* existe en français mais nous continuerons à utiliser le terme unboxing). Voici un exemple où le unboxing est mis en œuvre :

*Exemple 10-12 :*

```
class Program {
   public static void Main() {
      int i = 9 ;
      object o = i;    // i est 'boxé'.
      int j = (int)o;  // o est 'unboxé'.
   }
}
```

Le code IL de cette méthode `Main()` est le suivant :

```
.locals init ([0] int32 i,
              [1] object o,
```

```
          [2] int32 j)
  IL_0000:  ldc.i4.s   9
  IL_0002:  stloc.0
  IL_0003:  ldloc.0
  IL_0004:  box        [mscorlib]System.Int32
  IL_0009:  stloc.1
  IL_000a:  ldloc.1
  IL_000b:  unbox      [mscorlib]System.Int32
  IL_0010:  ldind.i4
  IL_0011:  stloc.2
  IL_0012:  ret
```

On voit que l'instruction `unbox` du langage IL est prévue spécialement pour l'opération de unboxing. Sans entrer dans les détails internes, sachez que l'instruction `unbox` place un pointeur sur la pile vers l'objet boxé qui se trouve sur le tas. C'est l'instruction IL `ldind` qui charge sur la pile, la valeur de l'objet référence.

L'instance ne peut être unboxée vers le type spécifié si elle n'est pas exactement de ce type. L'exception `InvalidCastException` peut donc être lancée par une opération de unboxing. De plus si la référence à unboxer est nulle l'exception `NullReferenceException` est lancée.

En C#, les opérations de boxing et d'unboxing sont implicites. C'est-à-dire que c'est le compilateur qui va générer les appels aux instructions IL `box` et `unbox` lorsque c'est nécessaire. Il n'en va pas nécessairement de même pour tous les langages .NET.

# *Les types primitifs*

C++ → C# Les types présentés ici sont assez similaires aux types C++.

Le mot-clé `unsigned` n'existe pas en C#. On utilise les types `byte`, `ushort`, `uint` et `ulong` pour typer les entiers non signés.

La taille d'un `long`/`ulong` est de huit octets en C#.

En C#, le type `decimal` (sur 16 octets), représente les réels d'une manière exacte, dans la limite de 28 chiffres significatifs.

En C#, le type `bool` n'accepte que les valeurs `true` et `false`. Aucune conversion avec des types entiers n'est permise.

En C# le type `char` est maintenant codé sur deux octets et respecte la norme UNICODE. Rappelons qu'en C++ ce type est codé sur un octet et respecte la norme ASCII.

C# Le langage C# possède plusieurs types primitifs, qui sont tous de type valeur. Chaque type primitif de C# correspond exactement à un type du CTS. Les types primitifs sont définis par les mots-clés : `bool`, `byte`, `sbyte` `char`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`. Les correspondances entre ces mots-clés et les types du Framework .NET sont définies ci-après.

Chaque type primitif permet l'écriture de constantes. En effet le développeur a souvent besoin d'initialiser ses variables de type primitif avec des valeurs de ce type.

## *Les types concernant la représentation des nombres entiers*

### *Les types*

| mots-clés C# | Types du CTS correspondants | Taille en bits | Intervalle de valeur |
|---|---|---|---|
| byte | System.Byte | 8 | [ 0 ; 255 ] |
| sbyte | System.SByte | 8 | [ -128 ; 127 ] |
| short | System.Int16 | 16 | [ -32768 ; 32767 ] |
| ushort | System.UInt16 | 16 | [ 0 ; 65635 ] |
| int | System.Int32 | 32 | [-2,1x10^9 ; 2.1x10^9 ] |
| uint | System.UInt32 | 32 | [ 0 ; 4.2x10^9 ] |
| long | System.Int64 | 64 | [-9,2x10^18 ; 9,2x10^18] |
| ulong | System.UInt64 | 64 | [ 0 ; 1,8x10^19] |

Les types non signés ushort, uint et ulong ainsi que le type signé sbyte ne sont pas CLS compliant (c'est-à-dire qu'ils ne sont pas conformes au CLS défini page 129). Concrètement les autres langages utilisant .NET ne sont pas tenus de les implémenter. Il ne faut donc pas utiliser ces types dans les arguments de méthodes susceptibles d'être appelées par un autre langage.

### *Définition des constantes de ces types*

Par défaut une constante d'un de ces types s'écrit simplement en base 10 (décimal). Par exemple :

```
int i = 1024 ;
```

Cependant vous avez la possibilité décrire une telle constante en base 16 (hexadécimale) :

```
int i = 0X00000400 ;
```

Voici un petit problème :

```
int  i = 1000000000 ;  // i = 1 milliard
long j = 10 * i ;      // j n'est pas égal à 10 milliards
```

Quelle est la valeur de j ? Elle n'est pas de 10 milliards. En effet le calcul s'étant effectué avec des int, (limité à deux milliards et quelque en valeur absolue) le résultat est de type int. Ensuite le résultat est finalement copié dans une variable de type long. Le problème principal est que rien n'est signalé au développeur par le compilateur (sauf s'il utilise le mot-clé checked). Pour résoudre ces problèmes, on doit signaler au compilateur que l'on veut utiliser des valeurs sur huit octets en ajoutant L à la constante entière :

```
int i = 1000000000 ;  // i = 1 milliard
long j= 10L * i ;      // j égal à 10 milliards
```

## *Les types concernant la représentation des nombres réels*

| Mots-clés C# | Types du CTS correspondants | Taille en bits | Intervalle de valeur | Plus petite valeur possible | Précision |
|---|---|---|---|---|---|
| float | System.Single | 32 | [-3,4x10^38 ; 3,4x10^38 ] | 1,4x10^-45 | 7 décimales |
| double | System.Double | 64 | [-1,8x10^308 ; 1,8x10^308] | 4,9x10^-324 | 15 décimales |
| decimal | System.Decimal | 128 | [ -8x10^28 ; 8x10^28 ] | 1x10^-28 | 28 décimales |

### *Particularité des types float et double*

Le type `float` (respectivement `double`) est défini par la norme ANSI IEEE 754. Toutes les valeurs de 32 (respectivement 64) bits ne représentent pas forcément un réel correct. Ces valeurs non correctes sont représentées par le champ statique constant `System.Single.Nan` (respectivement `System.Double.Nan`). `Nan` s'interprète par « *Not A Number* ».

Le type `float` (respectivement `double`) admet aussi deux autres valeurs très pratiques représentant l'infini positif et l'infini négatif. L'idée est que les opérations et les fonctions usuelles des réels vers les réels se comportent logiquement aux infinis. Les cas indéterminés (comme zéro multiplié par l'infini) retourne la valeur `Nan`. Ces valeurs sont représentées par les champs statiques constants `PositiveInfinity` et `NegativeInfinity`

Les types `Single` et `Double` présentent les méthodes suivantes :

| Méthodes | Description |
|---|---|
| `public static float/double Parse(string s)` | Même fonctionnalité que pour les types entiers. |
| `public string ToString()` | Même fonctionnalité que pour les types entiers. |
| `public static bool IsInfinity(float/double d)` | Cette méthode statique retourne `true` si l'entrée est infinie dans le type primitif concerné. |
| `public static bool IsNan(float/double d)` | Cette méthode statique retourne `true` si l'entrée n'est pas valide dans le type primitif concerné. |

Les types `single` et `double` présentent aussi les champs statiques constants suivants :

| Champs | Description |
|---|---|
| `Epsilon` | Plus petite valeur positive non nulle représentable avec le type primitif concerné. |
| `MaxValue` | Plus grande valeur positive représentable avec le type primitif concerné (existe aussi pour le type `decimal`). |
| `MinValue` | Opposé de `MaxValue` (existe aussi pour le type `decimal`). |

### *Le type decimal*

Avec ses 28 décimales, le type décimal est très utile dans les applications financières ou chaque décimale compte. Il vaut mieux l'éviter pour des applications de calcul intensif qui ne nécessitent pas une telle précision car il est coûteux de manipuler des instances de `decimal`.

### *Définition de constantes*

Par défaut les constantes réelles sont de type `double`. Voici quelques exemples :

```
double d1 = 10 ;    // OK, d1 vaut dix.
double d2 = 10.0 ;  // OK, d2 vaut dix.
double d3 = 10E3 ;  // OK, notation scientifique d3 vaut dix milles.
double d4 = 10.1E3; // OK, notation scientifique d4 vaut dix milles cent.
double d4 = 1.1E-1; // OK, notation scientifique d5 vaut zéro virgule onze.
```

Pour que la constante soit convertie en `float`, il faut la suffixer avec `f` ou `F`.

```
float  d1 = 1.2; // KO, la constante est un double qui ne peut être
                 // convertie implicitement en float (perte d'information)
float  d2 = 1.2f ;       // OK
float  d3 = (float)1.2 ; // OK la conversion est explicite.
float  d4 = 12E-1F ;     // OK la conversion est explicite.
```

Pour que la constante soit convertie en `decimal`, il faut la suffixer avec `m` ou `M`.

```
decimal d1 = 1.2 ; // KO, la constante est un double qui ne peut être
                   // convertie implicitement en decimal.
decimal d2 = 1.2m ;         // OK
decimal d3 = (decimal)1.2 ; // OK la conversion est explicite.
decimal d4 = 12E-1m ;       // OK la conversion est explicite.
```

## *Le type booléen*

Le mot-clé C# pour ce type est `bool`. Une variable de ce type ne peut prendre que deux valeurs : `true` ou `false`. Le fait que la valeur `true` soit représentée par un 1 ou un 0 par le CLR à l'exécution ne concerne en aucun cas le développeur. Si vous avez un tableau de booléens, il est préférable d'utiliser un des types prévus à cet effet décrit en page 573, qui optimise le stockage des valeurs en mémoire en stockant 8 booléens dans un octet.

Le mot-clé `bool` est un alias pour le type du CTS `System.Boolean`. Cette structure présente les deux méthodes « `public static bool Parse(string s)` » et « `public string ToString()` » décrite un peu plus haut. Notons que `Sytem.Boolean` présente aussi les deux champs statiques constants de type `string`, `FalseString` (égal à `"False"`) et `TrueString` (égal à `"True"`). Ces deux chaînes de caractères sont les images (respectivement les antécédents) des valeurs `false` et `true` par la méthode `ToString()` (respectivement `Parse()`).

## *Le type représentant un caractère*

Le mot-clé C# pour ce type est `char`. Une instance de ce type représente un caractère respectant la norme UNICODE. Voici quelques exemples de constantes de type `char` :

```
char  c1 = 'A' ;      // Lettre A en UNICODE.
char  c2 = '\x41' ;   // Le 65ème caractère UNICODE 'A'(41 est en Hexa).
char  c3 = (char)65 ; // Le 65ème caractère UNICODE 'A'.
char  c4 = '\u0041' ; // Le 65ème caractère UNICODE 'A'(41 est en Hexa).
```

Le mot-clé `char` représente le type du CTS `System.Char`. Cette structure présente notamment des méthodes statiques pour déterminer si un caractère est une lettre majuscule, une lettre minuscule, un chiffre etc. En voici quelques-unes :

| **Méthodes statiques de la classe `System.Char`** | **Description** |
|---|---|
| `bool IsLower(char ch )` | Renvoie `true` si `ch` est une minuscule (les minuscules accentuées sont aussi considérées comme des minuscules). |
| `bool IsUpper(char ch )` | Renvoie `true` si `ch` est une majuscule. |
| `bool IsDigit(char ch )` | Renvoie `true` si `ch` est un chiffre. |
| `bool IsLetter (char ch )` | Renvoie `true` si `ch` est une lettre. |
| `bool IsLetterOrDigit(char ch)` | Renvoie `true` si `ch` est une lettre ou un chiffre. |
| `bool IsPunctuation(char ch )` | Renvoie `true` si `ch` est un caractère de ponctuation. |
| `bool IsWhiteSpace(char ch )` | Renvoie `true` si `ch` est un espace. |
| `char ToUpper(char ch )` | Renvoie la majuscule de la lettre `ch`. |
| `char ToLower(char ch )` | Renvoie la minuscule de la lettre `ch`. |

## *Conversions entre nombres entiers et chaînes de caractères*

Les types primitifs présentent chacun les trois méthodes suivantes :

- `public static [`*`le type primitif concerné`*`].Parse(string s)`
  Cette méthode statique parse la chaîne de caractères en entrée de façon à retourner la valeur dans le type primitif concerné. Deux exceptions peuvent être lancées :

  - `-FormatException` si la chaîne ne contient pas que des chiffres avec éventuellement un des caractères « + » ou « - » en première position.
  - `-OverflowException` si le nombre représenté n'entre pas dans l'intervalle de valeurs du type primitif concerné.

- `public bool [`*`le type primitif concerné`*`].TryParse(string s, out [`*`le type primitif concerné`*`])`
  Cette méthode statique parse la chaîne de caractères en entrée de façon à retourner la valeur au moyen du paramètre `out` dans le type primitif concerné. Cette méthode retourne un booléen `true` si elle a pu effectivement parser une valeur dans le type concerné, sinon elle retourne `false`.
  `public string [`*`le type primitif concerné`*`].ToString()`
  Cette méthode écrit la valeur entière dans une chaîne de caractères. Cette méthode est la réécriture de la méthode `ToString()` de la classe `Object`.

L'exemple suivant affiche deux fois la chaîne de caractères `"-234"` :

*Exemple 10-13 :*

```
class Program {
   static void Main() {
      string s1 = "-234" ;
      int i = System.Int32.Parse(s1);
      string s2 = i.ToString();
      System.Console.WriteLine(s1) ;
      System.Console.WriteLine(s2) ;
   }
}
```

# *Opérations sur les types primitifs*

C++ → C# En C#, le modulo s'applique aussi aux types réels. De plus le langage C# dispose du mot-clé `checked` qui permet de forcer la vérification de toutes les conversions et opérations. Une exception est lancée si un problème est rencontré.

## *Opérations arithmétiques sur un même type primitif*

C# Les cinq opérations arithmétiques sont :

| Opérateur | Opération |
|---|---|
| + | L'addition. |
| - | La soustraction. |
| * | La multiplication. |
| / | La division. |
| % | Le modulo (i.e le reste de la division) |

Ces opérations s'appliquent à tous les types primitifs, entiers ou réels. Chacune de ces opérations accepte deux opérandes. Une facilité d'écriture est proposée avec les cinq opérateurs suivants :

| Facilité d'écriture | Equivalence |
|---|---|
| `i += j;` | `i = i+j;` |
| `i -= j;` | `i = i-j;` |
| `i *= j;` | `i = i*j;` |
| `i /= j;` | `i = i/j;` |
| `i %= j;` | `i = i%j;` |

## *Gestion de la division par zéro*

En ce qui concerne la *division par zéro* sur un type entier et le type `decimal`, l'exception `DivideByZeroException` est lancée.

En ce qui concerne la division par zéro sur les types `double` et `float`, la variable prend la valeur particulière `NaN` (Not a Number) (et non pas une valeur infinie comme on pourrait s'y attendre).

## *Gestion des dépassements de capacité*

Un *dépassement de capacité* est le fait d'obtenir une valeur hors de l'intervalle de valeurs autorisées à la suite d'une opération. Un dépassement de capacité provoque les conséquences suivantes :

- Sur des variables de type `decimal` l'exception `OverflowException` est lancée.
- Sur les variables de type `double` et `float`, la variable prend la valeur particulière `NaN` (Not a Number).
- Sur des variables de types entiers les bits significatifs qui dépassent ne sont pas pris en compte. On obtient donc une valeur erronée. Pour pallier ce comportement par défaut pour le moins dangereux, vous pouvez utiliser le mot-clé `checked` :

  *Exemple 10-14 :*

```
class Program {
   static void Main() {
      byte i = 255 ;
      checked {
         i += 1 ; // Une exception de type
                  // System.OverflowException est lancée.
      }
   }
}
```

  Le mot-clé `checked` peut aussi s'appliquer directement à une expression.

*Exemple 10-15 :*

```
...
      byte i = 255 ;
      i = checked( (byte)(i+1) ) ; // Une exception de type
...                                // System.OverflowException est lancée.
```

Le mot-clé `checked` ne peut s'appliquer à une classe ou à une méthode. Dans un bloc de code vérifié par le mot-clé `checked`, vous pouvez utiliser le mot-clé `unchecked` pour ponctuellement revenir au comportement par défaut.

## *Priorité des opérateurs*

En ce qui concerne la *priorité des opérateurs* elle est définie comme suit :

- + - * / sont prioritaires sur %.
- + et — sont de priorité égale.
- * et / sont de priorité égale.
- * et / sont prioritaires sur + et -.
- Lorsqu'il y a même priorité l'opérateur le plus à gauche dans l'expression est prioritaire. Mais cela n'a pas d'importance puisque les opérations de base avec la même priorité sont commutatives.
- On peut toujours rendre une opération prioritaire en la mettant entre parenthèses.

Par exemple :

*Exemple 10-16 :*

```
...
      int a = 3 ;
      int b = 10 ;
      int c = 4 ;
      int r1 =  b+c %a ;    // r1 = 11
      int r2 = (b+c)%a ;    // r2 = 2
      int r3 = a*b+c ;      // r3 = 34
      int r4 = a*b%c ;      // r4 = 2 (30 modulo 4)
...
```

## *Opérateurs de pré et post incrémentation et décrémentation*

C++ → C# Une différence avec C++ est qu'en C#, ces opérations s'appliquent aussi aux types réels. De plus, dans le cas d'un type entier on peut gérer le dépassement de capacité avec le mot-clé `checked`.

C# On peut incrémenter ou décrémenter d'une unité, une variable de type primitif entier ou réel, au moyen des opérateurs ++ et `--`. Par exemple :

- `i++;` équivaut à `i = i+1;`
- `i--;` équivaut à `i = i-1;`

Toute la difficulté vient du fait que ces deux opérateurs peuvent se placer soit avant soit après la variable, ce qui change l'ordre d'évaluation des expressions. Par exemple :

*Exemple 10-17 :*

```
...
      int i = 3 ;
      int j = i++ ; // j vaut 3 et i vaut 4
      int k = 3 ;
      int l = ++k ; // l vaut 4 et i vaut 4
...
```

Lorsque l'on utilise ces opérateurs avec des types entiers, il se peut que l'on provoque un dépassement de capacité. Comme pour tout dépassement de capacité sur des variables de type entier une exception n'est pas lancée...

*Exemple 10-18 :*

```
...
      byte i = 255 ;
      i++ ;  // i vaut 0
...
```

...à moins que l'on utilise le mot-clé `checked` :

*Exemple 10-19 :*

```
...
      byte i = 255 ;
      checked{ i++ ; } // Une exception de type
...                    // System.OverflowException est lancée.
```

## *Opérations arithmétiques entre types primitifs différents*

C++ → C# Les types primitifs invoqués étant différents, les comportements des opérations arithmétiques entre types primitifs différents ne sont pas les mêmes entre C++ et C#.

C# Lorsqu'une opération arithmétique s'effectue avec deux opérandes de types primitifs différents, le résultat a un type dont l'intervalle de valeur est au moins égal ou plus grand que ceux des types des opérandes. Certains mariages de type sont interdits par le compilateur. Voici le résultat de mes propres essais, exposés dans un tableau :

| Un des deux types | L'autre des deux types | Type retourné |
|---|---|---|
| ***Opérations entre types primitifs entiers*** | | |
| `sbyte byte short ushort` | `sbyte byte short ushort` | `int` |
| `sbyte byte short ushort` | `int` | `int` |
| `sbyte byte short ushort int long` | `uint` | `long` |

| | | |
|---|---|---|
| int long uint | int | long |
| ulong | int | **ERREUR de compilation** |
| long | long | long |
| ulong | ulong | ulong |
| ulong | long | **ERREUR de compilation** |
| ***Opérations entre types primitifs entiers et réels*** | | |
| sbyte byte short ushort int uint long ulong | float | float |
| sbyte byte short ushort int uint long ulong | double | double |
| sbyte byte short ushort int uint long ulong | decimal | decimal |
| ***Opérations entre types primitifs réels*** | | |
| float double | decimal | **ERREUR de compilation** |
| double | float | double |
| float | float | float |
| double | double | double |
| decimal | decimal | decimal |

De plus les transtypages (*cast* en anglais) sont autorisés de n'importe quel type entier ou réel vers n'importe quel type entier ou réel. Bien entendu ceci reste dangereux puisque les intervalles de valeurs de tous ces types sont tous différents. Le développeur peut toujours utiliser le mot-clé `checked`. Ainsi si une valeur doit être convertie dans un type où elle n'est pas représentée, une exception de type `System.OverflowException` sera lancée.

*Exemple 10-20 :*

```
...
      int i = -5 ;
      checked {
         byte b = (byte)i ; // Exception lancée. Le type byte ne
      }                     // représente pas de nombres négatifs.
...
```

Notez cependant, qu'il est possible d'arrondir un type réel en un type entier. Par exemple, on peut convertir le `double` 3.1415 en un `byte` de 3. C'est la valeur entière se trouvant avant la virgule lors de l'écriture du réel qui est alors choisie.

Lorsque plusieurs opérateurs sont utilisés dans une expression, ils sont exécutés selon leur ordre de priorité (décrit un peu plus haut). Si les types primitifs des variables sont différents, les résultats intermédiaires suivent les règles énoncées.

## *Opérations binaires (bit à bit)*

C++ → C# Rien de nouveau par rapport au C/C++ si ce n'est que tout ceci est disponible en 64 bits grâce aux types `long` et `ulong`.

C# Des opérations binaires (i.e bit à bit) peuvent être effectuées entre opérandes de type `int`, `long` , `uint`, `ulong` . Dans le cas d'une opération binaire, les variables de type `sbyte`, `short`, `byte`, `ushort` sont converties en type `int` :

| Opération | Aussi nommée | Opérateur |
|---|---|---|
| AND | « Et » logique | & |
| OR | « Ou » logique | \| |
| XOR | « Ou exclusif » logique | ^ |
| NOT | Inversion des bits | |

Seul l'opérateur «  » agit sur un seul opérande, contrairement aux autres qui agissent sur deux opérandes. Rappelons ces opérations :

| Bit A | Bit B | A AND B | A OR B | A XOR B |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

Des opérations de décalages peuvent être effectuées sur des variables de type `int`, `long`, `uint` et `ulong` grâce aux opérateurs :

- << décalage à gauche.
- >> décalage à droite.

Rappelons qu'un décalage à gauche d'une position en notation décimale provoque une multiplication par 10. De même, un décalage à gauche d'une position en notation binaire provoque

une multiplication par deux. Un décalage à droite d'une position en notation binaire provoque une division entière par deux.

*Exemple 10-21 :*

```
...
      uint a = 11 ;
      uint b = a << 2 ; // a est inchangé, b vaut 44
      uint c = a >> 2 ; // a est inchangé, c vaut 2
...
```

Précisons que :

- Si le décalage se fait sur une variable d'un type signé (i.e `int` ou `long`) le décalage est arithmétique, c'est-à-dire que le bit de signe est inchangé et n'est pas décalé.
- Si le décalage se fait sur une variable d'un type non signé (i.e `uint` ou `ulong`) le décalage est binaire, et le résultat reste positif.

# *Les structures*

C++ → C# En C++, les structures et les classes sont des notions similaires. En C# la différence de base est que les structures définissent des types valeur et les classes définissent des types référence.

En C# toutes les structures dérivent de la classe `Object`.

Contrairement au C++, les champs d'une structure sont privés par défaut. De plus tous les champs que l'on veut déclarer publics doivent être précédés du mot-clé `public`.

Contrairement au C++, les structures ne peuvent dériver d'aucune classe ou structure, et ne peuvent servir de base pour aucune classe ou structure dérivée.

En C#, une structure peut avoir un ou plusieurs constructeurs, mais le développeur n'a pas le droit de définir un constructeur par défaut (i.e sans argument). De plus la syntaxe d'initialisation des champs dans le constructeur ( : `nom_dechamp(valeur)`...) n'existe plus. Enfin le constructeur par défaut initialise à 0 les champs.

Une structure peut avoir des méthodes publiques et privées, mais les méthodes doivent être complètement définies à l'intérieur de la déclaration de la structure (idem pour les classes). Il n'y a donc plus d'opérateur de résolution de portée « :: ».

En C#, la possibilité de mettre des champs de bits dans une structure n'existe plus.

En C#, il n'est pas obligatoire de mettre un « ; » à la fin de la déclaration d'une structure (idem pour les classes).

Enfin, mais ceci est une différence générale, en C#, au sein d'un même espace de nom, il n'est pas nécessaire d'avoir déjà défini une structure pour l'utiliser.

C# En C#, une structure est déclarée avec le mot-clé `struct`. La notion de structure est assez proche de celle d'une classe. Les points communs sont :

- Une structure peut avoir des champs, des propriétés, des méthodes, des délégués, des événements et des types. Si l'un de ces membres n'est pas déclaré avec le mot-clé `public`, il n'est pas accessible hors de la classe.

```
...
struct Employee{
   int salaire ;          // Ce champ est privé.
   public string nom ;    // Ce champ est public.
}
...
```

- Les membres d'une structure sont accessibles hors de la définition d'une structure avec l'opérateur point « . » .

```
...
  Employee raymond ;
  raymond.nom = "Raymond" ;  // OK Le champ nom est public.
  raymond.salaire = 3000 ;   // KO Le champ salaire est privé.
...
```

- Toute structure dérive de la classe `Object`.
- Une structure peut être définie :
  - À l'intérieur d'une autre structure. Elle peut cependant être instanciée hors de la structure encapsulante, si son niveau de visibilité le permet.
  - À l'intérieur d'une classe. Elle peut cependant être instanciée hors de la classe encapsulante, si son niveau de visibilité le permet.
  - À l'extérieur de toute classe et de toute structure. On peut donc instancier cette structure partout dans l'espace de noms courant, et dans les zones du code utilisant cet espace de nom.

  Une structure ne peut être définie à l'intérieur d'une méthode.
- Une structure peut avoir un ou plusieurs constructeurs mais le développeur n'a pas le droit de définir un constructeur par défaut (i.e sans arguments). De plus le compilateur s'attend à ce que tous les champs de la structure soient initialisés dans tous les constructeurs.

```
...
struct Employee {
   int salaire ;
   public string nom ;
   Employee (int _salaire,string _nom){
      salaire = _salaire ;
      nom = _nom ;
   }
}
...
```

  Le constructeur par défaut initialise à 0 les champs de type valeur et à nul les champs de type référence.
- Une structure peut avoir des méthodes autres que les constructeurs :

```
...
struct Employee {
   int salaire ;
```

```
    public string nom ;
    public int GetSalaire(){return salaire;} // Cette méthode est publique.
}
...
```

Cependant les structures et les classes présentent des différences importantes :

- Une structure est un type valeur alors qu'une classe est un type référence, avec toutes les différences que cela induit.
- Les structures ne peuvent dériver d'aucune classe ou structure, et ne peuvent servir de base pour aucune classe ou structure dérivée, bien que toutes les structures dérivent implicitement de la classe `Object`.
- Contrairement aux champs d'une classe, les champs d'une structure ne peuvent être explicitement initialisés dans la déclaration même du champ.
- Le développeur ne peut surcharger le constructeur par défaut (i.e sans argument).
- Les instances des structures étant souvent stockées sur la pile, il vaut mieux qu'elles ne soient pas trop volumineuses. Dans le cas de très grosses structures, il vaut mieux opter pour des classes.

# *Les énumérations*

C++ → C# Les énumérations du C# restent assez proches de celles du C++. Cependant quelques différences sont à remarquer.

La classe `System.Enum` présente des fonctions d'aide à la manipulation des énumérations.

Les énumérations en C# sont idéales pour remplacer le mécanisme de drapeaux binaires en C++. Rappelons qu'en C++ le mécanisme de drapeaux binaires utilise les macros. Une telle approche empêche le bénéfice de la vérification du type par le compilateur.

C# Un type énumération définit un ensemble de valeurs. En C#, un tel type est défini avec le mot-clé `enum`. Les variables d'un type énumération prennent leurs valeurs dans cet ensemble. Par exemple :

*Exemple 10-22 :*

```
class Program {
  enum Marque { Renault, Peugeot, Lancia }
  static void Main() {
    Marque marque = Marque.Renault ;
    switch (marque){
      case Marque.Renault: System.Console.WriteLine("Renault") ; break ;
      case Marque.Peugeot: System.Console.WriteLine("Peugeot") ; break ;
      case Marque.Lancia:  System.Console.WriteLine("Lancia") ;  break ;
    }
  }
}
```

Une énumération peut être définie :

- À l'intérieur d'une structure ou d'une classe. Elle peut cependant être instanciée hors de la structure ou de la classe encapsulante, si son niveau de visibilité le permet.
- À l'extérieur de toute classe et de toute structure. On peut donc instancier cette énumération partout dans l'espace de noms courant, et dans les zones du code utilisant cet espace de noms.

Une énumération ne peut être définie à l'intérieur d'une méthode.

## *Les énumérations et les types entiers*

Le compilateur considère que la valeur d'une variable énumération est un entier de type `int`. Par conséquent chaque valeur possible de l'ensemble des valeurs de l'énumération est associée à une valeur entière. On peut alors :

- Transtyper (caster) explicitement une variable d'un type énumération en un entier :

```
...
int i = (int) marque ;
...
```

- Incrémenter ou décrémenter une variable d'un type énumération :

```
...
Marque marque  = Marque.Renault ;
Marque marque2 = marque++ ;
marque2        += 2 ;
...
```

- Associer nos propres valeurs entières dans l'ensemble des valeurs de l'énumération :

```
...
enum Marque{ Renault = 100 , Peugeot= Renault+1 , Lancia = 200 }
...
```

  Par défaut la première valeur est 0, puis il y a un incrément de 1 pour les suivantes. Il vaut mieux définir la valeur 0 comme la plus courante, car les constructeurs initialisent à 0 par défaut les champs de type énumération.
- Choisir n'importe quel autre type entier que `int` pour définir les valeurs du type énumération :

```
...
enum MarqueFR : byte { Renault , Peugeot } ;
enum MarqueIT : long { Fiat  ,   Ferrari } ;
...
```

Toute énumération dérive de la classe `Object`. Par conséquent les méthodes de cette classe sont accessibles sur une énumération. Notons que la méthode `Object.ToString()` est redéfinie pour chaque énumération, de façon que la chaîne de caractères retournée corresponde à la chaîne de caractères désignant l'état courant dans le code. Par exemple :

*Exemple 10-23 :*

```
class Program {
   enum Marque { Renault, Peugeot, Lancia }
   static void Main() {
      Marque  marque = Marque.Renault ;
      string s = marque.ToString();
      System.Console.WriteLine(s) ;
   }
}
```

Ce programme affiche :

```
Renault
```

## *La classe System.Enum*

La classe `System.Enum` dérive de la classe `System.ValueType`. Elle présente des méthodes statiques d'aide à la manipulation d'un type énumération. Parmi ces méthodes remarquons :

- `static string Format(Type type, object value, string format)`
  Cette méthode permet de récupérer une chaîne de caractères correspondant à la valeur de l'énumération : l'argument `format` doit être égal à `"G"` ou `"g"` pour renvoyer le nom (par exemple `"Renault"`). Il doit être `"D"` ou `"d"` pour renvoyer la valeur correspondante (par exemple `"100"`) .

  *Exemple 10-24 :*

  ```
  ...
        Marque  marque = Marque.Renault ;
        string s = System.Enum.Format( typeof( Marque ) , marque , "G" ) ;
        // Ici la chaîne de caractères s contient "Renault".
  ...
  ```

- `static object Parse(Type type, string value, bool ingnoreCase)`
  Convertit une chaîne de caractères en une des valeurs de l'énumération. Si l'argument `ignoreCase` vaut `true`, cette méthode ne tient pas compte des majuscules/minuscules :

  *Exemple 10-25 :*

  ```
  ...
        Marque marque = (Marque)
                        System.Enum.Parse(typeof(Marque),"ReNaUlt",true) ;
        // Ici marque prend la valeur Marque.Renault.
  ...
  ```

  Si la chaîne de caractères ne correspond à aucune valeur, l'exception `ArgumentException` est générée.

- `string [] GetNames(Type type)`
  Renvoie les différents noms de l'énumération dans un tableau de chaîne de caractères. Par exemple :

*Exemple 10-26 :*

```
...
      foreach( string s in System.Enum.GetNames( typeof(Marque) ) )
         System.Console.WriteLine(s) ;
...
```

- `object[] GetValues(Type type)`
  Renvoie les différentes valeurs de l'énumération dans un tableau d'objets. Par exemple (soyez conscient qu'ici la référence o référence tour à tour chaque valeur de l'énumération boxée) :

  *Exemple 10-27 :*

```
...
      foreach( object o in System.Enum.GetValues( typeof(Marque) ) )
         System.Console.WriteLine( o.ToString() ) ;
...
```

## *Indicateurs binaires*

On peut faire en sorte qu'une instance d'une énumération puisse contenir plusieurs valeurs de l'ensemble des valeurs de l'énumération. Cette notion est connue sous le nom d'*indicateurs binaires*. Les indicateurs binaires sont notamment utilisés pour manipuler les *drapeaux* (*flag* en anglais) pour stocker plusieurs informations binaires dans une même variable. Par exemple :

*Exemple 10-28 :*

```
// Dans cet exemple, pas besoin de plus qu'un octet.
[System.Flags()]
enum Drapeaux : byte {
   Drapeau1 = 0x01, // Le bit 1 est à 1, les autres à 0.
   Drapeau2 = 0x04, // Le bit 3 est à 1, les autres à 0.
   Drapeau3 = 0x10  // Le bit 5 est à 1, les autres à 0.
}
class Program {
   public static void Main() {
      // En binaire, Drap vaut 10001000.
      Drapeaux drapeau = Drapeaux.Drapeau1 | Drapeaux.Drapeau3 ;
      // (Si le bit 1 est positionné) équivalent à
      // (Si Drapeau1 positionné).
      if ((drapeau & Drapeaux.Drapeau1) > 0) {/* */}
      // (Si les bits 3 et 5 sont positionnés) équivalent à
      // (Si Drapeau2 et Drapeau3 positionnés).
      if ( (drapeau & Drapeaux.Drapeau2) > 0 &&
           (drapeau & Drapeaux.Drapeau3) > 0 )
      { /* */ }
      System.Console.WriteLine( drapeau.ToString() ) ;
   }
}
```

Remarquez que les indicateurs binaires doivent être marqués avec l'attribut `System.Flags`. Cet attribut permet de préciser au CLR et aux clients qu'il s'agit d'un type d'indicateur binaire et non pas d'une énumération classique. Notamment, la présence de cet attribut affecte le résultat de la méthode `ToString()` et cet exemple affiche ceci :

```
Drapeau1, Drapeau3
```

Il afficherait ceci si l'attribut `Flags` était absent :

```
17
```

Un exemple d'indicateur binaire présenté par le *framework* .NET est l'énumération `System.Threading.ThreadState` décrite page 143.

# *Les chaînes de caractères*

## *La classe System.String*

C++ → C# Le *framework* .NET introduit la classe `System.String` pour stocker des chaînes de caractères et cela rompt complètement avec les traditionnels (et dangereux) `char*` et `char[]` (adieu les `strcpy()` et autres fonctions dangereuses). La philosophie d'utilisation de la classe `String` est proche de celle du type `string` fourni par la STL du C++.

Une grosse différence avec le type de la STL est qu'en C#, une instance de la classe `String` est immuable, elle ne peut être modifiée. Elle garde donc constamment la valeur qui lui a été donnée par son constructeur.

Une autre grosse différence est que la convention C/C++ de terminer une chaîne de caractères par le caractère nul n'existe plus en C#.

C# Le mot-clé C# `string` est un alias vers la classe `System.String`. Une instance de cette classe représente une chaîne de caractères au format UNICODE.

La classe `String` est déclarée comme `sealed`, c'est-à-dire qu'aucune classe ne peut en dériver. De plus les instances de la classe `String` se comparent selon l'équivalence.

Les instances de la classe `String` sont *immuables* (*immutable* en anglais). C'est-à-dire qu'une chaîne de caractères est initialisée par un constructeur, mais elle n'est jamais modifiée. Les nombreuses méthodes de modification de chaînes de caractères retournent toujours une nouvelle instance de la classe `String` contenant la chaîne modifiée. Ce comportement nuit aux performances, aussi vous pouvez utiliser la classe `System.Text.StringBuilder` qui permet de manipuler directement des chaînes de caractères. Cette classe est présentée un peu plus loin.

Les trois caractéristiques précédentes (`sealed`, comparaison selon l'équivalence et immuabilité des instances) font qu'**en pratique les chaînes de caractères se manipulent quasiment comme des instances de type valeur. Cependant gardez bien à l'esprit que le type `String` est une classe et est donc de type référence**. Nous vous conseillons de revenir au début du présent chapitre pour être bien conscient de tout ce que cela implique (manipulation seulement par des références, stockage des instances de `String` sur le tas, prise en compte par le ramasse-miettes etc).

## *Chaînes de caractères constantes régulières*

C++ → C♯ Contrairement à C++, C♯ propose deux sortes de chaînes de caractères constantes :

- Les chaînes de caractères constantes régulières, similaires aux chaînes de caractères constantes de C++.
- Les chaînes de caractères constantes verbatim, décrites dans la section suivante.

C♯ Voici un exemple de constantes de type `string` :

```
string  s = "ABCDEFG" ;
```

On peut introduire des caractères spéciaux dans une constante chaîne de caractères. Comprenez bien que ces caractères sont interprétés par le compilateur C♯. En voici la liste :

| Terme français | Terme anglais | Représentation | Valeur |
|---|---|---|---|
| Nouvelle ligne | New line | `\n` | `0x000A` |
| Tabulation horizontale | Horizontal tab | `\t` | `0x0009` |
| Tabulation verticale | Vertical tab | `\v` | `0x000B` |
| Retour arrière | Backspace | `\b` | `0x0008` |
| Retour chariot | Carriage return | `\r` | `0x000D` |
| Saut de page | Form feed | `\f` | `0x000C` |
| `Antislash` | Backslash | `\\` | `0x005C` |
| Apostrophe simple | Single quote | `\'` | `0x0027` |
| Apostrophe doubles | Double quote | `\"` | `0x0022` |
| Caractère nul | Null | `\0` | `0x0000` |

Par exemple le programme suivant affiche `"hel\nlo"` :

*Exemple 10-29 :*

```
class Program {
   public static void Main() {
      string s = "hel\\nlo" ;
      System.Console.WriteLine(s) ;
   }
}
```

## Chaînes de caractères constantes verbatim

Le langage C# autorise la définition d'une chaîne de caractères constante *verbatim* en ajoutant le caractère « @ » juste devant la double apostrophe marquant le départ de la chaîne. Une chaîne de caractères constante verbatim a les particularités suivantes :

- Elle accepte tous les caractères tels qu'ils sont, y compris le caractère antislash « \ » mais mis à part le caractère double apostrophes « " ». En effet, ce dernier définit la fin de la chaîne.
- Elle accepte et prend en compte le passage à la ligne, dans l'écriture de la chaîne de caractères. Cette fonctionnalité est particulièrement utile lorsque l'on génère du code.

Voici un exemple :

*Exemple 10-30 :*

```
class Program {
   public static void Main() {
      string sReguliere ="\\bonjour\n\\comment ca va\n\\ce matin ?" ;
      string sVerbatim = @"\bonjour
\comment ca va
\ce matin ?" ;
      System.Console.WriteLine("Chaîne de caractères régulière :") ;
      System.Console.WriteLine(sReguliere) ;
      System.Console.WriteLine("Chaîne de caractères verbatim :") ;
      System.Console.WriteLine(sVerbatim) ;
   }
}
```

Voici l'affichage obtenu :

```
Chaîne de caractères régulière :
\bonjour
\comment ca va
\ce matin ?
Chaîne de caractères verbatim :
\bonjour
\comment ca va
\ce matin ?
```

## Manipulation de chaînes de caractères

La classe `String` présente plusieurs méthodes pour la manipulation de chaînes de caractères. Par exemple on peut obtenir la taille d'une chaîne, concaténer deux chaînes, rechercher une chaîne à l'intérieur d'une autre chaîne etc.

| Les méthodes (toutes publiques) | Description |
|---|---|
| `int Length()` | Renvoie le nombre de caractères. |
| `static int Compare( string s1,string s2)` | Renvoie 0 si `s1` est égal, caractère par caractère à `s2`. Sinon calcule un poids sur chaque chaîne et renvoie la différence `Poid(s2)-Poid(s1)`. Le poids est calculé selon l'ordre des caractères dans la norme UNICODE. |
| `int CompareTo(string s)` | Semblable à `Compare()` mais cette méthode est non statique. Retourne `Poids(de l'instance)-Poids(de l'argument)`. |
| `static string Concat( string s1,string s2)` | Retourne une nouvelle chaîne qui est la concaténation de `s1` puis `s2`. Notez que la notation `s1 + s2` est équivalente. |
| `static string Copy(string s)` | Retourne une nouvelle chaîne qui est égale à `s`. |
| `bool EndWith(string s)` | Retourne `true` si la chaîne représentée par l'instance courante se termine par la chaîne `s`. |
| `static string Format( string s, de un à quatre arguments)` | Met en forme une chaîne de caractères à partir d'au plus quatre arguments. Voir la section suivante pour la mise en forme. |
| `int IndexOf( char c )` | Renvoie l'index du caractère `c` dans la chaîne représentée par l'instance courante. Si le caractère est en première position l'index vaut 0. Si le caractère est absent de la chaîne cette méthode retourne -1. La méthode `LastIndexOf()` est équivalente, mis à part qu'elle balaye la chaîne de droite à gauche. |
| `int IndexOf( char c ,int pos )` | Idem, mais la recherche commence à partir du `pos`-ième caractère. |
| `int IndexOf( string s )` | Idem, mais on recherche une autre chaîne au lieu d'un caractère. |
| `int IndexOf( string s , int pos )` | Idem, mais la recherche commence à partir du `pos`-ième caractère. |
| `string Insert( int pos, string s )` | Retourne une nouvelle chaîne qui est la chaîne représentée par l'instance courante dans laquelle on a inséré la chaîne `s` à partir du `pos`-ième caractère. |
| `string PadLeft( int n )` | Retourne une nouvelle chaîne qui est la chaîne représentée par l'instance courante dans laquelle on a inséré `n` espaces au début. |

| | |
|---|---|
| `string PadLeft( int n , char c )` | Idem, mais le caractère de remplissage est `c`. |
| `string PadRight( int n )` | Idem, mais le remplissage se fait à droite, avec des espaces. |
| `string PadRight( int n , char c )` | Même chose mais le remplissage se fait à droite, avec le caractère `c`. |
| `string Remove( int pos, int n )` | Retourne une nouvelle chaîne qui est la chaîne représentée par l'instance courante dans laquelle on a enlevé `n` caractères à partir du `pos`-ième caractère. |
| `string Replace(string oldstr, string newstr)` | Retourne une nouvelle chaîne qui est la chaîne représentée par l'instance courante sur laquelle on a remplacé chacune des sous chaînes égale à `oldstring` par `newstring`. |
| `string Replace( char oldchar, char newchar )` | Retourne une nouvelle chaîne qui est la chaîne représentée par l'instance courante dans laquelle on a remplacé chaque caractère `oldchar` par `newchar`. |
| `string Split( char[] )` | Retourne un tableau de nouvelles chaînes obtenues à partir de la chaîne représentée par l'instance courante. Un tableau de caractères séparateurs est passé en argument. |
| `bool StartWith( string s )` | Retourne `true` si la chaîne représentée par l'instance courante commence par la chaîne `s`. |
| `string SubString(int pos, int n)` | Retourne une nouvelle chaîne égale à l'extraction de la sous chaîne à la `pos`-ième position de longueur `n` caractères. |
| `string ToLower()` | Retourne une nouvelle chaîne qui est la chaîne représentée par l'instance courante avec toutes les majuscules transformées en minuscules. |
| `string ToUpper()` | Retourne une nouvelle chaîne qui est la chaîne représentée par l'instance courante avec toutes les minuscules transformées en majuscules. |
| `string Trim()` | Retourne une nouvelle chaîne qui est la chaîne représentée par l'instance courante avec les espaces, en début et en fin de chaîne, supprimés. `TrimStart()` (respectivement `TrimEnd()`) est identique mais ne traite que le début (respectivement la fin). |
| `string Trim( char[] )` | Retourne une nouvelle chaîne qui est la chaîne représentée par l'instance courante, avec les caractères spécifiés dans le tableau en argument en début et en fin de chaîne supprimée. `TrimStart(char[])` (respectivement `TrimEnd(char[])`) est identique mais ne traite que le début (respectivement la fin). |

## *Mise en forme d'une chaîne de caractères*

C++ → C# En C/C++ la mise en forme des chaînes de caractères utilise des indicateurs (comme "%d") au sein de la chaîne de caractères fournie à `printf()`. Ces indicateurs contiennent une information sur le type de la variable à afficher.

En C# la mise en forme passe toujours par des indicateurs, mais ils sont différents et ne contiennent plus aucune information de type. En revanche ils contiennent l'information de position dans la liste des arguments de la méthode `Format()`.

C# Cette section détaille l'action de la méthode statique `Format()` de la classe `string` :

```
static string Format(string s, de un à quatre arguments) ;
```

En effet cette méthode est très utilisée et présente un très grand nombre d'options. Le tableau ci-dessous illustre plusieurs exemples de mise en forme avec les variables suivantes :

```
int i = 123 ;
double d = 3.1415 ;
string sOut = System.String.Format( Signature de Format() ) ;
```

| **Signature de Format()** | **Valeur de sOut** | **Remarque** |
|---|---|---|
| `"abcd"` | `"abcd"` | - |
| `"ab{0}cd",i` | `"ab123cd"` | - |
| `"ab{0}cd",d` | `"ab3,1415cd"` | - |
| `"ab{0}cd",i,d` | `"ab123cd"` | L'argument `d` n'est pas utilisé, mais aucune erreur n'est déclarée ni provoquée. |
| `"ab{0}cd{1}ef",i,d` | `"ab123cd3.1415ef"` | Utilisation de deux arguments. |
| `"ab{1}cd{0}ef",i,d` | `"ab3.1415cd123ef"` | Autre utilisation de deux arguments. |
| `"ab{0}{0}cd ",i` | `"ab123123cd"` | Double utilisation du même argument |
| `"ab{0}cd{1}ef",i` | `ERREUR à l'exécution` | Un deuxième argument est requis. |
| `"ab{0,6}cd",i` | `"ab 123cd"` | `i` est représenté par six caractères avec cadrage à droite. Trois caractères d'espace sont ajoutés. Si le nombre de caractères était inférieur au nombre de caractères requis pour afficher `i`, `i` aurait été affiché dans sa totalité quand même. |
| `"ab{0,-6}cd",i` | `"ab123 cd"` | Même chose mais le cadrage se fait à gauche. |
| `"ab{0:0000}cd",i` | `"ab0123cd"` | Affichage d'au moins quatre chiffres, quitte à mettre des zéros en tête. |

| | | |
|---|---|---|
| `"ab{0,6:0000}cd",i` | `"ab 0123cd"` | Affichage d'au moins quatre chiffres quitte à mettre des zéros en tête, représenté sur 6 caractères, avec cadrage à droite. |
| `"ab{0:####}cd",i` | `"ab123cd"` | Un caractère # à remplacer n'est pas utilisé. |
| `"ab{0:##}cd",i` | `"ab123cd"` | Le format n'est pas respecté. |
| `"ab{0:##.##}cd",d` | `"ab3,14cd"` | Au plus deux décimales sont affichées. |
| `"ab{0:##.#}cd",3.14` | `"ab3,1cd"` | Arrondi inférieur. |
| `"ab{0:##.#}cd",3.18` | `"ab3,2cd"` | Arrondi supérieur. |
| `"ab{0:##%}cd",0.143` | `"ab14%cd"` | 0.143 est multiplié par 10 puis arrondi inférieur. 0.147 aurait produit 15%. |
| `"ab{0:E}cd",d` | `"ab3,141500E+000cd"` | Représentation scientifique. |
| `"ab{0:X}cd",i` | `"ab7Bcd"` | Représentation hexadécimale avec des lettres majuscules. |
| `"ab{0:x}cd",I` | `"ab7bcd"` | Représentation hexadécimale avec des lettres minuscules. |

## *La classe System.Text.StringBuilder*

Comme nous l'avons vu au début de la présente section, les instances de la classe `String` sont immuables. Ce fait nuit aux performances, car à chaque modification d'une chaîne de caractères il faut allouer une nouvelle instance de `String`. Ceci est d'autant plus coûteux si vous manipulez des chaînes de caractères longues.

Si une de vos applications modifie souvent des chaînes de caractères ou manipule des chaînes de caractères volumineuses, il est plus efficace d'utiliser la classe `System.Text.StringBuilder`.

En général une instance de `StringBuilder` se construit à partir d'une instance de la classe `String` avec le constructeur `StringBuilder(string)`.

Pour bien comprendre la classe `StringBuilder`, il faut assimiler la notion de *capacité* d'une instance de `StringBuilder`. Pour cela nous présentons les propriétés suivantes de `StringBuilder` (qui sont toutes publiques et non statiques) :

- `int Length{get;}`
  Cette propriété donne le nombre de caractères de la chaîne.
- `int Capacity{get;set;}`
  Cette propriété représente le nombre de caractères alloués physiquement pour stocker la chaîne de caractères. Ce nombre est la capacité. La valeur de ce champ est toujours supérieure ou égale à la valeur du champ `Length`. Il est particulièrement utile de positionner ce champ à une valeur supérieure à la taille réelle de la chaîne. Vous pouvez ainsi prévenir des opérations d'allocation mémoire du ramasse-miettes (parfois coûteuses), en vous créant une marge pour manipuler la chaîne.

Notez que la valeur de ce champ peut être automatiquement incrémentée lorsqu'une manipulation sur la chaîne de caractères produit un résultat d'une taille plus grande que la capacité. La valeur de cette incrémentation dépend de l'implémentation du *framework*. Dans cette situation, l'implémentation courante de *Microsoft* double la capacité.

Si vous positionnez ce champ à une valeur inférieure à la valeur du champ `Length` l'exception `ArgumentOutOfRangeException` est lancée.

- `int MaxCapacity{get;}`
  Cette propriété donne la capacité maximale que peut avoir la chaîne de caractères. Dans l'implémentation courante de *Microsoft* ce champ vaut `Int32.MaxValue` soit 2.147.483.647 caractères.

La classe `StringBuilder` présente les méthodes suivantes :

- `int EnsureCapacity(int capacity)`
  Assure que la capacité est au moins égale à la valeur de `capacity`. Si ce n'est pas le cas, la capacité est augmentée. La nouvelle capacité est retournée.
- `StringBuilder Append(...)`
  Ajoute des caractères à la fin de la chaîne de caractères. Cette méthode existe en de nombreuses versions surchargées.
- `StringBuilder Insert(int index,...)`
  Insère des caractères à la position spécifiée par `index`. La position 0 signifie le premier caractère de la chaîne. Cette méthode existe en de nombreuses versions surchargées.
- `StringBuilder Remove(int startIndex,int length)`
  Supprime les caractères compris entre la position `startIndex` et la position `startIndex+length`. L'exception `ArgumentOutOfRangeException` est lancée si une de ces positions est négative ou excède la taille de la chaîne de caractères.
- `StringBuilder Replace(...old,...new,...)`
  Remplace le ou les caractères spécifiés par `old`, par le ou les caractères spécifiés par `new`. La recherche de `old` peut éventuellement se faire sur une sous chaîne de caractères en utilisant d'autres versions surchargées de cette méthode.

Voici un petit programme qui illustre tout ceci (l'affichage de chaque appel à `Program.Display()` est inséré en commentaire en blanc sur fond noir) :

*Exemple 10-31 :*

```
class Program{
   static void Display(System.Text.StringBuilder s) {
      System.Console.WriteLine("La chaîne : \"{0}\"",s) ;
      System.Console.WriteLine(" Length   : {0}", s.Length) ;
      System.Console.WriteLine(" Capacity : {0}", s.Capacity) ;
   }
   public static void Main(){
      System.Text.StringBuilder s = new
```

```
            System.Text.StringBuilder("hello") ;
        Display(s) ;
        //La chaîne : "hello"
        // Length   : 5
        // Capacity : 16

        s.Insert( 4 , "--salut--" ) ;
        Display(s) ;
        //La chaîne : "hell--salut--o"
        // Length   : 14
        // Capacity : 16

        s.Capacity = 18 ;
        Display(s) ;
        //La chaîne : "hell--salut--o"
        // Length   : 14
        // Capacity : 18

        s.Replace("salut","SALUT À TOUS") ;
        Display(s) ;
        //La chaîne : "hell--SALUT À TOUS--o"
        // Length   : 21
        // Capacity : 36

        s.EnsureCapacity(42) ;
        Display(s) ;
        //La chaîne : "hell--SALUT À TOUS--o"
        // Length   : 21
        // Capacity : 72
    }
}
```

Avant d'avoir recours aux services de la classe `StringBuilder`, vous devez être conscient qu'il est possible de violer la condition d'immutabilité des instances de la classe `String` si vous vous autorisez à utiliser du code non vérifiable. Cette possibilité est exposée en page 508.

# *Les délégations et les délégués*

## *Introduction*

C++ → C♯ C♯ définit la notion de délégués qui est absente en C++.

Cependant cette notion est conceptuellement proche de celle des pointeurs sur fonction et des pointeurs sur méthode du C++. Au même titre que ces derniers, un délégué est une référence vers une ou plusieurs méthodes (statiques ou non).

Les délégués sont néanmoins plus puissants que les pointeurs sur fonctions, puisqu'un même délégué peut référencer plusieurs méthodes. En outre la syntaxe est beaucoup plus claire et conviviale.

C# C# permet la création de classes particulières avec le mot-clé `delegate` en post-fixe. On appelle ces classes *délégations*. Les instances des délégations sont appelées *délégués*.

Conceptuellement, un délégué est une référence vers une ou plusieurs méthodes (statiques ou non). On peut donc « appeler/invoquer » un délégué avec la même syntaxe que l'appel d'une méthode. Ceci provoque l'appel des méthodes référencées. Notez que l'appel de ces méthodes est effectué par le même thread qui réalise « l'appel » au délégué. On parle donc d'appel synchrone.

Il existe de nombreuses délégations de base, définies dans le *framework* .NET. Vous pouvez trouver la liste de ces délégations dans l'article **MulticastDelegate Hierarchy** des **MSDN**. Comme l'indique cet article, chaque délégation, qu'elle soit propriétaire ou standard, est une classe dérivée de la classe `System.MulticastDelegate`. Cette classe est très spéciale, on ne peut en dériver explicitement. Seuls les compilateurs des langages .NET peuvent construire des classes qui dérivent de `MulticastDelegate`. Grâce à certaines méthodes de la classe `MulticastDelegate`, un délégué peut aussi servir à effectuer des appels asynchrones sur une méthode. Cette notion de délégués asynchrones fait l'objet d'une autre section, page 171.

Certains ouvrages utilisent les termes « classe déléguée » ou « type délégué » pour nommer les délégations. D'autres publications utilisent le terme « délégué » pour nommer les délégations. Dans ce cas il faut utiliser les termes « instance de délégué » ou « objet délégué » pour nommer ce que nous appelons ici « délégué ». Soyez vigilants, et repérez à chaque fois la différence entre les types et les instances.

## *Utilisation de délégués avec des méthodes statiques*

Voici un exemple utilisant deux délégations :

*Exemple 10-32 :*

```
public class Program {
  // Définition des délégations Deleg1 et Deleg2.
  delegate void Deleg1();
  delegate string Deleg2(string s);
  static void f1() {
    System.Console.WriteLine("Appel de f1.") ;
  }
  static string f2(string s) {
    string _s=string.Format("Appel de f2 avec l'argument \"{0}\"",s) ;
    System.Console.WriteLine(_s) ;
    return _s ;
  }
  public static void Main() {
    // Crée un délégué instance de Deleg1 référençant la méthode f1().
    Deleg1 d1 = new Deleg1(f1);
    // Appel de f1() avec le délégué d1.
    d1();
    // Crée un délégué instance de Deleg2 référençant la méthode f2().
    Deleg2 d2 = new Deleg2(f2);
```

```
    // Appel de f2("hello") avec le délégué d2.
    string s = d2("hello");
  }
}
```

Ce programme affiche :

```
Appel de f1.
Appel de f2 avec l'argument "hello"
```

Nous utilisons deux délégations, pour bien souligner le fait qu'un délégué ne peut référencer seulement une méthode dont la signature (même ordre, nombre et type des arguments et même type de retour) correspond à celle fournie lors de la déclaration de la délégation. Par exemple la ligne suivante aurait provoqué une erreur de compilation car la méthode, `f1()` n'a pas la signature adéquate :

```
Deleg2 d2  = new Deleg2(f1) ;
```

## *Inférence de la délégation par le compilateur C♯2*

Le compilateur C♯2 introduit une facilité avec la possibilité d'inférer le type d'un délégué que l'on crée. Vous pouvez ainsi assigner directement une méthode à un délégué créé implicitement. Le programme précédent peut ainsi être réécrit comme suit :

*Exemple 10-33 :*

```
public class Program {
   delegate void Deleg1() ;
   delegate string Deleg2(string s) ;
   static void f1() {
      System.Console.WriteLine("Appel de f1.") ;
   }
   static string f2(string s) {
      string _s = string.Format(
                     "Appel de f2 avec l'argument \"{0}\"", s) ;
      System.Console.WriteLine(_s) ;
      return _s ;
   }
   public static void Main() {
      Deleg1 d1 = f1; // Au lieu de   ...new Deleg1(f1)...
      d1() ;
      Deleg2 d2 = f2; // Au lieu de   ...new Deleg2(f1)...
      string s = d2("hello") ;
   }
}
```

Ne vous y trompez pas : ceci est juste une facilité d'écriture. Si vous analysez le code IL produit par cet exemple vous pourrez visualisez qu'il y a bien appel aux constructeurs des délégations `Deleg1` et `Deleg2`.

## *Utilisation de délégués avec des méthodes non statiques*

On peut obtenir le même comportement avec des méthodes non statiques. Naturellement lorsque l'on fournit la méthode à référencer, il faut la lier avec l'objet sur lequel elle doit être appelée. Par exemple :

*Exemple 10-34 :*

```
using System ;
public class Article {
   private int m_Prix = 0 ;
   public Article(int Prix) { m_Prix = Prix ; }
   public int IncPrix(int i) {
      m_Prix += i ;
      return m_Prix ;
   }
}
public class Program {
   public delegate int Deleg(int i) ;
   public static void Main() {
      // Crée un article de prix 100.
      Article article = new Article(100);
      // Crée un délégué référençant IncPrix() sur l'objet article.
      Deleg deleg = article.IncPrix;
      int p1 = deleg(20) ;
      Console.WriteLine(
         "Prix de article après incrément de 20 : {0}", p1) ;
      int p2 = deleg(-10) ;
      Console.WriteLine(
         "Prix de article après un décrément de 10 : {0}", p2) ;
   }
}
```

Ce programme affiche ceci :

```
Prix de article après incrément de 20 : 120
Prix de article après un décrément de 10 : 110
```

## *Utilisation de délégués avec plusieurs méthodes*

On peut référencer plusieurs méthodes (statiques ou non) de mêmes signatures avec le même délégué. Dans ce cas l'appel au délégué entraîne l'appel successif de ces méthodes par le même thread qui a provoqué l'appel. Ces méthodes sont appelées dans l'ordre dans lequel elles ont été ajoutées dans le délégué, chacune avec les mêmes arguments. Par exemple:

*Exemple 10-35 :*

```
using System ;
public class Article {
   public int m_Prix = 0 ;
   public Article(int Prix) { m_Prix = Prix ; }
```

```
   public int IncPrix(int i) {
      m_Prix += i ;
      return m_Prix ;
   }
}
public class Program {
   public delegate int Deleg(int i) ;
   public static void Main() {
      // Crée trois articles de prix différents.
      Article a = new Article(100) ;
      Article b = new Article(103) ;
      Article c = new Article(107) ;
      // Assigne 3 méthodes à un même délégué.
      Deleg deleg = a.IncPrix;
      deleg += b.IncPrix;
      deleg += c.IncPrix;
      // Incrémente les trois prix de 20 en un seul appel à deleg.
      int p1 = deleg(20);
      // Ici p1 vaut 127, le prix du dernier article.
      Console.WriteLine(
         "Prix après incrément de 20 : a:{0} b:{1} c:{2}",
         a.m_Prix , b.m_Prix , c.m_Prix) ;
      // Décrémente les trois prix de 10 en un seul appel à deleg.
      int p2 = deleg(-10);
      // Ici p2 vaut 117, le prix du dernier article.
      Console.WriteLine(
         "Prix après un décrément de 10 : a:{0} b:{1} c:{2}",
         a.m_Prix , b.m_Prix , c.m_Prix ) ;
   }
}
```

Cet exemple affiche :

```
Prix après incrément de 20 : a:120 b:123 c:127
Prix après un décrément de 10 : a:110 b:113 c:117
```

Si la signature retourne une valeur dans le cas où il y a plusieurs méthodes référencées c'est la valeur retournée par le dernier appel qui est renvoyée par le délégué.

Bien que cet exemple ne le montre pas, le même délégué peut à la fois référencer des méthodes statiques et des méthodes non statiques, de la même classe ou de classes différentes. Il suffit juste que ces méthodes aient la même signature et le même type de valeur de retour.

## *La classe System.Delegate*

Il faut savoir qu'en interne, lorsque vous placez plusieurs méthodes dans le même délégué, une instance de la classe `System.Deletage` est créée pour chaque méthode. En fait, la classe `MulticastDelegate` est une liste d'instances de la classe `System.Delegate`. L'exemple suivant montre comment utiliser la classe `Delegate` pour invoquer une à une les méthodes contenues dans un délégué. Notez le recours à la méthode `GetInvocationList()` pour obtenir la liste des délégués.

*Exemple 10-36 :*

```
using System ;
public class Article {
   public int m_Prix = 0 ;
   public Article(int Prix) { m_Prix = Prix ; }
   public int IncPrix(int i) {
      m_Prix += i ;
      return m_Prix ;
   }
}
public class Program {
   public delegate int Deleg(int i) ;
   public static void Main() {
      Article a = new Article(100) ;
      Article b = new Article(103) ;
      Article c = new Article(107) ;
      Deleg delegs = a.IncPrix ;
      delegs += b.IncPrix ;
      delegs += c.IncPrix ;
      int somme = 0 ;
      // Obtient la liste des délégués.
      Delegate[] delegArr = delegs.GetInvocationList();
      // Invoque une à une chaque méthode référencée.
      foreach (Deleg deleg in delegArr)
         somme += deleg(20);
      Console.WriteLine(
         "Prix après incrément de 20 : a:{0} b:{1} c:{2}",
         a.m_Prix , b.m_Prix , c.m_Prix) ;
      Console.WriteLine("Somme des prix:{0}", somme ) ;
   }
}
```

Cet exemple affiche :

```
Prix après incrément de 20 : a:120 b:123 c:127
Somme des prix:370
```

## *Subtilité dans la manipulation des délégués*

L'exemple précédent montre que l'on peut ajouter (respectivement supprimer) un délégué à un autre délégué de même classe avec l'opérateur « += » (respectivement « -= »). Si le délégué ajouté contient plusieurs références de méthodes, ce sont toutes ces références qui sont ajoutées à la fin de la liste des références du délégué cible. En revanche, dans le cas de la suppression, il se peut que l'opération ne puisse se faire si la sous liste de références du délégué supprimé n'apparaît pas dans la liste de référence du délégué cible. L'exemple suivant illustre ceci :

*Exemple 10-37 :*

```
using System ;
public class Article {
```

```
   public int m_Prix = 0 ;
   public Article(int Prix) { m_Prix = Prix ; }
   public int IncPrix(int i) {
      m_Prix += i ;
      return m_Prix ;
   }
}
public class Program {
   public delegate int Deleg(int i) ;
   public static void Main() {
      Article a = new Article(100) ;
      Article b = new Article(103) ;
      Article c = new Article(107) ;

      // Construction du délégué ( a.IncPrix , b.IncPrix , c.IncPrix ).
      Deleg deleg = a.IncPrix ;
      Deleg deleg1 = b.IncPrix ;
      deleg1 += c.IncPrix ;
      deleg += deleg1 ;
      deleg(10) ;
      Console.WriteLine("a:{0} b:{1} c:{2}",
         a.m_Prix , b.m_Prix , c.m_Prix ) ;

      // Essai de suppression du délégué ( a.IncPrix , c.IncPrix )
      // non présent dans ( a.IncPrix , b.IncPrix , c.IncPrix ).
      Deleg deleg2 = a.IncPrix ;
      deleg2 += c.IncPrix;
      deleg -= deleg2 ;
      deleg(10) ;
      Console.WriteLine("a:{0} b:{1} c:{2}",
         a.m_Prix , b.m_Prix , c.m_Prix ) ;

      // Essai de suppression du délégué ( a.IncPrix , b.IncPrix )
      // présent dans ( a.IncPrix , b.IncPrix , c.IncPrix ).
      Deleg deleg3 = a.IncPrix ;
      deleg3 += b.IncPrix;
      deleg -= deleg3 ;
      deleg(10) ;
      Console.WriteLine("a:{0} b:{1} c:{2}",
         a.m_Prix , b.m_Prix , c.m_Prix) ;
   }
}
```

Cet exemple affiche :

```
a:110 b:113 c:117
a:120 b:123 c:127
a:120 b:123 c:137
```

Dans le premier essai de suppression, le délégué `deleg` n'est pas modifié. Pour infirmer ou confirmer le bon déroulement d'une suppression, vous pouvez inspecter la taille de la liste des instances de la classe `Delegate` obtenue avec la méthode `GetInvocationList()`.

# *Les types nullables*

Les concepteurs de C♯2 ont ajouté la notion de type nullable pour pallier une faiblesse des types valeurs par rapport aux types références. Il est donc essentiel d'avoir bien assimilé ces deux notions présentées en début de chapitre avant de pouvoir aborder cette section.

## *La problématique d'une valeur nulle pour les types valeurs*

Une référence de type un type référence est nulle lorsqu'elle ne référence aucun objet. C'est la valeur prise par défaut par toute référence. Il suffit d'analyser le code de n'importe qu'elle application pour s'apercevoir que les développeurs exploitent intensivement les références nulles. En général, l'utilisation d'une référence nulle permet de communiquer une information :

- Une méthode qui doit retourner une référence vers un objet retourne une référence nulle pour signifier que l'objet demandé ne peut être fabriqué ou trouvé. Cela évite d'implémenter un code d'erreur de retour binaire.
- Lorsque vous rencontrez une méthode qui accepte un argument de type référence qui peut être nulle, cela signifie en général que l'argument est optionnel.
- Un champ de type référence nulle peut signifier que l'objet qui le présente est en cours d'initialisation, de mise à jour ou de destruction et n'a donc pas un état valide.

La notion de nullité est aussi largement exploitée dans les bases de données relationnelles pour signifier qu'une valeur dans un *enregistrement* n'a pas été assignée. La notion de nullité peut aussi servir pour désigner un attribut optionnel dans un élément XML.

Parmi les nombreuses différences entre les types valeurs et les types références, nous pouvons nous pencher sur le fait qu'une instance d'un type valeur ne peut avoir de valeur nulle. Cela pose en général de nombreux problèmes. Par exemple, comment interpréter une valeur entière nulle (non encore assignée) récupérée d'une base de donnée ? De multiples solutions existent mais aucune d'entre elles n'est pleinement satisfaisante :

- Si tout l'intervalle de valeurs entières n'est pas utilisable, on crée une convention. Par exemple, une valeur entière nulle est représentée par un entier égal à 0 ou -1. Les nombreux désavantages de cette solution sont évidents : contrainte à maintenir partout dans le code, possibilité d'évolution de l'intervalle de valeurs pris par ce champ particulier etc.
- On crée une structure *wrapper* contenant deux champs, un entier et un booléen qui, si positionné à `false`, signifie que nous avons une valeur nulle. Ici, on doit gérer une structure en plus pour chaque type valeur, et un état en plus pour chaque valeur.
- On crée une classe *wrapper* contenant un champ entier. Ici, le désavantage est qu'en plus du maintient d'une nouvelle classe, on surcharge le ramasse-miettes en créant de nombreux objet sur le tas.
- On utilise le boxing, par exemple en transtypant notre valeur entière en une référence de type `object`. En plus de ne pas être *type-safe*, cette solution a aussi le désavantage de surcharger le ramasse-miettes.

Pour pallier à ce problème récurrent, les concepteurs de C#2 ont décidé d'ajouter au langage la notion de *types nullables*.

## *La structure System.Nullable<T>*

Le *framework* .NET 2005 présente la structure générique `System.Nullable<T>` définie comme ceci :

```
public struct System.Nullable<T> {
   public Nullable(T value) ;
   public static explicit operator T(T? value) ;
   public static implicit operator T?(T value) ;

   public bool HasValue { get ; }
   public T Value { get ; }

   public override bool Equals(object other) ;
   public override int GetHashCode() ;
   public T GetValueOrDefault() ;
   public T GetValueOrDefault(T defaultValue) ;
   public override string ToString() ;
}
```

Cette structure répond bien à la problématique des valeurs nulles lorsque le type paramètre `T` prend la forme d'un type valeur tel que `int`. Voici un petit exemple qui illustre l'utilisation de cette structure. La première version de `Fct()` exploite la nullité d'une référence de type `string` tandis que la seconde version exploite la nullité d'une instance de la structure `Nullable<int>` :

*Exemple 10-38 :*

```
class Foo {
   static string Fct(string s) {
      if (s == null)
         return null ;
      return s + s ;
   }
   static System.Nullable<int> Fct(System.Nullable<int> ni){
      if (!ni.HasValue)
         return ni ;
      return (System.Nullable<int>) (ni.Value + ni.Value) ;
   }
}
```

## *Évolution de la syntaxe C# : Nullable<T> et le mot clé null*

La syntaxe C# vous permet d'assigner et de comparer le mot clé `null` à une instance de `System.Nullable<T>` :

```
Nullable<int> ni = null;
System.Diagnostics.Debug.Assert( ni == null ) ;
```

Ces deux lignes de code sont équivalentes à :

```
// Appel du  constructeur par défaut qui positionne HasValue à false.
Nullable<int> ni = new Nullable<int>() ;
System.Diagnostics.Debug.Assert( !nullable1.HasValue ) ;
```

L'utilisation de la structure `System.Nullable<T>` est intuitive, mais peut rapidement amener à se poser des questions. Il n'est pas évident que ces deux programmes sont équivalents (les deux codes IL générés sont équivalents) :

*Exemple 10-39 :*

```
class Program{
   static void Main(){
      System.Nullable<int> ni1 = 3 ;
      System.Nullable<int> ni2 = 3 ;
      bool b = (ni1 == ni2) ;
      System.Diagnostics.Debug.Assert(b) ;
      System.Nullable<int> ni3 = ni1 + ni2 ;
      ni1++;
   }
}
```

*Exemple 10-40 :*

```
using System ;
class Program {
   static void Main() {
      Nullable<int> ni1 = new Nullable<int>(3) ;
      Nullable<int> ni2 = new Nullable<int>(3) ;
      bool b = (ni1.GetValueOrDefault() == ni2.GetValueOrDefault()) &&
               (ni1.HasValue == ni2.HasValue);
      System.Diagnostics.Debug.Assert( b ) ;
      Nullable<int> ni3 = new Nullable<int>() ;
      if (ni1.HasValue && ni2.HasValue)
         ni3 = new Nullable<int>( ni1.GetValueOrDefault() +
                                  ni2.GetValueOrDefault() ) ;
      if (ni1.HasValue)
         ni1 = new Nullable<int>( ni1.GetValueOrDefault() + 1 ) ;
   }
}
```

En outre, cela peut sembler étrange que la l'instruction `ni++` appelée lorsque la variable `ni` est sensée être `null` ne provoque pas une exception de type `NullReferenceException`.

## *Évolution de la syntaxe C♯ : équivalence entre Nullable<T> et T ?*

En C♯2, vous pouvez faire suivre le nom d'un type valeur non nullable `T` par un point d'interrogation. Dans ce cas, le compilateur C♯2 remplacera toute expression `T?` par `Nullable<T>`. Pour simplifier, vous pouvez imaginer que ceci est un prétraitement effectué directement sur le code source, un peu comme un précompilateur. Ainsi, la ligne suivante...

```
int? i = null ;
```

...est équivalente à :

```
Nullable<int> i = null ;
```

Par exemple, les deux méthodes suivantes sont rigoureusement équivalentes :

*Exemple 10-41 :*

```
class Foo {
   static System.Nullable<int> Fct1( System.Nullable<int> ni ) {
      if ( !ni.HasValue )
         return ni ;
      return (System.Nullable<int>) ( ni.Value + ni.Value ) ;
   }
   static int? Fct2(int? ni){
      if (ni == null)
         return ni;
      return ni + ni ;
   }
}
```

En général, les instances de types nullables équivalentes se mélangent bien :

*Exemple 10-42 :*

```
class Program{
   static void Main(){
      int? ni1 = null ;
      int? ni2 = 9 ;
      int? ni3 = ni1 + ni2 ; // OK, ni3 vaut null.
      int? ni4 = ni1 + 3 ;   // OK, ni4 vaut null.
      int? ni5 = ni2 + 3 ;   // OK, ni5 vaut 12.
      ni1++ ;                // OK, ni1 reste à null.
      ni2++ ;                // OK, ni2 passe à 1.
   }
}
```

En revanche, le compilateur vous empêche de convertir implicitement un objet d'un type nullable dans le type sous-jacent. De plus, il est dangereux de réaliser une telle conversion explicitement sans tests préalables puisque vous risquer de lever une exception de type `InvalidOperationException`.

*Exemple 10-43 :*

```
class Program{
   static void Main(){
      int? ni1 = null ;
      int? ni2 = 9 ;
      int i1 = ni1 ;        // KO: Cannot implicitly convert
                            //     type 'int?' to 'int.
      int i2 = ni2 ;        // KO: Cannot implicitly convert
```

```
                              //     type 'int?' to 'int.
      int i3 = ni1 + ni2 ;    // KO: Cannot implicitly convert
                              //     type 'int?' to 'int.
      int i4 = ni1 + 6 ;      // KO: Cannot implicitly convert
                              //     type 'int?' to 'int.
      // OK à la compilation mais une exception de type
      // InvalidCastException est lancée à l'exécution,
      // car ni1 est toujours nulle.
      int i5 = (int)ni1 ;
   }
}
```

## *Pas de traitement spécial de bool? en C# 2.0*

Contrairement à ce que vous avez peut être vu dans les versions bêtas de C# 2.0, en version finale il n'y a plus de traitement spécial du type `bool?` par les mots-clés `if`, `while` et `for`. Ainsi cet exemple ne compile pas :

*Exemple 10-44 :*

```
class Program{
   static void Main(){
      bool? b = null ;
      // Cannot implicitly convert type 'bool?' to bool.
      if ( b ) { /*...*/ }
   }
}
```

## *Les types nullables et les opérations de boxing et de unboxing*

Lors de la conception de .NET 2.0, jusqu'au dernier moment les types nullables représentaient un artefact qui n'impactait pas le CLR. Ils étaient implémentés seulement à l'aide du compilateur C#2.0 et de la structure `System.Nullable<T>`. Cela posait un problème puisque lorsqu'une instance d'un type nullable était boxée elle ne pouvait être en aucun cas nulle. Il y avait bien une incohérence avec la manipulation des types références :

```
string s = null ;
     object os = s ;    // os est une référence nulle
     int?   i = null ;
     object oi = i ;    // oi n'était pas une référence nulle
```

Sous la pression de la communauté, les ingénieurs de *Microsoft* ont décidé de remédier à ceci. Ainsi, l'assertion du programme suivant est vérifiée :

*Exemple 10-45 :*

```
class Program{
   static void Main(){
     int? ni = null ;
     object o = ni ;    // boxing
     System.Diagnostics.Debug.Assert( o == null ) ;
```

```
   }
}
```

Du point de vue du CLR, une instance d'un type valeur `T` boxée peut être nulle. Si elle n'est pas nulle, le CLR ne stocke pas d'information quant à savoir si elle était originalement issue du type `T` ou `Nullable<T>`. Vous pouvez ainsi unboxer un tel objet dans l'un de ces deux types comme le montre l'exemple suivant. Soyez néanmoins conscient que vous ne pouvez pas unboxer une valeur nulle :

*Exemple 10-46 :*

```
class Program {
   static void Main() {
      int i1 = 76 ;
      object o1 = i1 ;      // boxing d'un int
      int? ni1 = (int?)o1 ; // unboxing en un int?
      System.Diagnostics.Debug.Assert( ni1 == 76 ) ;

      int? ni2 = 98 ;
      object o2 = ni2 ;     // boxing d'un int?
      int i2 = (int)o2 ;    // unboxing en un int
      System.Diagnostics.Debug.Assert( i2 == 98 ) ;

      int? ni3 = null ;
      object o3 = ni3 ;     // boxing d'un nullable nul
      int i3 = (int)o3 ;    // unboxing -> NullReferenceException levée !
   }
}
```

## *Les structures et les énumérations nullables*

La notion de type nullable peut s'utiliser aussi sur vos propres structures et énumérations. Cela peut mener à des erreurs de compilations rédhibitoires illustrées par l'exemple suivant où la structure `Nullable<MyStruct>` ne supporte pas les membres de `MyStruct`.

*Exemple 10-47 :*

```
struct Struct {
   public Struct(int i) { m_i = i ; }
   public int  m_i ;
   public void Fct(){}
}
class Program {
   static void Main(){
      Struct? ns1 = null ; // OK
      Struct? ns2 = new Struct?(3) ; // KO: Cannot implicitly convert
                                     // type 'int' to 'Struct'.
      Struct? ns3 = new Struct?() ;  // OK Struct.ctor() par défaut
                                     // est appelé.
      Struct? ns4 = new Struct(3) ;  // OK
```

```
      Struct? ns5 = new Struct() ;   // OK Struct.ctor() par défaut
                                     // est appelé.
      ns4.m_i = 8 ; // KO: System.Nullable<Struct>' does not
                    // contain a definition for 'm_i'.
      ns4.Fct() ;   // KO: System.Nullable<Struct>' does not
                    // contain a definition for 'Fct'.
   }
}
```

En revanche, si besoin est, le compilateur saura utiliser vos redéfinitions d'opérateurs :

*Exemple 10-48 :*

```
struct Struct {
   public Struct(int i) { m_i = i ; }
   public int  m_i ;
   public static Struct operator +( Struct a, Struct b) {
      return new Struct( a.m_i + b.m_i ) ; }
}
class Program {
   static void Main() {
      Struct? ns1 = new Struct(3) ;
      Struct? ns2 = new Struct(2) ;
      Struct? ns3 = null ;
      Struct? ns4 = ns1 + ns2 ; // OK, ns4.m_i vaut 5.
      Struct? ns5 = ns1 + ns3 ; // OK, ns5 vaut null.
   }
}
```

En ce qui concerne une instance d'une énumération nullable, ayez conscience que vous devez toujours obtenir la valeur sous-jacente pour l'utiliser. Par exemple :

*Exemple 10-49 :*

```
class Program{
   enum MyEnum { VAL1, VAL2 }
   static void Main(){
      MyEnum? e = null;
      if( e == null )
         System.Console.WriteLine("e est null") ;
      else
         switch(e.Value){  // Ici on est sûr que e n'est pas null.
            case MyEnum.VAL1: System.Console.WriteLine("e vaut VAL1") ;
                              break ;
            case MyEnum.VAL2: System.Console.WriteLine("e vaut VAL2") ;
                              break ;
         }
   }
}
```

# Définir un type sur plusieurs fichiers sources

C#2 présente la possibilité d'étaler la déclaration d'une classe, d'une structure ou d'une interface sur plusieurs fichiers sources. Les documentations anglaises ont recours à l'expression *partial type* pour nommer cette possibilité. Nous aurons recours dans la suite aux expressions *type défini sur plusieurs fichiers sources* et *définition partielle d'un type* qui sont plus parlantes que *type partiel*. Notez que vous ne pouvez déclarer une délégation ou une énumération sur plusieurs fichiers sources.

Pour déclarer un type sur plusieurs fichiers sources, il faut faire précéder un des mots clés class, struct ou interface par le mot clé partial dans chaque déclaration partielle. Les définitions partielles doivent impérativement appartenir au même espace de nom. De plus, le compilateur autorise qu'un type n'admettre qu'une seule définition partielle précédée du mot clé partial.

Les différents fichiers sources contenant les différentes définitions partielles d'un même type doivent tous être compilés en une seule fois. Une conséquence est que cette possibilité ne peut être utilisée pour étaler la définition d'un même type sur plusieurs modules d'un même assemblage ou sur plusieurs assemblages. Cette limitation est une conséquence du fait que la possibilité de définir un type sur plusieurs fichiers sources n'est que du *sucre syntaxique*. Elle ne concerne que le compilateur C#2 csc.exe et n'a aucune incidence ni sur la façon dont les types sont contenus dans les assemblages, ni par leurs traitements par le CLR à l'exécution.

Le fait de définir un type sur plusieurs fichiers sources a tendance à complexifier la lecture du code. Aussi, nous vous conseillons d'avoir recours à cette possibilité que lorsque vous souhaitez générer du code. Dans ce cas, il est appréciable de ne générer que partiellement le code d'un type afin de faire interagir aisément le code généré avec le code fait « à la main ». Cela évite d'avoir recours à des artifices pour rendre possible cette interaction. *Visual Studio 2005* inclut plusieurs générateurs de code qui exploite cette notion de définition partielle : générateur de formulaires *Windows Forms* (page 669), le générateur de `DataSet` typés (page 731), le générateur de pages web (page 869) etc.

Enfin, précisons qu'un type encapsulé dans un autre type peut être défini sur plusieurs fichiers sources. Dans ce cas chaque définition partielle d'un type encapsulé doit impérativement être rédigée dans une des définitions partielles du type encapsulant.

## Les modificateurs qui doivent être répétés

Les différentes définitions partielles d'un même type doivent toutes être précédées du mot clé partial. En outre, si le type est générique, la définition des types paramètres doit être répétées pour chaque définition partielle. Les noms des types paramètres ainsi que leurs positions dans la liste doivent être identiques pour chaque définition partielle.

## Les modificateurs qui peuvent être répétés sans répercussion sur leurs effets

Les modificateurs de type suivants peuvent être répétés ou non dans les différentes définitions partielles d'un type. Il suffit qu'ils soient présents sur une seule définition partielle pour que leurs effets se répercutent sur le type entier. Bien entendu, le compilateur C#2 détecte et sanctionne d'une erreur toutes les incohérences telle qu'une déclaration partielle ayant une visibilité publique et une autre ayant une visibilité interne.

- Les mots-clés abstract et sealed. Précisons qu'un même type ne peut être à la fois « abstract » et « sealed ».
- La visibilité d'un type.
- La classe de base d'un type.
- L'ensemble des contraintes pour un type paramètre.

## *Les modificateurs dont les effets se cumulent*

Les modificateurs de type suivant peuvent être répétés dans les différentes définitions partielles d'un type. Dans ce cas leurs effets se cumulent. Ici aussi le compilateur C# détecte et sanctionne d'une erreur toutes les incohérences telle que la définition répétée d'une même méthode dans plusieurs déclarations partielles d'un même type.

- Les membres. L'ensemble des membres d'un type défini sur plusieurs fichiers sources est l'union des membres de chaque définition partielle.
- Les attributs. L'ensemble des attributs s'appliquant sur un type défini sur plusieurs fichiers sources est l'union des attributs appliqués sur chaque définition partielle. En particulier, le type concerné sera marqué plusieurs fois par un même attribut si celui-ci marque plusieurs définitions partielles.
- Les interfaces implémentées. L'ensemble des interfaces implémentées par un type défini sur plusieurs fichiers sources est l'union des interfaces déclarées dans chaque définition partielle.

## *Les modificateurs dont les effets sont locaux*

Seul le mot clé unsafe rentre dans cette catégorie. Il peut être appliqué sur une ou plusieurs définitions partielles d'un même type. Dans ce cas, son effet n'est limité qu'aux membres déclarés dans les définitions partielles concernées.

# 11

# *Notions de classe et d'objet*

## *Remarques sur la programmation objet*

C++ → C♯ C♯ peut être considéré 'plus orienté objet' que C++. En effet la notion de fonction globale ou de variable globale n'existe pas en C♯. Seules les méthodes (statiques ou non) de classes existent.

C♯ Le concept de la *programmation procédurale* est construit autour de la notion de fonction. Tout programme est un ensemble de fonctions s'appelant entre elles.

Le concept de la *programmation objet* est construit autour des notions d'objet et de classe. **Une classe est l'implémentation d'un type de données** (au même titre que `int` ou une structure). **Un objet est une instance d'une classe**.

Dans la programmation objet, tout programme peut être vu comme un ensemble d'objets qui interagissent entre eux. Nous allons détailler comment C♯ traite la programmation orientée objet (POO). Cependant cet ouvrage n'est pas un ouvrage sur la POO. En effet, la plupart des concepts sont abordés mais cela ne suffit pas pour savoir les utiliser à bon escient.

## *Notions et vocabulaire*

C++ → C♯ En C♯, en français, le mot *attribut* est réservé pour d'autres notions que celle de champ d'une classe. Cette notion d'attribut est présentée page 248. De plus on verra que C♯ distingue la notion de champs et de propriétés.

Enfin puisque la plateforme .NET dispose d'un ramasse-miettes, un destructeur C♯ est très différent d'un destructeur C++.

C♯ Une *classe* est un type de données. Les notions de variable d'un type et *d'instance d'une classe* sont similaires, dans le sens où les traitements s'organisent autour des données. Un *objet* est une instance d'une classe. La notion de classe définit un concept alors que la notion d'objet

définit l'instance d'un concept. La notion de classe est au concept de *voiture* ce que la notion d'objet est à une voiture particulière. Une voiture dans la rue est une instance du concept de voiture.

Il est intéressant de souligner un paradoxe de la programmation par objets. Les objets n'existent qu'à l'exécution et sont crées, gérés et détruits par l'environnement d'exécution. Les développeurs ne font qu'écrire des classes qui lors de l'exécution, entraîneront la création d'objets. Aussi, certains préféreraient utiliser la terminologie programmation par classes.

Dans le contexte d'un langage objet tel que C♯, le terme *variable* est utilisé pour désigner un *objet* instance d'un type valeur, alloué dans le corps d'une méthode et dont la durée de vie ne dépasse pas la portée de cette méthode.

Une classe est constituée de plusieurs entités de différentes natures : les champs, les propriétés, les méthodes, les événements, les indexeurs et les types encapsulés. On appelle ces entités les *membres* de la classe. Nous allons avoir l'occasion dans le présent chapitre de détailler chacune de ces entités.

Chaque objet renferme une quantité d'information sous forme de valeurs stockées dans ses champs. L'ensemble de ces valeurs est appelé l'*état de l'objet*.

La notion de méthode se rapproche de celle de fonction, à ceci près qu'une méthode est appelée sur une instance de la classe. Une méthode agit sur l'objet sur lequel elle est appelée. Elle peut aussi bien lire ou écrire les champs de l'objet, qu'effectuer une action sur l'objet.

Un objet a une durée de vie qui est un sous intervalle de la durée d'exécution du programme qui l'héberge. Un objet est nécessairement construit à un moment donné et détruit plus tard à un autre moment. Une méthode est automatiquement appelée lors de la construction. Elle est nommée *constructeur*. Elle permet par exemple d'initialiser l'objet ou d'allouer des ressources consommées par l'objet. Il peut y avoir plusieurs constructeurs pour une même classe car il peut y avoir plusieurs façons de construire un objet.

Une méthode est automatiquement appelée lors de la destruction d'un objet par le ramasse-miettes de la plateforme .NET. La destruction des objets est un sujet sensible en .NET/C♯ qui doit absolument être bien compris.

# *Définition d'une classe*

C++ → C♯ C♯ ne connaît pas l'opérateur de résolution de portée « :: » tel qu'il est utilisé en C++ lors de la définition des classes (cet opérateur a été introduit avec C♯2 néanmoins, mais pour d'autres raisons).

Par conséquent, la définition d'une classe (le corps des méthodes ainsi que l'initialisation des champs statiques compris) doit se faire absolument à l'intérieur de la paire d'accolade après le mot-clé `class`.

Le concept de classes et méthodes amies d'une classe, présent dans le C++, a totalement disparu en C♯. La raison est que ce concept représente une violation du concept d'encapsulation qui est plus prépondérant en POO. Signalons cependant le nouveau concept .NET 2.0 d'assemblages amis qui se rapproche de ceci (présenté en page 27).

Enfin le caractère « ; » n'est pas obligatoire à la fin de la définition d'une classe.

C♯ La définition d'une classe se fait avec le mot-clé `class`. Voici un exemple :

*Exemple 11-1 :*

```
class Program{
   // Ici sont placés les membres de la classe Program.
   static void Main() {
      // Ici sont placées les instructions de la méthode Main().
   }
   // Ici aussi, sont placés les membres de la classe Program.
}
```

Les membres d'une classe sont des entités déclarées dans la classe. Il y a six sortes de membres :

- Les champs
- Les propriétés
- Les indexeurs
- Les méthodes
- Les événements
- Les types encapsulés dans la classe

À part quelques détails que nous soulignerons, tout ce qui va être dit ici à propos des membres d'une classe est aussi valable pour les membres d'une structure.

## *Accès aux membres*

C++ → C# L'accès aux membres est plus simple en C# qu'en C++. Puisque le mode d'allocation (statique ou dynamique) d'un objet n'est plus du ressort du programmeur, il n'y a plus lieu de faire la différence entre les objets de type valeur et les objets de type référence. Ainsi l'opérateur flèche « -> » disparaît et seul reste l'opérateur point « . ».

C# Lorsqu'une zone de code a accès à un membre (non statique) d'une classe (respectivement d'une structure), on peut utiliser l'opérateur point « . » à partir d'une instance de cette classe (respectivement de cette structure) pour accéder au membre. Ceci est valable que ...

- ...le membre soit un champ une propriété une méthode un événement ou un type.
- ...l'objet soit de type valeur (cas d'une structure) ou de type référence (cas d'une classe).

# *Les champs*

C++ → C# Quelques petites différences entre C# et C++.

Tout d'abord, les champs (statiques ou non) peuvent être initialisés directement dans leurs déclarations au sein de la classe. La valeur d'un champ non statique d'un objet sera affectée avant l'appel au constructeur. La valeur d'un champ statique sera affectée avant la création de toutes les instances de la classe.

En revanche, on perd la syntaxe C++ d'initialisation de champs directement après le prototype du constructeur :

```
ctor(int champ) :m_Champ(champ) {}
```

Les spécifications du compilateur C# sont moins permissives que celles du C++. Elles obligent que tous les champs non statiques de type valeur soient initialisés avant la fin de l'appel d'un constructeur de la classe. Les champs de type référence non initialisés sont automatiquement positionnés à `null`.

Enfin un champ peut être `const`, c'est-à-dire initialisé directement au sein de la classe ou `readonly`, c'est-à-dire initialisé au sein de la classe ou dans les constructeurs. Dans les deux cas le champ n'est plus modifiable après son initialisation.

C# La notion de champ d'une classe est la même que la notion de champ d'une structure. L'ensemble des états des champs non statiques d'une instance d'une classe représente l'état de cette instance. Un champ peut être de n'importe quel type (primitif, énumération, structure, classe, interface...).

Lorsqu'un objet A a pour champ un autre objet B, on dit qu'il y a une agrégation de A sur B si B est de type valeur. Si B est de type référence, on dit que A a une *référence* sur B. Cette distinction est importante pour une approche de type UML ou OMT.

## *Initialisation d'un champ*

Tous les champs de type valeur doivent être initialisés avant que l'objet puisse être utilisé et donc, avant la fin de la construction de l'objet. Les champs peuvent être initialisés de deux façons :

- Directement lors de leur déclaration au sein de la classe (`Champs1` et `Champs2` dans l'exemple).
- Dans tous les constructeurs de la classe (`Champs1` et `Champs3` dans l'exemple).

*Exemple 11-2 :*

```
class Foo {
   int Champ1 = 4 ;
   int Champ2 = 3 ;
   int Champ3 ;
   public Foo() { // Un constructeur public de la clase Foo.
      Champ1 = 7 ;
      Champ3 = 6 ;
   }
}
```

On voit bien avec `Champ1` que les deux façons ne sont pas incompatibles, cependant l'initialisation de `Champ1` dans la classe est inutile ici, puisque c'est son initialisation dans le constructeur qui sera exécutée en dernier.

## *Champs constants*

On peut rendre un champ constant, c'est-à-dire qu'il prend sa valeur à l'initialisation (au sein de la classe ou dans le constructeur) puis ne peut plus être modifié. C# propose deux façons de rendre un champ constant :

- On le déclare avec le mot-clé `const`.
- On le déclare avec le mot-clé `readonly`.

`readonly` autorise l'initialisation du champ au sein de la classe ou dans le constructeur, alors que `const` n'autorise que l'initialisation du champ au sein de la classe.

*Exemple 11-3 :*

```
class Foo {
   const      int Champ1 = 4 ;
   readonly   int Champ2 = 3 ;
   // Un constructeur public de la classe Foo.
   public Foo() {
      // Champ1 ne peut être initialisé ici, sous peine d'une
      // erreur de compilation.
      Champ2 = 7 ;
   }
}
```

Ceci implique qu'un champ de type référence déclaré avec `const` doit impérativement être initialisé lors de sa déclaration au sein de la classe. Dans le cas contraire le compilateur détecte une erreur.

Une autre implication est qu'un champ de type référence déclaré avec `readonly` doit impérativement être initialisé avec un objet alloué avant la fin d'au moins un constructeur. Dans le cas contraire le compilateur ne dit rien, mais ce champ ne sert alors absolument à rien puisqu'il ne référence aucun objet (i.e il est positionné à `null`) et ne peut être modifié.

## *Un problème potentiel à l'initialisation des champs*

Le compilateur C# est incapable de détecter une boucle dans le graphe des agrégations/références d'objets. C'est-à-dire qu'il ne voit pas que le programme suivant n'est pas valide. En effet `B` instancie un objet de la classe `A`, qui instancie un objet de la classe `B`, qui instancie un objet de la classe `A` etc ... La terminaison du programme est provoquée soit par le dépassement de limite de la taille de la pile (à cause des nombreux appels de méthodes, imbriqués) soit par le dépassement de limite de la taille du tas.

*Exemple 11-4 :*

```
class Program {
   static void Main() {
      B b = new B () ;
   }
}
class A {
   B m_Champ = new B () ;
}
class B {
   A m_Champ = new A () ;
}
```

# Les méthodes

C++ → C# De nombreuses différences existent entre C# et C++ en ce qui concerne les méthodes. Notez que la possibilité d'avoir des méthodes constantes (i.e qui ne changent pas les champs de l'objet) du C++, n'existe pas en C#.

En C++ le passage d'argument à une méthode ou une fonction peut se faire soit par pointeur, soit par valeur soit par référence.

C# permet pour toute méthode, le passage d'argument par valeur et par référence. Sous certaines conditions C# permet aussi le passage par pointeur.

Par défaut, C# fait passer les arguments de type valeur par valeur, et les arguments de type référence par référence.

La syntaxe du passage par référence des arguments de type valeur a changé.

C# introduit la possibilité qu'un argument ne soit que 'out', i.e on ne s'intéresse qu'à sa valeur de sortie.

Enfin les arguments par défaut du C++ ont disparu. Ils sont remplacés par un concept beaucoup plus puissant, proche du passage d'argument en nombre indéterminé du C++ (utilisé notamment par la fonction `printf`).

C# Une méthode d'instance d'une classe est une fonction qui porte sur les objets de la classe. Une méthode peut manipuler les champs et propriétés de la classe afin d'effectuer un calcul ou afin de modifier l'état de l'objet.

*Exemple 11-5 :*

```
public class Article {
   public decimal PrixHT = 10 ;
   public decimal TVA = 19.6M ;
   public decimal GetPrixTTC() {  // Une méthode de la classe Article.
      return PrixHT + PrixHT * (TVA / 100);
   }
}
```

Les méthodes font partie de la classe au même titre que les champs et les propriétés. Les méthodes jouent un rôle essentiel en C# et rassemblent une grande partie des points à maîtriser de ce langage.

Pour une bonne compréhension de ce qui suit, il est préférable de bien avoir assimilé les notions de type valeur et de type référence, décrites page 339.

## Passage d'arguments par valeur et par référence

En théorie de la programmation il n'a que deux façons bien distinctes de faire passer une variable (un objet) sous forme d'un argument, à une méthode :

- Le *passage d'argument par référence* :
  Une référence sur l'objet est passée. Ceci implique que la méthode appelée et la méthode appelante utilisent le même objet. La conséquence majeure est que si la méthode appelée modifie l'état de l'objet, la méthode appelante utilisera l'objet avec ces modifications.

- Le *passage d'argument par valeur* :
  La valeur (l'état) de l'objet est passée. Ceci implique que la méthode appelée et la méthode appelante utilisent chacune un objet différent. Le compilateur s'occupe de créer une copie (un clone) de l'objet passé sur la pile de l'unité d'exécution courante. Cette copie n'est utilisable que par la fonction appelée et est détruite lorsque l'unité d'exécution sort de la fonction appelée et retourne à la fonction appelante. La conséquence majeure est que si la fonction appelée modifie l'état de l'objet (du clone), la fonction appelante ne verra jamais ces modifications.

## *Les règles que C# applique par défaut*

Le fait qu'il existe un passage d'argument par valeur et un passage d'argument par référence est à mettre en relation avec le fait qu'en C# chaque type est soit un type valeur, soit un type référence. Très logiquement, par défaut C# applique cette règle :

- Une variable (un objet) de type valeur est passée par valeur à une méthode.
- Un objet de type référence est passé par référence à une méthode.

Par exemple :

*Exemple 11-6 :*

```
public class Article { public int Prix = 0 ; }
class Program {
   static void Main() {
      int i = 10 ;   // int est un type valeur.
      Article article = new Article() ; // Article est un type référence.
      article.Prix = 10 ;
      fct(i, article);
      // Ici i vaut 10 et article.Prix vaut 100.
   }
   static void fct(int i, Article article) {
      // L'entier i n'est pas le même que l'entier i de Main().
      // L'instance de la classe Article passée est bien la même que
      // l'instance de la classe Article de la méthode Main.
      i = 100;
      article.Prix = 100;
   }
}
```

## *La possibilité de forcer le passage d'argument par référence*

C++ → C# En C#, on peut forcer le passage d'argument par référence, de même qu'en C++.

En C# la syntaxe a changé et corrige le fait qu'en C++, l'utilisateur n'a pas toujours conscience de passer par référence un argument. En effet, en C++, la syntaxe ne change pas à l'appel de la fonction mais seulement dans le prototype de la fonction (avec l'utilisation de « & »).

En C# ce problème n'existe plus. Lorsqu'un argument est passé par référence il faut utiliser le mot-clé `ref` , à la fois lors de la déclaration de la méthode, et lors de l'appel.

C# C# offre la possibilité de forcer le passage par référence d'un argument. Cette technique est utilisable pour les types valeur et référence. Cette technique permet de passer par référence des arguments de type valeur. Nous allons voir juste après que cette technique impacte aussi le passage des objets de type référence. Cette technique utilise le mot-clé `ref`, à la fois lors de la déclaration de la méthode, et lors de l'appel, par exemple :

*Exemple 11-7 :*

```
class Program {
   static void Main() {
      int i = 10 ;   // int est un type valeur.
      fct(ref i);
      // Ici i vaut 100.
   }
   static void fct( ref int i ) {
      // L'entier i est le même que l'entier i de Main.
      i = 100 ;
   }
}
```

## *Passer par référence un argument de type référence*

C++ → C# Nous présentons ici une technique de passage d'argument en C#, conceptuellement proche de l'utilisation d'un double pointeur en C++.

C# . Passer un argument de type référence par référence permet à la méthode appelée d'agir directement sur la référence. Concrètement, la méthode appelée peut modifier l'objet qui est référencé par la référence passée. Ceci est illustré dans l'exemple suivant :

*Exemple 11-8 :*

```
public class Article { public int Prix = 0 ; }
class Program {
   static void Main() {
      Article articleA = null ;
      Article articleB = null ;
      // Article est un type référence.
      fct(articleA, ref articleB) ;
      // Ici, articleA ne référénce aucun objet,
      // c'est toujours une référence nulle.
      // articleB référence le second objet alloué dans fct().
   }
   static void fct(Article articleA, ref Article articleB) {
      if (articleA == null)
         articleA = new Article();
      if (articleB == null)
         articleB = new Article();
   }
}
```

À l'instar de cet exemple, on utilise souvent cette technique pour déléguer l'allocation d'un ou plusieurs objets, dans une méthode. Dans ce cas, il est préférable d'utiliser la technique d'argument `out`, exposée un peu plus loin.

## *Initialisation des arguments*

C++ → C♯ Le compilateur C++ ne détecte pas l'utilisation de variables ou d'objets non initialisés explicitement. Ceci est un énorme problème puisque C++ n'initialise pas les variables et les objets implicitement. En C++, de nombreux bugs sont dus à une variable non initialisée.

De plus, certains compilateurs ne respectent pas la norme C++ (discutable il est vrai) et initialisent à zéro la mémoire allouée aux variables. Le code est ainsi peu portable puisque des compilateurs différents génèrent des comportements différents.

Le compilateur C♯ corrige ces problèmes en imposant que les arguments passés à une méthode soient initialisés.

C♯ C♯ oblige le développeur à initialiser ses objets de type valeur avant de les passer à une méthode. Cette règle s'applique, que l'argument soit passé par valeur ou par référence. Dans le cas d'un argument de type référence, soit la référence est sur un objet déjà construit, auquel cas C♯ a déjà forcé son initialisation, soit la référence n'est sur aucun objet, auquel cas elle doit avoir été initialisée avec le mot-clé `null`. Voici un exemple important, à bien assimiler :

*Exemple 11-9 :*

```
public class Article { public int Prix = 0 ; }
class Program {
   static void Main() {
      // Cette variable de type valeur doit etre initialisée
      // avant d'être passée à la méthode fct().
      int i = 10 ;
      // Ces deux références doivent être initialisées
      // avant d'être passées à la methode fct().
      Article articleA = null ;
      Article articleB = new Article() ;
      articleB.Prix = 100 ;
      fct(i, articleA, articleB) ;
      // Ici i vaut 10.
      // articleA ne référence aucun objet.
      // articleB.Prix vaut 10.
   }
   static void fct(int i, Article articleA, Article articleB) {
      if (articleA == null)
         articleA = new Article() ;
      articleA.Prix = i ;
      articleB.Prix = i ;
   }
}
```

## *Récupération d'information à partir d'une méthode (les paramètres out)*

C++ → C# En C++, pour récupérer le résultat d'une méthode il faut soit utiliser la valeur de retour, soit passer des arguments par référence ou par pointeur. Dans le deuxième cas, l'intention du développeur d'utiliser un argument pour récupérer un résultat, n'est pas évidente, puisque la même syntaxe permet aussi de passer des informations à la fonction. C# offre une syntaxe élégante pour pallier ce problème.

C# C# permet à une méthode de retourner des informations (par exemple les résultats d'un calcul) de deux façons différentes :

- Directement par le paramètre de retour de la méthode. Le problème de cette technique largement utilisée dans beaucoup de langages, est l'unicité du paramètre de retour. Le mot-clé `return` est utilisé dans le corps de la méthode pour définir le contenu du paramètre de retour. Notez qu'il peut y avoir plusieurs utilisations du mot-clé `return` au sein d'une méthode. Si la méthode n'a rien à retourner, elle doit être déclarée avec le type `void` comme type de retour. Dans ce cas il n'est pas obligatoire d'utiliser le mot-clé `return` dans le corps de la méthode.
- C# permet à un ou plusieurs paramètres de la méthode, d'être des paramètres de retour. Dans ce cas les paramètres sont signalés avec le mot-clé `out` dans le prototype de la méthode et lors des appels de la méthode. De plus les arguments n'ont pas besoin d'être initialisés. La méthode est obligée d'initialiser ces paramètres. Elle n'a pas le droit de se servir de ces paramètres tant qu'elle ne les a pas explicitement initialisés.

*Exemple 11-10 :*

```
public class Article { public int Prix = 0 ; }
class Program {
   static void Main() {
      int i ;  // i n'est pas initialisé.
      Article articleA ; // articleA n'est pas initialisé.
      Article articleB = new Article() ; // articleB référence un objet.
      articleB.Prix = 100 ;
      fct( out i, out articleA, out articleB ) ;
      // Ici i vaut 10. articleA.Prix vaut 10 et articleB référence
      // l'objet créé dans fct(). articleB.Prix vaut 10 mais articleB
      // ne référence plus le même objet qu'avant l'appel de fct().
   }
   static void fct(out int i, out Article a, out Article b) {
      i = 10 ;
      a = new Article() ;
      b = new Article() ;
      a.Prix = i ;
      b.Prix = i ;
   }
}
```

## *La possibilité d'avoir des arguments variables en nombre et en type*

C++ → C# C++ permet à une fonction d'avoir des arguments par défaut. C++ permet aussi à une fonction d'avoir un nombre d'arguments variable. Cette seconde possibilité est par exemple utilisée dans la fonction `printf()`.

C# ne permet pas d'avoir des arguments par défaut. Cependant C# permet à une fonction d'avoir un nombre d'arguments variable. Pour remplacer les arguments par défaut du C++ on peut aussi surcharger la méthode, mais cette technique est assez lourde.

C# C# permet à une méthode d'avoir un nombre quelconque d'arguments de même type ou non. La syntaxe de cette possibilité utilise le mot-clé `params`. L'argument qui utilise le mot-clé `params` doit être le dernier argument dans la liste des arguments d'une méthode. Notez que C# compile une telle méthode avec l'attribut `System.ParamArrayAttribute`. À l'exécution, le CLR comprend alors que la méthode présente cette possibilité. Voici un exemple d'utilisation :

*Exemple 11-11 :*

```
using System ;
class Program {
   static void Main() {
      fct("Appel1") ;
      fct("Appel2", 67, 3.1415, "hello", 8) ;
      fct("Appel3", "bonjour", 2.7, 1729, 691, "au revoir") ;
   }
   static void fct(string str, params object[] args) {
      Console.WriteLine(str) ;
      foreach (object obj in args) {
         if (obj is int) Console.WriteLine(" int:" + obj) ;
         else if (obj is double) Console.WriteLine(" double:" + obj) ;
         else if (obj is string) Console.WriteLine(" string:" + obj) ;
         else Console.WriteLine(" autre type:" + obj) ;
      }
   }
}
```

Cet exemple affiche :

```
Appel1
Appel2
 int:67
 double:3,1415
 string:hello
 int:8
Appel3
 string:bonjour
 double:2,7
 int:1729
 int:691
 string:au revoir
```

Cet exemple utilise l'opérateur `is` expliqué dans le prochain chapitre. Comme vous le voyez, cet opérateur permet de tester le type d'un objet. De plus nous spécifions que les arguments correspondant à l'argument qui utilise le mot-clé `params`, doivent être de type `object`. C'est grâce à cette astuce que nous pouvons faire varier le type des arguments, car tous les types héritent du type `object`.

## *Surcharge de méthodes*

C++ → C♯ La surcharge de méthodes existe en C♯, tout comme en C++.

C♯ C♯ permet à plusieurs méthodes d'une même classe d'avoir le même nom. Cette possibilité est appelée *surcharge* (*overloading* en anglais) du nom d'une méthode. Dans ce cas, les listes des arguments de ces méthodes doivent être différentes deux à deux. En effet, dans le cas contraire il y aurait ambiguïté lors de l'appel. L'argument de retour n'est pas pris en compte par le compilateur pour cette différenciation. C'est-à-dire que ce dernier génère une erreur si deux méthodes ont le même nom, la même liste de paramètres, mais un type de paramètre de retour différent. En revanche, si deux méthodes d'un même type ont le même nom et des listes de paramètres différentes, elles peuvent avoir un type de retour différent. Il est néanmoins fortement déconseillé que les différentes surcharges aient des types de retour différents car ce dernier est en général fortement lié à la sémantique associée au nom commun des surcharges.

D'autres ambiguïtés peuvent apparaître si certaines de ces méthodes acceptent des arguments variables en type et en nombre. Dans ce cas le compilateur C♯ lève l'ambiguïté en préférant les méthodes sans arguments variables en type et en nombre. Cependant ce type d'ambiguïté est à éviter afin de garder un code lisible. Par exemple :

*Exemple 11-12 :*

```
using System ;
class Program {
   static void Main() {
      fct("Appel1", "bonjour") ;  // Appel de la surcharge 3.
      fct("Appel2") ;             // Appel de la surcharge 1.
      fct("Appel3", 10) ;         // Appel de la surcharge 2.
      fct("Appel4", 10, 11) ;     // Appel de la surcharge 4.
      fct("Appel5", 10.1) ;       // Appel de la surcharge 3.
   }
   // Surcharge 1 :
   static void fct(string str) {
      System.Console.WriteLine("surcharge 1") ;
   }
   // Surcharge 2 :
   static void fct(string str, int i) {
      System.Console.WriteLine("surcharge 2") ;
   }
   // Surcharge 3 :
   static void fct(string str, params object[] Args) {
      System.Console.WriteLine("surcharge 3") ;
   }
   // Surcharge 4 :
```

```
   static void fct(string str, params int[] Args) {
      System.Console.WriteLine("surcharge 4") ;
   }
}
```

En page 477 nous expliquons le type d'ambiguïté qui peuvent apparaître à cause des paramètres génériques et comment le compilateur C♯2 les résout.

# *Les propriétés*

C++ → C♯ Le concept de propriété n'existe pas en C++.

C♯ C♯ permet aux classes et aux structures d'avoir des propriétés. Les propriétés permettent d'accéder à l'état d'une instance avec la même syntaxe que l'accès à un champ, mais sans passer directement par un champ. L'accès se fait par des méthodes spéciales appelées les *accesseurs* de la propriété. L'avantage est double :

- L'utilisateur de la classe veut un accès à l'état de l'objet avec la même syntaxe que l'accès à un champ. Il n'a pas à appeler explicitement une méthode.
- Le développeur de la classe veut intercepter tous les accès à l'état de l'objet. Pour cela il utilise les accesseurs.

Il y a deux types d'accès à l'état d'un objet. Aussi, il y a deux accesseurs possibles pour chaque propriété :

- À chaque accès en lecture d'une propriété l'accesseur `get` de la propriété est appelé.
- À chaque accès en écriture d'une propriété l'accesseur `set` de la propriété est appelé.

Par exemple :

*Exemple 11-13 :*

```
public class Foo {
   private int m_ChampPrivate = 10 ;
   public int Prop {  // Une propriété de type int.
      get{ return m_ChampPrivate;}
      set{ m_ChampPrivate = value;}
   }
}
public class Program {
   static void Main() {
      Foo foo = new Foo() ;
      foo.Prop = 56 ;     // L'accesseur set est appelé.
      int i = foo.Prop ;  // L'accesseur get est appelé.
   }
}
```

Les accesseurs se codent comme des méthodes mis à part que :

- Ils ne peuvent être appelés explicitement comme des méthodes.

- Dans leurs déclarations, ils n'ont pas d'arguments ni d'entrée ni de retour.

Une propriété peut être de n'importe quel type (valeur ou référence).

À l'instar de cet exemple, il y a en général pour chaque propriété publique un champ privé qui lui correspond et vice versa.

## *Accesseur get*

L'accesseur `get` d'une propriété a pour contrainte de retourner un objet de même type que sa propriété (ou une référence vers un objet si c'est un type référence).

Une propriété n'est pas obligée d'implémenter l'accesseur `set`. Dans ce cas on dit qu'elle est accessible en lecture seule (*read only* en anglais, bien que l'on ne parle pas ici du mot-clé `readonly`). Une propriété accessible en lecture seule ne peut pas être initialisée. En revanche, on peut retourner un objet dont l'état est calculé dans l'accesseur `get`.

*Exemple 11-14 :*

```
public class Foo {
   private int m_ChampPrivate = 10 ;
   public bool Prop { // Une propriété de type bool en lecture seule.
      get{ return (m_ChampPrivate >100);}
   }
}
public class Program {
   static void Main() {
      Foo foo = new Foo() ;
      bool b = foo.Prop ;  // L'accesseur get est appelé.
   }
}
```

## *Accesseur set*

Dans le corps de l'accesseur `set`, le mot-clé `value` est utilisable. Il représente un objet du même type que la propriété (ou une référence si c'est un type référence). C'est l'objet (ou la référence) fourni lors de l'assignation de la propriété.

Une propriété n'est pas obligée d'implémenter l'accesseur `get`. Dans ce cas on dit qu'elle est accessible en écriture seule. Une propriété accessible en écriture seule est inaccessible en lecture. En revanche, on peut calculer un nouvel état pour l'objet qui présente la propriété, en tenant compte de la valeur passée.

*Exemple 11-15 :*

```
public class Foo {
   private int m_ChampPrivate = 10 ;
   public int Prop { // Une propriété de type int en écriture seule.
      set{ m_ChampPrivate = value*2;}
   }
}
public class Program {
```

```
   static void Main() {
      Foo foo = new Foo() ;
      foo.Prop = 56 ;  // L'accesseur set est appelé.
   }
}
```

## *Remarques*

Une propriété est obligée d'avoir au moins un accesseur. Elle a soit un accesseur `get`, soit un accesseur `set`, soit les deux.

Il est conseillé de ne pas modifier l'état d'un objet dans les accesseurs `get`. Il faut considérer les accesseurs `get` comme un moyen d'intercepter et de contrôler un accès à l'état de l'objet. On peut par exemple en profiter pour loguer ces accès.

Il est conseillé de ne pas lancer d'exception à partir d'un accesseur, à moins quelle soit rattrapée dans l'accesseur même. En effet, syntaxiquement le client n'a pas nécessairement conscience d'appeler un accesseur lors de l'accès à une propriété. Le client peut donc être surpris par une exception.

En page 419, nous expliquons que les accesseurs d'une même propriété peuvent avoir des niveaux de visibilité différents.

# *Les indexeurs*

C++ → C♯ Comme le langage C++, le langage C♯ permet de définir l'opérateur d'accès à un tableau « [ ] » dans une classe. Cependant C♯ permet une définition beaucoup plus complète de cet opérateur. En effet, il peut y avoir plusieurs index, on peut utiliser n'importe quel type pour les index (y compris vos propres classes) et il peut y avoir plusieurs définitions de cet opérateur pour une même classe.

C♯ C♯ permet de considérer un objet comme un tableau à une ou plusieurs dimensions. En effet, C♯ autorise l'utilisation de l'opérateur [ ] directement après le nom d'un objet. Cet opérateur accepte une liste de un ou plusieurs paramètres, de n'importe quel type (entier, chaîne de caractères, objet de toutes classes etc). Les paramètres doivent être séparés par des virgules.

Bien évidemment, la classe de l'objet doit prévoir le fait que l'opérateur [ ] puisse être utilisé directement après le nom d'un objet. Pour cela elle déclare un ou plusieurs *indexeurs*. Les indexeurs des classes peuvent être considérés comme des propriétés particulières, aux différences suivantes près :

| **Propriété** | **Indexeur** |
|---|---|
| Identifiée par son nom. | Identifié par sa signature. |
| Accédée à partir de son nom, comme un champ. | Accédé avec l'opérateur d'accès aux éléments [ ]. |
| Peut être statique ou non. | Ne peut pas être statique. |
| L'accesseur `get` n'a pas de paramètres. | L'accesseur `get` a la liste des paramètres de l'indexeur. |

| L'accesseur `set` peut utiliser le paramètre `value` implicite. | L'accesseur `set` a la liste des paramètres de l'indexeur en plus du paramètre `value` implicite. |
|---|---|

Comme pour les propriétés, les accesseurs des indexeurs sont facultatifs, bien qu'il en faille au moins un par indexeur. La définition des accesseurs autorise un accès en lecture seule, écriture seule ou lecture/écriture. Les paramètres des indexeurs ne peuvent utiliser les mots-clés `ref` et `out`.

L'exemple suivant expose la syntaxe des indexeurs. Nous présentons l'accès à un tableau de personnes. Les personnes peuvent être accédées par leurs noms (une chaîne de caractères) en lecture seulement. Les personnes peuvent aussi être accédées par leurs index dans le tableau interne à l'objet qui les contient, en lecture/écriture.

*Exemple 11-16 :*

```
using System ;
public class Personnes {
   // Tableau privé interne qui contient les noms des personnes.
   string [] m_Noms ;
   // Le constructeur qui initialise le tableau.
   public Personnes(params string [] noms){
      m_Noms = new string[noms.Length] ;
      // Copie le tableau.
      noms.CopyTo(m_Noms,0) ;
   }
   // L'indexeur qui retourne l'index à partir du nom.
   public int this[string nom]{
      get{ return Array.IndexOf(m_Noms,nom);}
   }
   // L'indexeur qui retourne le nom à partir de l'index.
   public string this[int index]{
      get{ return m_Noms[index];}
      set{ m_Noms[index] = value;}
   }
}
class Program {
   static void Main() {
      Personnes tableau = new Personnes (
         "Anna" , "Ingrid" , "Maria" , "Ulrika" ) ;
      Console.WriteLine(tableau [1]) ; // Affiche "Ingrid"
      int index = tableau["Maria"] ;
      tableau[index] = "Marie" ;
      Console.WriteLine(tableau[index]) ; // Affiche "Marie"
   }
}
```

En général les classes qui ont des indexeurs doivent présenter la possibilité d'être utilisées avec la syntaxe avec la syntaxe des mots-clés `foreach` et `in`.

# *Les événements*

## *Introduction*

C++ → C♯ C♯ définit la notion d'événement qui est complètement absente en C++. En C++ il faut implémenter soi-même la plomberie du mécanisme d'événement, par exemple au moyen du design pattern (Gof) « observateur ».

C♯ C♯ permet a un objet de présenter des *événements*. Le concept d'événement rassemble :

- Des *abonnés* à l'événement. Ces derniers sont prévenus chaque fois que l'événement est déclenché. Ils ont la possibilité de s'abonner et de se désabonner à l'événement dynamiquement (i.e durant l'exécution du programme). En C♯, un abonné est représenté par une méthode.
- L'événement lui-même, qui peut à tout moment être déclenché par l'objet. En interne, l'événement a la connaissance de ses abonnés. La responsabilité de l'événement est de prévenir ses abonnés lorsqu'il est déclenché. L'événement a la possibilité de fournir des informations à ses abonnés, lorsqu'il est déclenché avec l'objet argument de l'événement. En C♯, un événement est semblable à un délégué membre d'une classe ou d'une structure, mis à part qu'il est déclaré avec le mot clé `event`.

Le concept événement/abonnés n'est pas nouveau et est connu sous le nom de publisher/subscriber (éditeur/souscripteur) en Java. Le même comportement est obtenu avec l'utilisation du *design pattern* (Gof) « observateur » (sujet/observateur).

Les événements sont très utilisés dans les applications avec une interface graphique (*Windows Forms* et ASP.NET). En effet, chaque action possible sur l'interface (clic sur un bouton, texte tapé, ComboBox déroulée, etc) déclenche l'appel à la méthode adéquate. Lors d'un clic souris, l'argument de l'événement peut être par exemple la position de la souris sur l'écran. Dans la plateforme .NET, l'héritage combiné avec les événements sont à la base de tous les contrôles graphiques.

## *La syntaxe C♯*

Supposons que nous ayons créé dans une classe un événement qui s'appelle *`EventName`*. Les noms des entités concernées par l'événement sont fonction du nom *`EventName`*. Un argument d'événement est un objet d'une classe dérivée de la classe `System.EventArgs`. Vous avez donc la possibilité de créer vos propres types d'arguments d'événement en créant vos propres classes dérivées de `System.EventArgs`.

Pour améliorer la lisibilité du code, il est préférable que la classe d'argument de l'événement *`EventName`* s'appelle *`EventName`*`EventArgs`.

Les abonnés sont matérialisés par des méthodes (statiques ou non). Le fait de prévenir les abonnés se traduit par un appel à chacune de ces méthodes. Ces méthodes ont en général la signature suivante :

- Retour du type `void`.
- Premier argument de type `objet`. Lorsque la méthode est appelée, cet argument référence l'objet qui contient l'événement.

- Deuxième argument de type `System.EventArgs`. Lorsque la méthode est appelée cet argument référence l'argument de l'événement.

Un événement est un membre d'une classe, déclarée avec le mot-clé `event`. Un événement est une instance de la délégation *`EventName`*`EventHandler`. Un événement a la possibilité d'être statique ou non. Les niveaux de visibilité s'appliquent aux événements. Comme un événement est un membre d'une classe, la déclaration d'un événement peut être précédée par un ou plusieurs des mots-clés suivants :

```
new        public      protected   internal   private   static
virtual    sealed      override    abstract    extern
```

Un événement contient :

- Un délégué qui renferme les références vers les méthodes abonnées. La délégation a la même signature que les méthodes abonnées et doit s'appeler *`EventName`*`EventHandler`.
- Un accesseur `add` qui est invoqué lorsqu'une méthode est abonnée à l'événement.
- Un accesseur `remove` qui est invoqué lorsqu'une méthode est désabonnée à l'événement.

Ces trois entités sont définies automatiquement par le compilateur C# dès que l'événement est déclaré. Vous pouvez néanmoins définir vos propres corps d'accesseurs. Si vous changez le corps d'un accesseur, vous devez aussi coder le corps de l'autre accesseur. Dans le corps d'un accesseur, le mot-clé `value` désigne le délégué à ajouter ou à supprimer. Par exemple les deux définitions suivantes d'événement sont acceptables :

```
// sans accesseurs
public event ClickButtonEventHandler ClickButton ;
// avec accesseurs
private event ClickButtonEventHandler m_ClickButton ;
public event ClickButtonEventHandler ClickButton{
   add   { m_ClickButton += value ; }
   remove   { m_ClickButton -= value ; }
}
```

Notez qu'il est très rare d'avoir à modifier les accesseurs `add` et `remove`. Cependant, cette facilité peut être utile afin de restreindre le niveau de visibilité de l'événement. Comprenez que les possibilités d'exploiter le polymorphisme sur un événement (i.e le fait de pouvoir utiliser les mots-clés `virtual`, `new`, `override` et `abstract` dans la déclaration d'un événement) ne s'applique qu'à ces deux accesseurs `add` et `remove`. Clairement, cette facilité et séduisante pour le puriste mais extrêmement peu usitée.

En C# l'événement est déclenché par l'appel à son délégué. Ce n'est pas le cas d'autres langages .NET. Par exemple VB.NET déclenche un événement au moyen du mot-clé `RaiseEvent` qui n'existe pas en C#. Le compilateur impose que l'appel au délégué doit être réalisé au sein de la classe qui déclare l'événement. Aussi, cette classe a la possibilité de présenter une méthode publique `On`*`EventName`*`()` qui permet de déclencher l'événement hors de la classe.

```
...
public void OnClickButton(System.EventArgs Arg){
   if(ClickButton != null )
      // Déclenche l'événement.
```

```
      ClickButton(this,Arg);
}
...
```

## *Un exemple*

Voici un exemple où un objet de la classe `AgenceDePresse` publie des bulletins d'information sur la France et le monde. La publication se fait à l'aide des deux événements `InfoMonde` et `InfoFrance`. L'argument d'un événement bulletin d'information (instance de la classe `InfoEventArgs`) contient la description du bulletin d'information. Enfin, les instances de la classe `Abonne` ont la possibilité de recevoir un bulletin d'information en abonnant leur méthode `ReceptionInfo()`.

*Exemple 11-17 :*

```
using System ;

class Abonne{
     private string m_Nom ;
     public Abonne( string Nom ) { m_Nom = Nom ; }
     // Méthode à appeler lorsqu'un événement est déclenché
     // (i.e lorsqu'un bulletin d'information est publié).
     public void ReceptionInfo(object sender, EventArgs e){
        InfoEventArgs info = e as InfoEventArgs ;
        if( info != null ){
           Console.WriteLine( m_Nom + " reçoit l'info : " +
                              ((InfoEventArgs)e).GetBulletinInfo() ) ;
     }
   }
}

//La classe d'argument de l'événement bulletin d'information.
class InfoEventArgs: EventArgs{
     private string m_Description ;
     public string GetBulletinInfo() { return m_Description ; }
     public InfoEventArgs (string Description) {
        m_Description = Description ;
     }
}

// Définition du type délégué 'handler d'un bulletin d'information'.
public delegate void InfoEventHandler(object sender, EventArgs e) ;

// Classe contenant les événements bulletin d'information.
class AgenceDePresse {
   // Définition des événements bulletins d'information.
   public event InfoEventHandler InfoFrance;
   public event InfoEventHandler InfoMonde;
   // Méthodes de déclenchement des événements
```

```
   // (i.e de publication de bulletin d'information).
   public void OnInfoFrance(InfoEventArgs bulletinInfo) {
      if( InfoFrance != null )
         InfoFrance(this,bulletinInfo) ;
   }
   public void OnInfoMonde(InfoEventArgs bulletinInfo) {
      if( InfoMonde != null )
         InfoMonde(this,bulletinInfo) ;
   }
}

class Program {
   public static void Main(){
      // Création de l'agence de presse.
      AgenceDePresse afp = new AgenceDePresse () ;
      // Création des abonnés.
      Abonne raymond = new Abonne("Raymond") ;
      Abonne olivier = new Abonne("Olivier") ;
      Abonne mathieu = new Abonne("Mathieu") ;
      // Création des abonnements aux événements bulletins d'info.
      afp.InfoFrance += raymond.ReceptionInfo ;
      afp.InfoFrance += olivier.ReceptionInfo ;
      afp.InfoMonde  += olivier.ReceptionInfo ;
      afp.InfoMonde  += mathieu.ReceptionInfo ;
      // Publication de bulletins d'information
      // (déclenchement des événement).
      afp.OnInfoFrance(new InfoEventArgs("Hausse du prix du tabac.")) ;
      afp.OnInfoMonde(new InfoEventArgs(
         "Nouvelle élection au États-Unis.")) ;
      // Résilation d'abonnement.
      afp.InfoFrance -= new InfoEventHandler(olivier.ReceptionInfo ) ;
      // Publication de bulletins d'information.
      afp.OnInfoFrance(new InfoEventArgs("Baisse des impôts.")) ;
   }
}
```

Le programme affiche :

```
Raymond reçoit l'info : Hausse du prix du tabac.
Olivier reçoit l'info : Hausse du prix du tabac.
Olivier reçoit l'info : Nouvelle élection au Etats-Unis.
Mathieu reçoit l'info : Nouvelle élection au Etats-Unis.
Raymond reçoit l'info : Baisse des impôts.
```

Dans cet exemple, rien ne change si vous enlevez les deux occurrences du mot-clé `event` (i.e rien ne change si nous utilisons deux délégués au lieu d'utiliser deux événements). Il y a deux raisons à cela. La première est que nous n'utilisons pas d'accesseurs d'événements. La seconde est que nous ne déclenchons pas un de ces événements hors de la classe `AgenceDePresse`. En effet, une utilité du mot-clé `event` est qu'il empêche l'événement d'être déclenché à l'extérieur de sa classe. Le déclenchement d'un événement est aussi interdit à partir des méthodes des classes dérivées

de la classe définissant l'événement, y compris si l'événement est virtuel. Ainsi le compilateur C# produirait une erreur si nous avions écrit la ligne suivante dans le corps de la méthode `main()` de l'Exemple 11-17 :

*Exemple 11-18 :*

```
public static void Main(){
   ...
      afp.InfoFrance(
         afp, new InfoEventArgs("Hausse du prix du tabac.")) ;
   ...
   }
```

En C# la notion d'événement est donc relativement proche de celle de délégué. Le mécanisme permet surtout de fournir un cadre standard à la gestion des événements. Notez que le langage VB.NET présente plusieurs mots-clés spécialement conçus pour la gestion des événements (`RaiseEvent`, `WithEvents` etc).

## *Événements asynchrones*

Lorsqu'un événement est déclenché, il faut bien être conscient que c'est le même thread qui exécute l'instruction de déclenchement et les méthodes abonnées à l'événement.

Un problème potentiel est que certaines méthodes abonnées s'exécutent en un temps inacceptable. Un autre problème potentiel est qu'une méthode abonnée peut lever une exception. Cette dernière empêchera alors les exécutions des autres méthodes abonnées non encore exécutées et remontera jusqu'à la méthode qui a déclenché l'événement. Il est clair que cela constitue un problème puisque la méthode qui déclenche un événement doit absolument être totalement découplée des méthodes abonnées.

Pour pallier ces problèmes, il serait bien pratique que les méthodes abonnées soient exécutées d'une manière asynchrone (la notion d'appel de méthode asynchrone est décrite page 171). En outre, il serait aussi intéressant que différents threads exécutent les différentes méthodes abonnées. Ainsi, une méthode abonnée qui s'exécute en un temps inacceptable ou qui lance une exception ne gène ni l'exécution du déclencheur de l'événement, ni les exécutions des autres méthodes abonnées. Le pool de threads du CLR qui est décrit en page 167 est particulièrement adapté à ces deux contraintes.

Ceux qui ont compris la syntaxe d'un appel asynchrone de méthode risquent d'être tentés d'utiliser directement la fonction `BeginInvoke()` sur l'événement lui-même. Le problème de cette solution est que seule une méthode abonnée sera exécutée. La bonne solution, illustrée par l'exemple suivant qui est basée sur l'Exemple 11-17 nécessite le parcours explicite de la liste des méthodes abonnées :

*Exemple 11-19 :*

```
...
   // Méthodes de déclenchement des événements
   // (i.e de publication de bulletin d'information).
   public void OnInfoFrance(InfoEventArgs BulletinInfo){
      if( InfoFrance != null ){
         Delegate[] abonnes = InfoFrance.GetInvocationList() ;
```

```
            foreach(Delegate _abonne in abonnes ){
               InfoEventHandler abonne = (InfoEventHandler) _abonne ;
               abonne.BeginInvoke(this,BulletinInfo,null,null) ;
            }
            // Attend un peu pour permettre les exécutions asynchrones
            // celles ci se faisant sur des threads du pool,
            // qui sont de threads background. Sans cet artifice,
            // les threads du pool n'auraient pas le temps de commencer
            // leur travail que le programme serait déjà fini.
            System.Threading.Thread.Sleep(100) ;
         }
      }
      ...
```

Notez enfin qu'il n'y a pas lieu d'appeler la méthode `EndInvoke()` puisque les méthodes abonnées ne sont pas sensées retourner de résultat à la méthode qui a déclenché l'événement.

### *Se protéger des exceptions lancées par les méthodes abonnées dans le cas synchrone*

Invoquer les méthodes abonnées à un événement d'une manière asynchrone est une technique efficace pour protéger le code qui déclenche un événement des exceptions remontées par les méthodes abonnées. Il est cependant possible d'avoir le même niveau de protection en invoquant les méthodes d'une manière synchrone. Pour cela, il faut invoquer les différentes méthodes abonnées une à une, comme le montre l'exemple suivant :

*Exemple 11-20 :*

```
...
   // Méthodes de déclenchement des événements
   // (i.e de publication de bulletin d'information).
   public void OnInfoFrance(InfoEventArgs BulletinInfo){
      if( InfoFrance != null ){
         Delegate[] abonnes = InfoFrance.GetInvocationList() ;
         foreach(Delegate _abonne in abonnes ){
            InfoEventHandler abonne = (InfoEventHandler) _abonne ;
            try{
               abonne(this,BulletinInfo) ;
            }
            catch(Exception){ /*traitement d'exception*/ }
         }
      }
   }
   ...
```

## *Les types encapsulés*

On peut définir un type à l'intérieur d'un autre type. Dans ce cas le type défini à l'intérieur est appelé *type encapsulé* (*nested type* en anglais). Il y a au moins deux avantages à définir un type dans un autre :

- On peut restreindre la visibilité du type encapsulé. C'est-à-dire qu'un type défini dans une classe ne pourra être utilisé partout dans le code, mais seulement là où vous l'avez décidé (en général, seulement à l'intérieur de la classe encapsulante).
- On peut accéder à tous les membres de la classe encapsulante, à partir des méthodes du type encapsulé. Voici un exemple pour illustrer cette possibilité qui n'est pas triviale. Les instances de `TypeEncapsule` ont un libre accès au champ privé `m_i` d'une instance de `TypeEncapsulant` :

*Exemple 11-21 :*

```
public class TypeEncapsulant {
   private int m_i = 10 ;
   public class TypeEncapsule {
      public void Add(TypeEncapsulant Foo, int i) {
         Foo.m_i += i ;   // Accès au champ privé TypeEncapsulant.m_i.
      }
   }
}
public class Program {
   static void Main() {
      TypeEncapsulant foo1 = new TypeEncapsulant() ;
      TypeEncapsulant.TypeEncapsule foo2 = new
                           TypeEncapsulant.TypeEncapsule() ;
      foo2.Add(foo1, 3) ;
      // Ici foo1.m_i vaut 13.
   }
}
```

Notez qu'à l'extérieur de la définition de la classe `TypeEncapsulant`, la classe `TypeEncapsule` s'appelle `TypeEncapsulant.TypeEncapsule`.

# *Encapsulation et niveaux de visibilité*

## *Les niveaux de visibilité des membres*

C++ → C♯ Deux nouveaux niveaux de visibilité sont introduits avec C♯ : interne (`internal`) et interne protégé (`internal protected`). Ces nouveaux niveaux de visibilité ne concernent que les membres des classes et pas les membres des structures. Leur signification est liée à la notion d'assemblage.

Une autre différence avec le C++ est au niveau syntaxique. Les mots-clés de visibilité (`private`, `protected public` ou `internal`) d'un membre doivent être écrits devant chaque membre.

Si aucun de ces mots-clés n'est précisé, le membre est déclaré `private` par défaut, dans une classe mais aussi dans une structure.

La possibilité du C++ de dériver d'une classe de base avec les mots-clés `private protected` ou `public` afin de conserver ou de restreindre les niveaux de visibilité des membres de la classe

de base dans la classe dérivée, n'existe pas en C#. En C# on ne peut que conserver le niveau de visibilité des membres de la classe de base.

C# L'*encapsulation* est un puissant concept de la programmation par objets qui permet de maîtriser la visibilité que l'on a des membres d'une classe ou d'une structure vus de l'extérieur de celle-ci.

L'utilisateur d'une classe (c'est-à-dire un développeur qui écrit du code qui instancie ou référence une instance de la classe), n'a donc accès qu'à certains membres. Cela réduit d'autant la complexité d'utilisation des objets puisque le nombre de points d'accès est réduit.

Pour bien comprendre les différents *niveaux de visibilité* il faut avoir compris la notion d'assemblage qui fait l'objet du chapitre 2 et la notion de classe dérivée, qui fait l'objet du chapitre 12.

Il y a cinq niveaux de visibilité différents. Chaque niveau de visibilité porte sa restriction à une partie du code source extérieur de la classe. Seul le niveau de visibilité public n'induit aucune restriction. Chaque membre d'une classe a un niveau de visibilité (et un seul) parmi :

- Le niveau de visibilité *public* : mot-clé `public`.

  Un membre qui a le niveau de visibilité *public* est accessible partout dans le code de l'assemblage courant et dans le code des assemblages référençant l'assemblage courant.

- Le niveau de visibilité *privé* : mot-clé `private`.

  Un membre qui a le niveau de visibilité *privé* et qui est défini dans un type `T` est accessible seulement à partir du code contenu dans `T` ainsi qu'à partir du code des types encapsulés de `T`.

- Le niveau de visibilité *protégé* : mot-clé `protected`.

  Le membre peut être accédé seulement dans le code des méthodes de la classe et des méthodes des classes dérivées. Ceci reste valable pour le code des méthodes des classes dérivées définies dans d'autres assemblages.

- Le niveau de visibilité *interne* : mot-clé `internal`.

  Le membre peut être accédé partout dans le code. Cependant, si la classe est utilisée dans d'autres assemblages le membre qualifié d'interne est invisible dans ces autres assemblages.

- Le niveau de visibilité *interne protégé* : mot-clé `internal protected`.

  Le membre peut être accédé partout dans le code de l'assemblage courant, à partir d'une instance de la classe. Cependant, si la classe est utilisée dans d'autres assemblages le membre qualifié d'interne protégé n'est visible que dans le code des méthodes des classes dérivant de la classe concernée. Notez que c'est le seul cas où deux mots-clés peuvent être utilisés pour déclarer un niveau de visibilité.

Résumons tout ceci dans un tableau. En colonne les niveaux de visibilité d'un membre d'une classe `A`. En ligne les zones de code. Un OK signifie que la zone de code concernée a accès à ce membre.

| | public | internal protected | internal | protected | private |
|---|---|---|---|---|---|
| Méthodes de la classe A. | OK | OK | OK | OK | OK |
| Méthodes d'une classe dérivée de A, dans le même assemblage. | OK | OK | OK | OK | |
| Méthodes d'une autre classe dans le même assemblage. | OK | OK | OK | | |
| Méthodes d'une classe dérivée de A dans un assemblage différent. | OK | OK | | OK | |
| Méthodes d'une autre classe dans un assemblage différent. | OK | | | | |

Les niveaux de visibilité interne, et interne protégé, ne peuvent s'appliquer aux membres d'une structure.

Un événement ne peut être déclenché hors de son type, quelque soit son niveau de visibilité. Même le code d'un type encapsulé de son propre type ne peut déclencher un événement. Seuls les accesseurs de l'événement sont impactés par son niveau de visibilité.

Par défaut (i.e si on ne spécifie pas explicitement un niveau de visibilité) les membres d'une classe et d'une structure sont privés.

## *Visibilité des types*

Un type qui n'est pas encapsulé dans un autre type ne peut avoir que les niveaux de visibilité interne et publique (ATTENTION, on parle du niveau de visibilité du type et pas du niveau de visibilité des membres du type). Si aucun niveau de visibilité n'est affecté explicitement à un type, le niveau interne est choisi par défaut. En page 27 nous introduisons la notion d'*assemblage amis*. Les assemblages amis d'un assemblage ont la possibilité d'accéder à ses types non publics.

## *Visibilité des accesseurs d'une propriété ou d'un indexeur*

C#2 permet aux accesseurs d'une propriété ou d'un indexeur d'avoir un niveau de visibilité différent. Pour exploiter cette possibilité il suffit d'assigner un niveau de visibilité à un des deux accesseurs. Ce niveau de visibilité doit être différent et plus restrictif que celui de la propriété ou de l'indexeur qui définit l'accesseur. L'autre accesseur aura automatiquement le niveau de visibilité de la propriété ou de l'indexeur qui le défini. En général, cette possibilité est utilisée pour faire en sorte que le niveau de visibilité de l'accesseur `set` soit plus restrictif que celui de l'accesseur `get` :

*Exemple 11-22 :*

```
class Foo {
   public int Prop {
      get { return 5 ; }
```

```
      private set { }
   }
   protected string this[int index] {
      private get { return "hello" ; }
      set {  }
   }
}
```

Cette possibilité n'est pas exploitable sur les accesseurs `add` et `remove` d'un événement.

### *Remarque sur les membres privés*

Puisque les membres privés d'une classe `A` ne sont accessibles qu'au sein de la classe, cela signifie qu'une instance `I1` de la classe `A` peut accéder aux membres privés d'une instance `I2` de la classe `A`. Soyez conscient de cette possibilité. En effet ceci n'a rien de logique et découle d'un choix arbitraire fait lors de la spécification du langage C#. Ce choix n'est pas forcément le même dans d'autres langages orientés objet tel que *Smalltalk* par exemple.

### *Remarque sur les niveaux de visibilité du langage IL*

Tout programme écrit en langage C# se compile en un programme écrit en langage IL. Il se trouve que le langage IL présente un sixième niveau de visibilité qui n'est pas supporté par C#. Ce niveau de visibilité est *interne ET protégé*. Pour saisir son effet, imaginez que le mot clé C# `internal protected` se compile en un niveau de visibilité *interne OU protégé*. Si nous l'avions représenté dans le tableau précédent, seul les deux premières lignes auraient été cochées pour la colonne de ce sixième niveau de visibilité.

## *Le mot-clé this*

C++ → C# Comme en C++, le mot-clé `this` existe en C#. En C#, ce mot-clé, utilisable dans les méthodes non statiques, est une référence (et non un pointeur comme en C++) vers l'objet courant.

C# Dans toutes les méthodes non statiques, de toutes les classes, C# permet d'utiliser le mot-clé `this`. Ce mot-clé est une référence vers l'objet sur lequel opère la méthode courante. Le mot-clé `this` ne peut donc être utilisé dans les méthodes statiques.

En C#, le mot-clé `this` permet à des arguments d'une méthode d'instance et à des champs (ou propriétés) d'instances de la classe d'avoir des noms similaires. Ainsi, dans le corps de la méthode, il faut faire précéder le nom en question par `this` pour désigner la variable globale au lieu de la variable locale (i.e le champ au lieu de l'argument). Par exemple :

*Exemple 11-23 :*

```
class Foo {
   string str ; // Champ non statique de la classe.
   void fct( string str ){  // Méthode non statique.
      this.str = str ;  // Le champ str est égale à l'argument str.
   }
}
```

Une autre utilité du mot-clé `this` est de communiquer une référence de l'objet courant à d'autres méthodes, éventuellement d'autres classes. Ce type d'utilisation témoigne en général d'une architecture objet évoluée. Par exemple, le *design pattern* (Gof) nommé communément « visiteur » utilise cette facilité :

*Exemple 11-24 :*

```
class Foo {
   void fct(){  // Méthode non statique.
      fct2(this);
   }
   static void fct2( Foo f ){
      // Travaille avec l'objet référencé par f...
   }
}
```

En outre, le mot-clé `this` est utilisé par la syntaxe des indexeurs décrite page 409. Le mot-clé `this` est aussi utilisable au sein des méthodes non statiques des structures. Bien qu'une structure soit un type valeur, le mot-clé `this` représente bien l'instance courante. Ceci est bien une facilité C# car il n'y a pas d'opération de boxing réalisée et donc pas de référence.

Pour ceux qui sont intéressés par le langage IL, sachez qu'en IL, la référence `this` est passée comme l'argument zéro à une méthode d'instance, décalant ainsi d'une position dans la liste les autres arguments.

# *Construction des objets*

## *Déclaration des constructeurs*

C++ → C# Le langage C# n'a pas la syntaxe C++ d'initialisation de champs, qui permet d'accomplir cette tâche immédiatement après le prototype du constructeur (`ctor(int Champs) : m_Champs(Champs) {}`).

Le compilateur C# est moins permissif et impose que tous les champs de type valeur soient initialisés avant la fin de l'exécution d'un constructeur.

Nous verrons que la syntaxe de passage d'arguments au constructeur d'une classe de base, à partir d'un constructeur d'une classe dérivée, est différente par rapport à celle du C++.

Nous verrons qu'il peut exister en C# un constructeur de classes, appelé lors du chargement de la classe dans un domaine d'application. C'est une possibilité qui n'existe pas en C++.

La facilité de constructeur de transtypage du C++ n'existe pas en C#. On peut évidemment construire un objet de classe A à partir d'un autre objet de classe B (en définissant un constructeur de la classe A qui prend un objet de B en paramètre) mais la syntaxe de la création de A par constructeur de transtypage :

explicite `A objA = (A)objB;`

ou implicite `A objA = objB;`

de C++ n'existe pas en C#. Il faut écrire :

`A objA=new A(objB).`

Toutefois la surcharge des opérateurs de transtypage est possible en C#, ce qui rend quand même les exemples possibles.

C# Une méthode est automatiquement appelée lorsque l'objet est construit. On appelle ce type de méthode un *constructeur* (*ctor* en abrégé). En C#, syntaxiquement, une méthode constructeur porte le nom de la classe, et ne retourne rien (même pas le type `void`). Il peut y avoir :

- Aucun constructeur : dans ce cas le compilateur fournit automatiquement un *constructeur par défaut*, qui n'accepte aucun argument. Un tel constructeur est public.
- Un seul constructeur : dans ce cas, ce sera toujours lui qui sera appelé. Le compilateur ne fournit pas de constructeur par défaut.
- Plusieurs constructeurs : dans ce cas ils diffèrent selon leurs signatures, le constructeur est alors une méthode surchargée. Le compilateur ne fournit pas de constructeur par défaut.

Pour résumer, à partir du moment où il y a au moins un constructeur, le compilateur ne fournit plus de constructeur par défaut dans le cas d'une classe. Ceci est illustré dans les exemples de la prochaine section.

Dans le cas d'une structure c'est différent puisqu'il est interdit de définir un constructeur sans argument. Le constructeur par défaut du compilateur est donc toujours accessible. Ceci est illustré dans les exemples de la prochaine section.

Lorsqu'un constructeur retourne, il est impératif que tous les champs non statiques de type valeur de la classe aient été initialisés d'une manière ou d'une autre.

Les constructeurs se plient aux mêmes règles de niveau de visibilité que les autres méthodes. Il est légitime de se demander à quoi sert un constructeur privé ou protégé, i.e un constructeur inaccessible hors de sa classe. Ce type d'utilisation témoigne en général d'une architecture objet évoluée. Par exemple, le *design pattern* (Gof) nommé communément *singleton* utilise cette facilité. Une autre utilisation d'un constructeur privé se fait dans une classe qui n'a que des membres statiques. Il n'y a alors pas lieu d'instancier une telle classe. Sachez cependant que C#2 introduit la possibilité de déclarer une classe comme statique, ce qui annule l'intérêt d'une telle pratique.

La déclaration d'un constructeur d'une classe dérivée peut appeler un constructeur d'une classe de base.

## *Accès à un constructeur lors de la construction d'un objet*

C++ → C# Contrairement au C++, en C# lors de la construction d'un objet avec l'opérateur `new`, il est impératif de mettre une paire de parenthèses pour signaler explicitement que l'on veut appeler le constructeur sans argument. Nous précisons que C++ ne fait pas la différence entre ces deux syntaxes.

C# Un constructeur est appelé à la création de chaque objet. Voici un exemple dans le cas d'une structure :

*Exemple 11-25 :*

```
struct Article {
   public int Prix ;
   public Article(int Prix) { this.Prix = Prix ; }        // ctor 1
```

```
   public Article(double Prix) { this.Prix = (int)Prix ; }// ctor 2
}
class Program {
   static void Main() {
      Article a ; // Appel au ctor par défaut de la structure.
      Article b = new Article(6) ;   // Appel au ctor 1.
      Article c = new Article(6.3) ; // Appel au ctor 2.
   }
}
```

Voici un exemple dans le cas d'une classe :

*Exemple 11-26 :*

```
class Article {
   public int Prix ;
   public Article(int Prix) { this.Prix = Prix ; }         // ctor 1
   public Article(double Prix) { this.Prix = (int)Prix ; } // ctor 2
   public Article() { Prix = 0 ; }                         // ctor 3
}
class Program {
   static void Main() {
      Article a = new Article() ;   // Appel au ctor 3.
      Article b = new Article(6) ;  // Appel au ctor 1.
      Article c = new Article(6.3); // Appel au ctor 2.
   }
}
```

Le mot-clé `new` est requis pour appeler un constructeur d'un type référence. Pour un type valeur, le mot-clé `new` n'est requis que pour appeler un constructeur avec arguments. Si vous n'utilisez pas le mot-clé `new`, le constructeur par défaut est automatiquement appelé.

# *Destruction des objets*

## *Destructeur, finaliseur et la méthode Finalize()*

C++ → C# La notion de ramasse-miettes est étrangère à la norme C/C++. Il existe bien quelques implémentations de ce type d'algorithme pour C++ mais dans le monde *Microsoft*, elles sont peu utilisées.

Les *destructeurs*, si utilisés et si sujets à controverses en C++ (exceptions lancées dans un destructeur, destructeur virtuel...) existent toujours en C#. Cependant le développeur ne maîtrise absolument pas le moment de l'appel d'un destructeur.

C# Pour une bonne compréhension de la présente section, il faut avoir assimilé la section concernant le ramasse-miettes en page 116. Par rapport à C++ nous parlerons de finaliseur plutôt que de destructeur.

Dans la section concernant le ramasse-miettes, nous expliquons qu'il existe un thread alloué et géré par le CLR qui est dédié à l'appel des *finaliseurs* des objets. Le finaliseur d'une classe est

la réécriture de la méthode `Finalize()` de la classe `Object`. Les structures n'admettent pas de finaliseurs. Le CLR considère qu'une classe qui ne réécrit pas cette méthode n'a pas de finaliseur. Nous avons vu que le finaliseur d'un objet est exécuté après sa collecte par le ramasse-miettes. La mémoire détenue pour stocker l'état de l'objet sera libérée à la prochaine collecte. En conséquence, les objets dont la classe admet un finaliseur survivent à une collecte de plus que les autres. En outre vous ne maîtrisez ni l'identité du thread qui invoque les finaliseurs, ni le moment durant lequel ils sont invoqués. Les objets avec finaliseurs sont donc globalement plus coûteux et moins pratiques à utiliser.

Le compilateur C# interdit la réécriture directe de la méthode `Finalize()` de la classe `Object`. Vous êtes contraint d'utiliser la « syntaxe du destructeur » qui reprend le nom de la classe préfixée par le caractère tilde « ~ ». Le programme suivant illustre cette syntaxe. Ce programme crée une instance de la classe `Article`, puis détruit la seule référence vers cet objet puis contraint le ramasse-miettes à effectuer une collecte :

*Exemple 11-27 :*

```
using System.Threading ;
public class Article {
   public int m_Prix ;
   public Article(int prix) { this.m_Prix = prix ; }   // Constructeur.
   ~Article() {  // Finaliseur défini avec 'la syntaxe du destructeur'.
      System.Console.WriteLine( "Thread#{0} Thread finaliseur.",
         Thread.CurrentThread.ManagedThreadId );
   }
}
class Program {
   static void Main() {
      System.Console.WriteLine( "Thread#{0} Thread principal.",
         Thread.CurrentThread.ManagedThreadId ) ;
      Article a = new Article(300) ;
      a = null; // Le nouvel objet Article n'est plus référencé.
      System.GC.Collect(); // Force une collecte du ramasse-miettes.
      System.GC.WaitForPendingFinalizers();
   }
}
```

Ce programme affiche :

```
Thread#1 Thread principal.
Thread#2 Thread finaliseur.
```

Nous verrons un peu plus loin que le code d'un finaliseur ne devrait contenir que des instructions pour libérer des ressources non gérées. Il est notamment dangereux d'accéder à un objet à partir du finaliseur parce qu'il se peut que cet objet ait déjà été finalisé. En effet, le thread dédié du ramasse-miettes invoque les finaliseurs dans un ordre imprévisible.

Enfin, en page 129 nous décrivons la notion de finaliseur critique qui permet d'ajouter des garanties quant à la fiabilité de la libération des ressource.

## *L'interface IDisposable et sa méthode Dispose()*

C++ → C# Cette technique vise à minimiser les problèmes dus au non déterminisme du ramasse-miettes de C# et n'a donc aucun équivalent en C++.

C# Il existe une technique pour pouvoir appeler automatiquement une certaine méthode sur un objet, quand on décide que l'on n'aura plus besoin de ce dernier. Comprenez bien que cette technique n'est qu'une aide aux développeurs et ne modifie en aucun cas le comportement du ramasse-miettes. Concrètement il faut que la classe implémente l'interface `System.IDisposable` qui n'a qu'un seul membre, la méthode `Dispose()` :

*Exemple 11-28 :*

```
using System.Threading ;
public class Article : System.IDisposable {
   public int m_Prix ;
   public Article(int prix) { this.m_Prix = prix ; } // constructeur
   public void Dispose() {
      System.Console.WriteLine("Thread#{0} dispose.",
         Thread.CurrentThread.ManagedThreadId ) ;
   }
}
class Program {
   static void Main() {
      System.Console.WriteLine("Thread#{0} Thread principal.",
         Thread.CurrentThread.ManagedThreadId ) ;
      Article a = new Article(300) ;
      try {
         // Ici, vous pouvez utiliser l'objet 'a'
         // c'est sa zone de validité.
      }
      finally{
         a.Dispose() ;
      }
   }
}
```

Ce programme affiche :

```
Thread#1 Thread principal.
Thread#1 dispose.
```

Notez bien que le client doit appeler la méthode `Dispose()` à partir d'un block `finally` pour éviter qu'une exception compromette l'appel à `Dispose()`.

Le langage C# fournit une syntaxe avec le mot clé `using` pour invoquer implicitement la méthode `Dispose()` au sein d'un block `try/finally`. Ainsi, la méthode `Main()` du programme précédent peut être réécrite comme suit :

*Exemple 11-29 :*

```
...
   static void Main() {
      System.Console.WriteLine("Thread#{0} Thread principal.",
         Thread.CurrentThread.ManagedThreadId ) ;
      Article a = new Article(300) ;
      using(a){
         // Ici, vous pouvez utiliser l'objet 'a'
         // c'est sa zone de validité.
      }// Cette accolade provoque l'appel automatique de a.Dispose().
       // Elle signe la fin de la zone de validité de 'a'.
   }
...
```

ATTENTION : l'utilisation du mot-clé `using` présentée ci-dessous n'a rien à voir avec les espaces de noms et les alias.

Notez qu'une zone de validité peut être commune à plusieurs objets, de mêmes classes ou de classes différentes :

*Exemple 11-30 :*

```
...
   static void Main() {
      System.Console.WriteLine("Thread#{0} Thread principal.",
         Thread.CurrentThread.ManagedThreadId ) ;
      Article a = new Article(300) ;
      Article b = new Article(400) ;
      using(a) using(b){
         // Ici, éventuellement utilisation des objets 'a' et 'b'.
      }// Fin de la zone de validité des objets 'a' et de 'b'.
   }
...
```

Les méthodes `Dispose()` seront appelées dans l'ordre inverse de la déclaration des `using` à l'accolade de la fin de la zone de validité (ici `b.Dipose()` puis `a.Dispose()` ).

Pour des raisons de sémantique de nom de méthode, vous pouvez être tenté d'utiliser une méthode par exemple nommée `Close()` au lieu d'implémenter l'interface `IDisposable` et sa méthode `Dispose()`. Il vaut mieux éviter cette pratique car le simple fait d'implémenter l'interface `IDispose` est un moyen standard d'exposer à vos clients qu'ils doivent utiliser la méthode `Dispose()`. Il existe même des analyseurs de code capables de détecter que `Dispose()` n'est pas appelée sur un objet dont la classe implémente `IDisposable`.

Si vous optez cependant pour l'existence d'une méthode `Close()` il faut encapsuler l'appel à cette méthode dans la méthode `Dispose()`, certaines classes .NET le font. Il est alors important de bien documenter cela afin d'éviter de voir appeler successivement `Close()` et `Dispose()`. Parfois une telle pratique peut provoquer la levée d'exceptions.

Lorsqu'une classe implémente l'interface `IDisposable` il est conseillé d'interdire l'appel à toutes les autres méthodes sur un objet qui a été « disposé » en lançant une exception de type `System.ObjectDisposedException`. Pour laisser une plus grande marge de manœuvre aux clients de

votre classe, il est aussi conseillé de permettre d'appeler plusieurs fois la méthode `Dispose()` sur un même objet. Bien entendu seul le premier appel aura un effet. Avec ces règles, la classe `Article` peut être réécrite comme ceci (notez le recours à un champ privé booléen) :

*Exemple 11-31 :*

```
public class Article : System.IDisposable {
   public int m_Prix ;
   public Article(int prix) { this.m_Prix = prix ; }

   private bool m_bDisposed = false;
   public void Fct() {
      if (m_bDisposed)
         throw new System.ObjectDisposedException("Nom de l'objet");
      // Ici le corps de Fct().
   }
   public void Dispose() {
      if (!m_bDisposed){
         m_bDisposed = true;
         // Ici libération des ressources.
      }
   }
}
```

L'implémentation précédente de `Article` a une faiblesse. Dans une application multi-threaded, il se peut que le test sur `m_bDisposed` dans une méthode ait lieu entre le test dans la méthode `Dispose()` et l'affectation à `true` dans la méthode `Dispose()`. Pour la même raison, il se peut aussi que le code de libération des ressources de la méthode `Dispose()` soit appelée deux fois par deux thread différents. Pour pallier ce problème sans rendre la classe toute entière *thread-safe* vous pouvez par exemple synchroniser l'accès à `m_bDisposed` comme ceci :

*Exemple 11-32 :*

```
public class Article : System.IDisposable {
   public int m_Prix ;
   public Article(int prix) { this.m_Prix = prix ; }

   private object m_SyncRootDisposed = new object();
   private bool m_bDisposed = false ;
   public void Fct() {
      lock ( m_SyncRootDisposed )
         if ( m_bDisposed )
            throw new System.ObjectDisposedException("Nom de l'objet") ;
      // Ici le corps de Fct().
   }
   public void Dispose() {
      bool bOldDisposedState = true;
      lock ( m_SyncRootDisposed )
         if ( !m_bDisposed ) {
            bOldDisposedState = false;
            m_bDisposed = true ;
```

```
            }
         if ( !bOldDisposedState ) {
            // Ici libération des ressources.
         }
      }
   }
```

## *Conseils d'utilisation des finaliseurs et de IDisposable*

Il est légitime de se demander pourquoi et quand vous devez affubler vos classes avec un finaliseur et/ou avec une méthode `Dispose()`. La réponse en ce qui concerne le finaliseur est simple :

> Toutes les classes qui maintiennent une ou plusieurs ressources non gérées (telle qu'un handle vers un objet *Windows* par exemple) doivent avoir un finaliseur qui libère ces ressources. On peut ajouter qu'il est essentiel qu'une telle classe doit toujours garder ses ressources non gérées privées. Ces deux conditions permettent de garantir que les ressources non gérées seront libérées. En outre, seules les classes doivent être habilitées à maintenir des ressources non gérées puisque les structures n'ont pas de finaliseur.

La remarque précédente implique qu'une classe qui ne maintient que des ressources gérées n'a pas besoin de finaliseur. En effet, il ne servirait alors à rien d'assigner les champs de types référence à nul car un finaliseur est exécuté strictement entre deux collectes du ramasse-miettes et son exécution implique la perte des références contenues dans les champs de l'objet concerné. On peut être tenté d'implémenter un finaliseur dans une classe qui maintient des références vers des objets gérées qui ont une méthode `Close()` ou `Dispose()` (comme une connexion à une base de données par exemple). En fait, ce genre de classe maintient directement ou indirectement par l'intermédiaire d'autres classes des ressources non gérées. L'appel à la méthode `Close()` ou `Dispose()` ne permet donc que d'anticiper le moment où ces ressources non gérées seront libérées. Il ne sert donc à rien d'invoquer ces méthodes dans un finaliseur. Il faut les invoquer avant, à partir d'une méthode `Dispose()`. Et ceci soulève une autre question : quand doit-on implémenter l'interface `IDisposable` ?

Une partie de la réponse est simple : si votre classe maintient des références vers des objets dont la durée de vie est la même que celle de ses instances et dont l'implémentation présente une méthode telle que `Dispose()` ou `Close()`, il faut qu'elle implémente une méthode `Dispose()` qui appelle les méthodes `Dispose()` et `Close()` de ces objets. Si votre classe ne répond pas au critère cité, la décision d'implémenter ou non l'interface `IDisposable` est plus compliquée puisque vous êtes face à un dilemme. D'un coté il est bon d'avoir une méthode `Dispose()` pour mettre à nul les références que vous maintenez vers d'autres objets gérés de façon à augmenter leurs chances d'être libérés lors de la prochaine collecte. D'un autre coté implémenter l'interface `IDisposable` complexifie votre code ainsi que celui de vos clients. Le choix doit donc se faire en fonction des besoins en mémoire de vos applications et de la taille des objets gérés référencés.

Lorsqu'une de vos classes implémente un finaliseur, vous pouvez être tenté d'implémenter aussi l'interface `IDisposable` pour permettre au client de libérer les ressources (gérées et non gérées) au plus tôt. Dans ce cas nous vous conseillons d'appliquer les règles suivantes :

- Appliquer les règles énoncées pour implémenter `IDisposable`.

- Implémentez une méthode `protected virtual void Dispose(bool bDisposeManagedRes)`. Appelez cette méthode à partir de la méthode `Dispose()` avec le paramètre `bDisposeManagedRes` égal à `true`. Appelez cette méthode à partir du finaliseur avec le paramètre `bDisposeManagedRes` égal à `false`.
- Dans la méthode `Dispose(bool bDisposeManagedRes)`, libérez les ressources non gérées et ne libérez les ressources gérées que si `bDisposeManagedRes` est égal à `true`.
- Dans la méthode `Dispose()`, après avoir appelé la surcharge `Dispose(bool bDisposeManagedRes)` appelez `GC.SuppressFinalize(this)`. Vous indiquez ainsi au ramasse-miettes qu'il n'aura pas besoin d'appeler le finaliseur de cet objet.

Si vous appliquez ces conseils, vous soulagez ainsi le thread finaliseur de la libération inutile des ressources gérées, vous vous protégez des oublis de l'appel à `Dispose()` de la part de vos clients et vous optimisez l'exécution des clients qui n'oublient pas d'appeler `Dispose()`. Voici une classe `Foo` qui suit ces conseils (sans la synchronisation des accès à `m_bDisposed`) :

*Exemple 11-33 :*

```
public class Foo : System.IDisposable {
   public Foo() {
      // Ici éventuellement allocation des ressources.
   }
   private bool m_bDisposed = false;
   public void Fct() {
      if ( m_bDisposed )
         throw new System.ObjectDisposedException("Nom de l'objet");
      // Ici le corps de Fct().
   }
   public void Dispose() {
      Dispose(true);
      System.GC.SuppressFinalize(this);
   }
   ~Foo() { Dispose(false) ; }
   protected virtual void Dispose(bool bDisposeManagedRes) {
      if ( !m_bDisposed ) {
         m_bDisposed = true;
         // Ici libérer les ressources non-gérées.
         if ( bDisposeManagedRes ) {
            // Ici libérer les ressources gérées.
         }
      }
   }
}
```

Dans le cas où `Foo` est une classe de base qui admet des classes dérivées qui maintiennent des ressources, il est judicieux de réécrire la méthode `Dispose(bool bDisposeManagedRes)` puisque celle-ci est virtuelle et protégée. Il est intéressant de remarquer que le design des classes du *framework* suit les règles énoncées.

# Les membres statiques

C++ → C# En C#, il existe plusieurs petites différences de syntaxe par rapport aux membres statiques de C++. En C# :

- Nous accédons aux membres statiques avec cette syntaxe `NomDeLaClasse.NomDuMembreStatic`. Nous n'avons pas besoin de l'opérateur de résolution de portée comme en C++.
- Nous ne pouvons pas accéder à un membre statique au moyen d'une instance de la classe.
- Nous pouvons avoir des classes statiques.

Au niveau conceptuel rien ne change mis à part qu'il peut exister un constructeur statique et des classes purement statiques.

C# Les champs, les propriétés, les méthodes et les événements d'une classe ont la possibilité d'être déclarés avec le mot-clé `static`. Ce mot-clé signifie que le membre appartient à la classe, et non aux objets, comme c'est le cas des membres non statiques. Les membres statiques sont parfois appelés *membres partagés,* car ils sont partagés par les instances de la classe. D'ailleurs le langage VB.NET utilise le mot-clé `Shared` qui veut dire partagé en anglais à la place du mot clé C# `static`. Les membres non statiques sont appelés *membres d'instances*.

Les membres déclarés comme statiques subissent les mêmes règles de niveau de visibilité que les autres membres. De plus, parmi les différentes sortes de membres, seuls les types encapsulés et les indexeurs ne peuvent être définis avec le mot-clé `static`.

## Champs propriétés et événements statiques

Les champs, propriétés ou événements statiques d'une classe, se distinguent par les caractéristiques suivantes :

- Ils existent indépendamment des instances de la classe, y compris si aucune instance de la classe n'a encore été créé.
- Ils sont partagés par toutes les instances de la classe.
- Ils sont accessibles dans toutes les méthodes de la classe par leurs noms, et à l'extérieur de la classe (selon leurs niveaux de visibilité) par la syntaxe : `NomDeLaClasse.NomDuChampStatic`

Les champs peuvent être initialisés directement durant leurs déclarations au sein de la classe. Comme pour l'initialisation des champs d'instances, l'initialisation des champs statiques est un sujet subtil. En effet, lorsque la classe est créée, les champs statiques sont d'abord initialisés à leurs valeurs par défaut. Puis ils sont initialisés avec leur valeur d'initialisation (pour ceux qui en ont une), dans leur ordre d'apparition dans la classe. Enfin le constructeur statique (voir plus loin) est appelé. Tout ceci permet de faire des références croisées comme ci-dessous, bien que ce soit fortement déconseillé :

*Exemple 11-34 :*

```
public class Program {
   static int a = b + 2 ;  // a référence b pour son initialisation.
   static int b = a + 3 ;  // b référence a pour son initialisation.
   public static void Main() {
      System.Console.WriteLine("a = {0}", a) ;  // a = 2
      System.Console.WriteLine("b = {0}", b) ;  // b = 5
   }
}
```

Le cas de figure suivant porte aussi à confusion, d'ailleurs le résultat est différent du précédent. En effet, la construction de la classe `CB` se fait au milieu de la construction de la classe `CA`, puisque `CA` est appelée en premier mais `CA` utilise `CB` !

*Exemple 11-35 :*

```
public class CA {
   static public int a = CB.b + 2 ;
}
public class CB {
   static public int b = CA.a + 3 ;
}
public class Program {
   public static void Main() {
      System.Console.WriteLine("CA.a = {0}", CA.a) ;  // CA.a = 5
      System.Console.WriteLine("CB.b = {0}", CB.b) ;  // CB.b = 3
   }
}
```

## *Méthodes statiques*

Les méthodes statiques se distinguent par les caractéristiques suivantes :

- Une méthode statique n'a pas accès aux méthodes, champs, propriétés et événements non statiques de la classe.
- Une méthode statique n'est pas accessible à partir d'une référence vers une méthode de la classe,
- Une méthode statique est accessible dans toutes les méthodes (statiques ou non) de la classe par son nom, et à l'extérieur de la classe (si son niveau de visibilité le permet) par la syntaxe : `NomDeLaClasse.NomDeLaMéthodeStatique()`
- Une méthode statique ne peut utiliser le mot-clé `this` expliqué dans la section précédente.

## *Constructeur statique (ou constructeur de classe)*

C# autorise la définition d'un *constructeur statique* parfois nommé aussi *constructeur de classe* ou *cctor* (pour *class constructor*). Comme toutes les méthodes statiques, cette méthode n'a accès qu'aux membres statiques et peut servir, par exemple, à les initialiser. La syntaxe est la même que celle d'un constructeur sans argument, avec le mot-clé `static` au début. Un constructeur statique n'a pas de niveau de visibilité. En outre, ni le moment exact de l'appel à ce constructeur

ni quel thread réalise cet appel ne sont fixés par la spécification du langage C#. Cependant, les points suivant de la spécification C# peuvent vous aider à évaluer ce moment :

- Le constructeur statique est appelé par le CLR lorsque celui-ci charge les métadonnées de types de la classe concernée, dans le domaine d'application courant.
- Le constructeur statique doit être appelé avant toute création d'instance de la classe.
- Le constructeur statique doit être appelé avant qu'un membre statique de la classe soit accédé.
- Le constructeur statique doit être appelé après que les champs statiques initialisés explicitement dans le code, soient initialisés.
- Le constructeur statique d'une classe chargée par le CLR doit être appelé au moins une fois durant l'exécution du programme.

Vous avez la possibilité de forcer l'appel au constructeur d'une classe avec la méthode `static void RunClassConstructor(type t)` de la classe `System.Runtime.CompilerServices.RuntimeHelpers`. Naturellement, l'exécution n'a lieu que si le constructeur de classe concerné n'a pas déjà été appelé.

Voici une classe qui compte le nombre d'instances couramment valides, ce nombre étant initialisé dans son constructeur statique :

*Exemple 11-36 :*

```
public class Article {
   static int NbArticles ;      // Un champ statique.
   static Article() {          // Le constructeur statique.
      NbArticles = 0;
   }
   int Prix = 0 ;               // Un champ privé non statique.
   public Article(int Prix) {  // ctor
      this.Prix = Prix ;
      NbArticles++ ;
   }
   ~Article() { NbArticles-- ; }// finaliseur
}
```

## *Classes statiques*

C# permet la définition de *classes statiques*. Une classe statique ne peut avoir que des membres statiques. N'ayant pas de constructeur d'instance, une classe statique ne peut être instanciée. En outre on ne peut dériver d'une classe statique. Notez qu'une structure ne peut être statique.

Pour définir une classe statique il suffit d'utiliser le mot-clé `static` dans sa définition :

*Exemple 11-37 :*

```
static class Program {
   static void Main() { }
}
```

# *Surcharge des opérateurs*

C++ → C# En C# la surcharge des opérateurs arithmétiques présente beaucoup de différences avec le C++. On peut déjà dire que la surcharge d'opérateurs est plus simple, moins permissive mais moins puissante en C# qu'en C++.

Tous d'abord, de la quarantaine des opérateurs surchargeables en C++, il n'en reste qu'une vingtaine en C#. Ceux qui n'existent pas en langage C# sont :

- les opérateurs « `+=` » « `%=` » etc ;
- les opérateurs « `&&` » « `||` » ;
- l'opérateur d'affectation « `=` » ;
- l'opérateur modulo « `%` » ;
- les opérateurs d'allocation/désallocation « `new` » « `delete[]` » etc ;
- les opérateurs très proches du langage « `->` » « `->*` » « `,` » ;
- l'opérateur d'appel d'une fonction « `()` » qui permettait les *foncteurs* (fonctions objets) sur laquelle repose une grande partie de la STL. Cette notion de foncteur reste cependant implémentable d'un autre façon comme nous l'expliquons en page 591.

L'opérateur d'indexation « `[]` » a donné naissance en C# aux indexeurs présentés un peu plus haut. Les opérateurs de transtypages existent en C# et sont décrits dans la présente section.

La syntaxe de déclaration des opérateurs a aussi changé. En effet en C++ une surcharge peut être interne à la classe (la méthode de surcharge de l'opérateur est non statique et dans la classe) ou externe (la méthode de surcharge de l'opérateur est une fonction amie de la classe). En C# la méthode de surcharge de l'opérateur est une fonction statique de la classe obéissant à certaines règles énoncées ci-dessous. Notez qu'en C# la notion « d'amitié » du C++ n'existe pas.

C# Pour présenter les opérateurs surchargeables en C#, nous allons les classer en trois catégories :

- Les opérateurs arithmétiques « `+` » « `-` » etc.
- Les opérateurs de conversion de type, aussi appelés opérateurs de transtypage.
- Les opérateurs de comparaison.

## *Surcharge des opérations arithmétiques*

Le langage C# permet à une classe de surcharger des opérateurs arithmétiques tels que l'opérateur plus « `+` » ou l'opérateur de négation « `!` », avec le mot-clé `operator`. Concrètement lorsque le compilateur C# rencontre des objets de cette classe manipulés avec un opérateur surchargé par la classe, il appelle la méthode adéquate, qui correspond à l'opérateur.

Une méthode qui surcharge un opérateur est statique. À l'instar de toutes les méthodes statiques, elle est sujette aux niveaux de visibilité. Les opérateurs arithmétiques que l'on peut redéfinir se classent en deux catégories :

- Les *opérateurs unaires*, qui n'agissent que sur un seul opérande. La méthode statique surchargeant un opérateur unaire n'accepte qu'un argument du type de sa classe. De plus elle renvoie aussi une valeur du type de sa classe.

- Les *opérateurs binaires* qui agissent sur deux opérandes. La méthode statique surchargeant un opérateur binaire a deux arguments : le premier du type de la classe et le deuxième d'un type quelconque. Elle renvoie une valeur de type quelconque.

Tous les opérateurs ne sont pas surchargeables, et c'est heureux. En effet, le code devient vite plus complexe lorsque des opérateurs sont surchargés, à moins que la signification de l'opérateur soit vraiment logique dans le contexte de la classe. En effet, chaque opérateur surchargeable a une signification bien connue des développeurs. Ainsi l'opérateur « + » est utilisé pour l'addition ou la concaténation. Il est logique de l'utiliser dans une classe représentant des nombres (par exemple des nombres fractionnaires, complexes ou les quaternions etc) d'autres objets mathématiques contenus dans un ensemble avec une structure de groupe additive (matrice, vecteur, fonction, forme etc) ou des chaînes de caractères. Mais, que penser d'une utilisation de cet opérateur dans une classe représentant des personnes ? C'est pour cette raison que nous vous conseillons de ne surcharger les opérateurs que lorsque cela est logique dans le contexte de la classe.

Voici la liste des opérateurs surchargeables :

| **Opérateurs unaires surchargeables** | `+ - ! ++ --` |
|---|---|
| Opérateurs binaires surchargeables | `+ - * / % & \| ^ << >>` |

Certains opérateurs sont à la fois binaires et unaires. Par exemple l'opérateur « - » peut à la fois rendre un opérande négatif, ou soustraire deux opérandes.

Les opérateurs unaires d'incrémentation « ++ » et de décrémentation « -- » surchargés, provoquent l'appel vers la même méthode, qu'ils soient postfixés ou préfixés. Rappelons que lorsque l'on utilise ces opérateurs, l'ordre d'évaluation des opérateurs est différent, selon leur position par rapport à l'opérande.

Voici un exemple d'utilisation des opérateurs. Nous créons une classe `Distance` et une classe `Surface`. En interne les valeurs sont sauvées dans des doubles ou une unité représente un mètre (respectivement un mètre carré). Nous surchargeons les opérateurs plus « + » et étoile « * » entre deux distances, l'opérateur d'incrémentation « ++ » d'un mètre sur une distance, et l'opérateur de division « / » par une distance ou par une surface pour une surface :

*Exemple 11-38 :*

```
public class Distance {
   double m_Mesure = 0.0 ;
   public double Mesure { get { return m_Mesure ; }
                          set { m_Mesure = value ; } }
   public Distance(double d) { m_Mesure = d ; }
   public static Distance operator +(Distance d1, Distance d2) {
      return new Distance(d1.m_Mesure + d2.m_Mesure) ;
   }
   public static Surface operator *(Distance d1, Distance d2) {
      return new Surface(d1.m_Mesure * d2.m_Mesure) ;
   }
   public static Distance operator ++(Distance d) {
```

```
      return new Distance(d.m_Mesure++) ;
   }
}
public class Surface {
   double m_Mesure = 0.0 ;
   public double Mesure {
      get { return m_Mesure ; }
      set { m_Mesure = value ; }
   }
   public Surface(double d) { m_Mesure = d ; }
   public static Distance operator /(Surface s, Distance d) {
      return new Distance(s.m_Mesure / d.Mesure) ;
   }
   public static double operator /(Surface s1, Surface s2) {
      return s1.m_Mesure / s2.m_Mesure ;
   }
}
class Program {
   static void Main() {
      Distance d1 = new Distance(5.3) ;
      Distance d2 = new Distance(2.4) ;
      Distance d3 = d1 + d2 ;
      Surface s1 = d1 * d2 ;
      Surface s2 = d3 * d2 ;
      Distance d4 = s1 / d3 ;
      double dRapport = s1 / s2 ;
      Distance d5 = d1++ ; // Aprés ceci d5 mesure 6.3, d1 mesure 5.3.
      Distance d6 = ++d1 ; // Aprés ceci d6 mesure 5.3, d1 mesure 5.3.
   }
}
```

Soyez conscient que dans ce code, nous ne testons pas la nullité potentielle des références en entrée des opérateurs. Nous acceptons donc le fait qu'une exception de type `NullReferenceException` sera levée automatiquement par le CLR le cas échéant.

La façon dont nous avons codé l'opérateur « ++ » dans `Distance` provoque un résultat inattendu. En effet l'appel de cet opérateur retourne un nouvel objet `Distance` dont la mesure est celle de la distance source incrémenté de un. Ce comportement est particulièrement dangereux puisqu'en général, l'opérateur « ++ » modifie l'objet sur lequel il est appelé. Une façon plus logique de coder cet opérateur est :

*Exemple 11-39 :*

```
...
   public static Distance operator ++(Distance d) {
      d.m_Mesure++ ; return d ;
   }
...
      Distance d5 = d1++ ; // Après ceci d5 mesure 6.3, d1 mesure 6.3.
      Distance d6 = ++d1 ; // Après ceci d6 mesure 7.3, d1 mesure 7.3.
...
```

Cet exemple montre bien la nécessité de coder les opérateurs dans la logique de leur utilisation habituelle en C#. Sinon le code devient vite complètement illisible.

## *Opérateurs de conversion de type (de transtypage)*

C++ → C# Comme C++, C# autorise la définition d'opérateurs de transtypage explicites ou implicites. La syntaxe du langage C# oblige ces opérateurs à être statiques.

En C#, les constructeurs de transtypages n'existent pas. Rappelons qu'en C++ les constructeurs de transtypage et les opérateurs de transtypages ont la même fonctionnalité. La différence vient du fait que les constructeurs de transtypages sont dans la classe destination et les opérateurs de transtypages dans la classe source. Ceci peut mener à une ambiguïté lorsque les deux formes coexistent. Ce problème n'existe donc pas en C#.

C# L'opération de transtypage représente la construction d'un objet instance d'une classe destination, à partir d'un objet instance d'une classe source. C# permet de créer vos propres opérateurs de transtypage dans la classe source, c'est-à-dire des méthodes automatiquement appelées lorsque vous écrivez :

```
CDest objDest = objSrc ;
```

Le code de ces méthodes contient principalement des recopies de champs et de propriétés. Il existe deux types d'opérateurs de transtypage.

- Les *opérateurs de transtypage implicite*. L'écriture suivante suffit à provoquer l'appel à l'opérateur de transtypage implicite de `CSrc` vers `CDest`, déclaré dans `CSrc`.

  ```
  CDest objDest = objSrc ;
  ```

- Les *opérateurs de transtypage explicite*. L'écriture précédente ne suffit pas à provoquer l'appel à l'opérateur de transtypage explicite de `CSrc` vers `CDest`, déclaré dans `CSrc`. Dans ce cas, le compilateur génère une erreur et le développeur doit alors explicitement exprimer le transtypage comme ceci :

```
CDest objDest = (CDest) objSrc ;
```

Comme tous les opérateurs, les opérateurs de transtypage sont des méthodes statiques. Ils sont déclarés dans la classe source et ils portent le nom de la classe destination. L'utilisation de l'un des mots-clés `explicit` ou `implicit` est obligatoire.

Voici un exemple de code où un objet de classe `Distance` peut être transtypé implicitement en une variable `double` et explicitement en objet de classe `DistanceEntière`. Notez la nécessité de tenir compte de la nullité potentielle de la référence en entrée des opérateurs :

*Exemple 11-40 :*

```
public class Distance {
   public double m_Mesure = 0.0 ;
   public Distance(double d) { m_Mesure = d ; }
   public static implicit operator double(Distance d) {
      // Doit tenir compte du cas ou d est nulle.
      if (object.ReferenceEquals(d, null))
         return 0.0 ;
```

```
      return d.m_Mesure ;
   }
   public static explicit operator DistanceEntiere(Distance d) {
      if (object.ReferenceEquals(d, null))
         // Vous pouvez aussi préférer retourner la référence
         // nulle dans ce cas.
         return new DistanceEntiere(0) ;
      // Notez la nécessité de construire un nouvel objet pour un
      // type destination référence.
      return new DistanceEntiere((int)d.m_Mesure) ;
   }
}
// Distance entière signifie que la distance est codée sur un entier.
public class DistanceEntiere {
   public int m_Mesure = 0 ;
   public DistanceEntiere(int i) { m_Mesure = i ; }
}
class Program {
   static void Main() {
      Distance d1 = new Distance(5.3) ;
      // OK le transtypage est implicite.
      double dbl1 = d1 ;
      // La forme explicite est aussi acceptée.
      double dbl2 = (double)d1 ;
      // Erreur de compilation !! : le transtypage doit être explicite.
      DistanceEntiere de1 = d1 ;
      // OK le transtypage est explicite.
      DistanceEntiere de2 = (DistanceEntiere)d1 ;
      // Teste le cas où la référence source est nulle.
      Distance d2 = null ;
      double dbl3 = (double)d2 ;
      DistanceEntiere de3 = (DistanceEntiere)d2 ;
   }
}
```

Il est toujours préférable de déclarer ses opérateurs de transtypage comme explicites. Dans le cas contraire votre code risque d'être permissif, c'est-à-dire que le compilateur aura à faire des choix. Au mieux il déclarera une erreur car il ne saura pas quel choix faire. Au pire, il ne donnera pas d'avertissements et fera le choix contraire à votre attente. Par exemple imaginons qu'une instance de la classe `Distance` puisse être convertie implicitement en une instance du type `double` ou une instance de la classe `string`. Quel choix de transtypage le compilateur doit-il faire lorsque l'on veut afficher cet objet avec la méthode `Console.WriteLine(object)` ?

*Exemple 11-41 :*

```
public class Distance {
   public double m_Mesure = 0.0 ;
   public Distance(double d) { m_Mesure = d ; }
   public static implicit operator double(Distance d) {
      if (object.ReferenceEquals(d, null))
```

```
            return 0.0 ;
         return d.m_Mesure ;
      }
      public static implicit operator string(Distance d) {
         if (object.ReferenceEquals(d, null))
            return null ;
         return string.Format("Distance:{0:##.##} mètres", d.m_Mesure) ;
      }
   }
   class Program {
      static void Main() {
         Distance d1 = new Distance(5.3) ;
         // Erreur de compilation : Opérateur de transtypage ambiguë.
         // Doit-on transtyper d1 en un double ou une string ?

         System.Console.WriteLine(d1);
         // OK pas d'ambiguïté, mais il vaut mieux déclarer les opérateurs
         // de transtypage comme explicite.

         System.Console.WriteLine( (string) d1);
      }
   }
```

## *Opérateurs de comparaison*

C++ → C# Comme C++, C# permet de redéfinir le comportement des opérateurs de comparaison « == » et « != ».

Cependant les motivations sont différentes en C#. Par défaut le comportement de ces opérateurs sur des objets de type référence est de comparer si les références référencent le même objet. Or, assez souvent, on souhaite comparer le contenu.

De plus en C# les opérateurs de comparaison ne sont pas définis par défaut sur les structures, contrairement à C++.

C# Les *opérateurs de comparaison* sont les suivants :

| Opérateur | Nom | Commentaires |
|---|---|---|
| == | Opérateur d'égalité. | Lorsque cet opérateur est défini, le compilateur oblige le développeur à définir aussi l'opérateur d'inégalité. |
| != | Opérateur d'inégalité. | Lorsque cet opérateur est défini, le compilateur oblige le développeur à définir aussi l'opérateur d'égalité. |
| <= | Opérateur inférieur ou égal. | Lorsque cet opérateur est défini, le compilateur oblige le développeur à définir aussi l'opérateur supérieur ou égal. |

| | | |
|---|---|---|
| >= | Opérateur supérieur ou égal. | Lorsque cet opérateur est défini, le compilateur oblige le développeur à définir aussi l'opérateur inférieur ou égal. |
| < | Opérateur inférieur strictement. | Lorsque cet opérateur est défini, le compilateur oblige le développeur à définir aussi l'opérateur supérieur strictement. |
| > | Opérateur supérieur strictement. | Lorsque cet opérateur est défini, le compilateur oblige le développeur à définir aussi l'opérateur strictement. |

Par défaut, si ces opérateurs ne sont pas redéfinis, les règles suivantes sont appliquées selon la nature des types des opérandes :

- Si les deux opérandes sont de types primitifs ou énumération, le compilateur va d'abord essayer de transtyper un des deux opérandes pour qu'il soit du type de l'autre. S'il y parvient, les opérateurs de comparaison vont comparer le contenu des objets.
- Si les deux opérandes sont de même type structure, le compilateur produit une erreur. La comparaison par défaut sur des structures n'existe pas en langage C#.
- Si les deux opérandes sont de type référence et que l'opérateur de comparaison est l'égalité ou l'inégalité, le compilateur va d'abord vérifier que les classes des deux objets référencés sont les mêmes. Cette condition peut être remplie même si les types des références sont différents. En effet, l'héritage, qui fait l'objet du prochain chapitre, permet à un objet et à une référence vers cet objet d'être de types différents. Si les deux objets référencés ont la même implémentation, les opérateurs de comparaison vont vérifier si c'est le même objet qui est référencé ou non.
- Si les deux opérandes sont de type référence et que l'opérateur de comparaison n'est pas l'égalité ou l'inégalité, une erreur est produite par le compilateur. La comparaison par défaut autre que l'égalité/inégalité, sur des types référence n'existe pas en langage C#.
- Si un opérande est de type valeur et l'autre de type référence, le compilateur va produire une erreur.

Lorsque vous décidez de redéfinir les opérateurs d'égalité et d'inégalité, il est conseillé que vous redéfinissiez aussi la méthode `Equals()` de la classe `Object`. Si vous suivez ce conseil, il faut que les deux surcharges appellent la méthode `Equals()`, comme dans l'exemple suivant : (plus d'information concernant la méthode `Object.Equals()` sont disponibles en page 345, notamment on y explique pourquoi la compilation du programme suivant émet un avertissement quant à la méthode `Object.GetHashCode()`).

*Exemple 11-42 :*

```
public class Distance {
   private double m_Mesure = 0.0 ;
   public Distance(double d) { m_Mesure = d ; }
   public override bool Equals(object obj) {
      Distance d = obj as Distance;
      // Vérifie que l'objet auquel on se compare est bien une distance.
```

```
      // Par la même occasion traite le cas où obj est nulle.
      if( !Distance.ReferenceEquals( d , null ) )
         // Comparaison du contenu.
         return m_Mesure == d.m_Mesure;
      return false;
   }
   public static bool operator ==(Distance d1, object d2) {
      // Traite les cas où une ou les deux références sont nulles.
      if( Distance.ReferenceEquals( d1 , null ) ) {
         return Distance.ReferenceEquals( d2 , null ) ;
      }
      return d1.Equals(d2) ;
   }
   public static bool operator !=(Distance d1, object d2) {
      return ! (d1 == d2) ;
   }
}
class Program {
   static void Main() {
      Distance d1 = new Distance(5.2) ;
      Distance d2 = new Distance(5.2) ;
      Distance d3 = new Distance(7.3) ;
      Distance d4 = null ;
      // Toutes ces assertions sont vrais.
      System.Diagnostics.Debug.Assert( d1 == d2 ) ;
      System.Diagnostics.Debug.Assert( d1 != d3 ) ;
      System.Diagnostics.Debug.Assert( d1 != null ) ;
      System.Diagnostics.Debug.Assert( null != d1 ) ;
      System.Diagnostics.Debug.Assert( !(d1 == null) ) ;
      System.Diagnostics.Debug.Assert( !(null == d1) ) ;
      System.Diagnostics.Debug.Assert( d4 == null ) ;
   }
}
```

Comme vous pouvez le constater en analysant cet exemple, surcharger les opérateurs == et =! n'est pas une manipulation aisée :

- Il faut d'abord envisager les cas où une voire les deux références sont nulles. Par convention, deux références nulles doivent être considérées comme égales.
- Il faut aussi envisager le cas où l'autre référence à laquelle on se compare ne référence pas un objet de type `Distance`.
- Enfin, en interne, il faut penser à éviter l'utilisation des opérateurs == et != au risque de provoquer une boucle infinie. C'est pour cela que nous faisons appel à la méthode `Object.ReferenceEquals()`.

## *La surcharge des opérateurs et le CLS*

La surcharge des opérateurs est présente en C#, principalement pour améliorer la lisibilité du code source C#. La surcharge des opérateurs n'est pas *CLS Compliant*. Cela veut dire que d'autres

langages .NET ne la supportent pas et ne peuvent pas l'utiliser. Aussi, si votre classe est destinée à être utilisée par d'autres langages .NET, nous vous conseillons de prévoir des méthodes publiques avec les mêmes fonctionnalités que vos opérateurs redéfinis. Par exemple si vous surchargiez l'opérateur « + » dans une classe, nous vous conseillons de prévoir une méthode statique `Add()` remplissant la même fonctionnalité. Rappelons que si vous avez un doute quant aux respect de votre code par rapport au CLS, vous pouvez utilisez l'attribut `CLSCompliant` que nous décrivons en page 131.

Voici la liste des noms de substitution donnés aux méthodes qui ont pour but d'offrir la même fonctionnalité qu'un opérateur redéfini. Ces noms sont conseillés, et le compilateur vous autorise à utiliser vos propres identificateurs.

| **Opérateur** | **Nom de substitution conseillé.** |
|---|---|
| *Opérateurs de transtypage* | |
| Transtypage implicite vers le type `Xxx` | `ToXxx FromXxx` |
| Transtypage explicite vers le type `Xxx` | `ToXxx FromXxx` |
| *Opérateurs arithmétiques binaires* | |
| `+` | `Add` |
| `-` | `Subtract` |
| `*` | `Multiply` |
| `/` | `Divide` |
| `%` | `Mod` |
| `^` | `Xor` |
| `&` | `BitwiseAnd` |
| `\|` | `BitwiseOr` |
| `<<` | `LeftShift` |
| `>>` | `RightShift` |
| *Opérateurs arithmétiques unaires* | |
| `+` | `Plus` |
| `-` | `Negate` |
| `++` | `Increment` |
| `--` | `Decrement` |
| | `OnesComplement` |

| | |
|---|---|
| ! | Not |
| *Opérateurs de comparaison* | |
| == | Equals |
| != | Compare |
| <= | Compare |
| >= | Compare |
| < | Compare |
| > | Compare |

# 12

# *Héritage/dérivation polymorphisme et abstraction*

## *Objectif : réutilisation de code*

### *La problématique*

La loi suivante est fondamentale en programmation :

> La complexité d'un programme en fonction de sa taille, grandit plus vite qu'une fonction linéaire de sa taille.

Cette loi est illustrée par la Figure 12-1 :

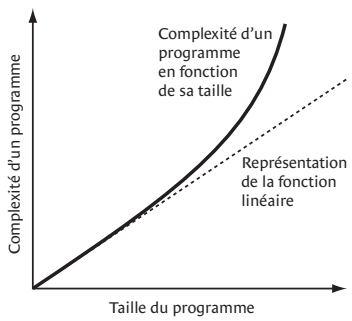


*Figure 12-1 : Complexité d'un programme en fonction de sa taille*

Un programme deux fois plus gros qu'un autre est donc plus que deux fois plus long à écrire, tester et maintenir.

Cette loi est empirique. Le travail de du concepteur d'un programme consiste à ce que la complexité reste aussi proche que possible de la fonction linéaire. Le mécanisme de dérivation ou d'héritage est un point important pour l'aider dans cette tâche mais nous verrons en fin du présent chapitre que ce n'est pas le seul.

Concrètement, on part du constat que dans un programme, différentes classes ont des fonctionnalités identiques :

- Dans un programme de gestion de personnel d'une entreprise, il peut y avoir une classe pour les secrétaires, une classe pour les techniciens, une classe pour les cadres. Toutes ces classes ont ceci de commun qu'une instance représente un employé, avec les attributs `Nom`, `Age`, `Adresse`, `Salaire` et les méthodes `Evalue()`, `ModifieLesHoraires()`, `Augmente()`.
- Dans un programme de dessin, il peut y avoir une classe pour les cercles, une classe pour les rectangles, une classe pour les triangles. Toutes ces classes ont ceci de commun qu'une instance a les attributs `CouleurDuTrait`, `TailleDuTrait` et les méthodes `Dessine()`, `Translation()`, `Rotation()`, `Grossi()`.
- Dans un programme qui communique avec différents protocoles de communication, chaque point de communication a, indépendamment du protocole sous-jacent, les attributs `NbOctetsEnvoyés/Reçus`, `DateDeCréation` et la méthode `EnvoieUnStream()`.

## *Une solution : l'héritage*

L'idée de la réutilisation est de cerner ces similitudes, et de les réunir dans une classe que l'on nomme *classe de base* (par exemple `Employe`, `FigureGeometrique` ou `PointDeCommunication`).

Une *classe dérivée* est une classe qui *hérite* des membres d'une classe de base. Concrètement si la classe `Technicien` hérite de la classe `Employe`, la classe `Technicien` hérite des champs `Nom` et `Age` et des méthodes `Evalue()` et `ModifieLesHoraires()`. On dit aussi que la classe dérivée est une spécialisation de la clase de base et qu'une classe de base d'une classe dérivée est une *super classe* de cette classe dérivée.

Une des difficultés dans la conception d'un programme est de cerner correctement les similitudes entre classes, et ce le plus tôt possible dans le cycle de développement. L'autre difficulté majeure est de définir comment les objets interagissent entre eux. Voici quelques phrases à se dire pour confirmer l'intuition.

- Un Technicien **est un** employé, un Rectangle **est une** figure géométrique, une socket **est un** point de communication.
- Un Technicien **a un** nom, un Rectangle **a une** couleur de trait, une socket **a un** nombre d'octets envoyés depuis sa création.
- Un Technicien **peut être** évalué, un Rectangle **peut être** grossi, une socket **peut être** utilisée pour envoyer un stream.

# *L'héritage d'implémentation*

## *La syntaxe*

C++ → C# La syntaxe pour l'héritage d'implémentation est la même en C# et C++. Conceptuellement il existe une grosse différence entre C++ et C#. C# ne supporte pas l'héritage multiple. Une classe C# ne peut dériver que d'une seule classe de base. Nous verrons que ce manque est en partie compensé par la formalisation, en C#, du concept, plus simple d'interface et d'implémentation de plusieurs interfaces par une classe.

C# En C#, la syntaxe pour indiquer que la classe `Technicien` dérive de la classe `Employe` est la suivante :

```
// La classe Technicien hérite de la classe Employe.
class Technicien : Employe {  ...  }
```

Une classe ne peut dériver que d'une seule classe de base. En langage objet, on dit que C# supporte l'*héritage d'implémentation simple* mais pas l'*héritage d'implémentation multiple*. Ce n'est pas le cas de tous les langages orientés objet. Les langages C++ et EIFFEL supportent l'héritage d'implémentation multiple alors que le langage Java ne le supporte pas.

En C#, si une classe `C` dérive d'une classe `B`, rien n'empêche la classe `B` de dériver d'une classe `A`. On dit que `C` dérive indirectement de `A`. En outre la classe `B` est à la fois une classe dérivée et une classe de base.

## *Les niveaux de visibilité protégé et interne protégé*

C++ → C# Comme C++, C# permet de protéger des membres d'une classe de base. La notion de protection d'un membre est identique entre les deux langages. Le mot-clé est aussi le mot `protected`, et la seule différence est qu'en C# il faut écrire ce mot-clé pour chaque membre protégé. C# ajoute la notion de membre interne protégé (mots-clés `internal protected`).

C# Rappelons les définitions des niveaux de visibilité protégé et interne protégé que nous présentons en page 418 :

- Le niveau de visibilité *protégé* : mot-clé `protected`.

  Un membre qui a le niveau de visibilité *protégé* et qui est défini dans un type `T` est accessible seulement à partir du code contenu dans `T` (types encapsulés de `T` compris) ainsi qu'à partir du code des types dérivés de `T`. Ceci reste valable pour le code des types dérivés de `T` définis dans d'autres assemblages.

- Le niveau de visibilité *interne protégé* : mot-clé `internal protected`.

  Un membre qui a le niveau de visibilité *interne protégé* et qui est défini dans un type `T` est accessible partout dans le code de l'assemblage courant. Cependant, si le type `T` est utilisé dans d'autres assemblages le membre qualifié d'interne protégé n'est visible que dans le code des méthodes des types dérivant de `T`. Notez que c'est le seul cas où deux mots-clés peuvent être combinés pour déclarer un niveau de visibilité.

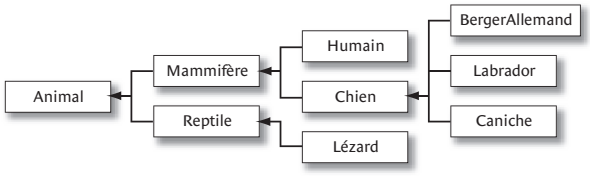


*Figure 12-2 : Exemple de schéma de dérivation*

## *Schéma de dérivation*

Une classe de base peut être aussi une classe dérivée. Tout ceci peut être illustré dans un schéma de dérivation. Par exemple :

Il existe des langages de modélisation pour les schémas objets, le plus courant étant le langage UML (*Unified Modelling Langage*). Nous avons utilisé la notation du diagramme de classes de ce langage pour représenter le schéma de la Figure 12-2. Comme vous le constatez, en UML, une flèche blanche pointe vers une classe de base, et se divise vers les classes dérivées.

*Visual Studio 2005* présente la possibilité de visualiser un diagramme de classes ressemblant à un diagramme de classes UML. Vous pouvez avoir accès à cette possibilité en cliquant droit sur un projet et en sélectionnant *View Class Diagram*. Un fichier XML d'extension `.cd` est alors inséré au projet. Il contient la mise en forme de votre diagramme. Voici par exemple un extrait de la hiérarchie des classes de contrôles de *Windows Forms* présentées en page 679 :

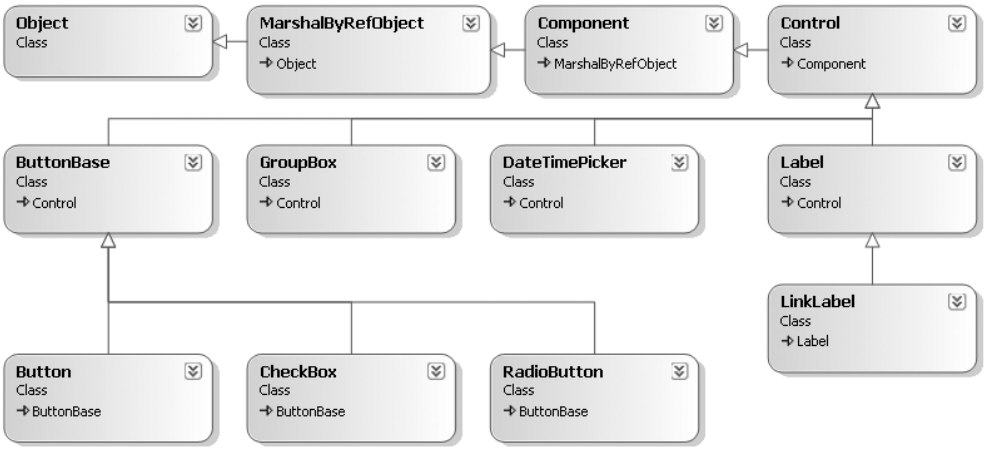


*Figure 12-3 : Diagramme de classe de Visual Studio 2005*

## *Appel aux constructeurs d'une classe de base*

C++ → C♯ Comme C++, C♯ permet d'appeler explicitement un constructeur de la classe de base dans chaque constructeur d'une classe dérivée. Comme C++, si l'appel n'est pas explicite-

ment déclaré, le compilateur appelle le constructeur sans argument de la classe de base s'il y en a un, sinon il y a une erreur de compilation.

Cependant, la syntaxe du langage C# est légèrement différente de celle du langage C++. Comme il n'y a pas d'héritage d'implémentation multiple en C#, la classe de base est unique. Donc il n'est pas nécessaire de préciser le nom de la classe de base. Dans ce contexte, en C# le nom de la classe de base est remplacé par le mot-clé `base`.

C# En C#, chaque constructeur d'une classe dérivée appelle un constructeur de la classe de base. Cet appel peut être implicite, c'est-à-dire qu'il n'y a pas de code qui montre clairement que cet appel doit se faire. Dans ce cas c'est le constructeur sans argument de la classe de base qui est appelé. Si la classe de base n'a pas de constructeur sans argument, alors une erreur est produite à la compilation. Rappelons les principes suivants :

- Si une classe ne déclare pas de constructeur : le compilateur fournis automatiquement un constructeur par défaut, qui n'accepte pas d'argument.
- Si une classe déclare un seul constructeur : il n'y a qu'un seul constructeur, celui-ci. Le compilateur ne fournit pas de constructeur par défaut.
- Si une classe déclare plusieurs constructeurs : ils diffèrent selon leurs signatures et on dit que le constructeur est surchargé. Le compilateur ne fournit pas de constructeur par défaut.

L'appel au constructeur de la classe de base peut être explicite à partir d'un constructeur d'une classe dérivée. Pour cela le mot-clé `base` doit être utilisé avec la même syntaxe que dans l'exemple suivant :

*Exemple 12-1 :*

```
public class Employe {
   string m_Nom ;
   short m_Age ;
   // Constructeur acceptant comme liste d'arguments : string, short
   public Employe(string Nom, short Age) {
      m_Nom = Nom ;
      m_Age = Age ;
   }
   // Constructeur sans arguments.
   public Employe() {
      m_Nom = "n/a" ;
      m_Age = 0 ; // <- Pas nécessaire, un short est initialisé à 0.
   }
}
class Technicien : Employe {  // Technicien hérite de Employe.
   string m_Competences ;
   // Appel au constructeur de la classe de base acceptant comme liste
   // d'arguments : string, short
   public Technicien(string nom, short age, string competences)
       : base(nom,age) {
      m_Competences = competences ;
   }
}
```

```
class Program {
   static void Main() {
      Technicien roger = new Technicien("Roger", 45, "Dépanneur PC") ;
   }
}
```

### *Niveau de visibilité des membres de la classe de base*

C++ → C# En C# vous ne pouvez pas spécifier le niveau de visibilité qu'a une classe dérivée sur les membres de sa classe de base. En C# tout se passe comme en C++ lorsque ce contrôle d'accès est `public`.

### *Les classes dont on ne peut dériver (sealed)*

C++ → C# Contrairement à C++, le langage C# permet de spécifier clairement qu'une classe ne peut en aucun cas être une classe de base.

C# C# permet de spécifier qu'une classe ne peut en aucun cas être une classe de base. C'est-à-dire qu'aucune classe ne peut dériver de celle-ci. Il suffit de mettre le mot-clé `sealed` devant la déclaration de la classe. On utilise parfois le terme de *classe finalisée* pour nommer cette possibilité. Par exemple :

```
sealed class Foo{  /* Ici les membres de la classe. */ }
```

Une classe finalisée peut être une classe dérivée. D'ailleurs, comme tous les types, les classes finalisées dérivent de la classe `Object`. De plus, les structures peuvent être vues comme des classes finalisées de type valeur.

## *Méthodes virtuelles et polymorphisme*

C++ → C# Attention nous ne parlons ici que de méthodes virtuelles, c'est-à-dire ayant une implémentation dans la classe de base. Les méthodes virtuelles pures du C++, appelées méthodes abstraites en C# seront vues un peu plus loin.

Les concepts de polymorphisme et de méthodes virtuelles sont identiques en C++ et en C#. Cependant C# permet d'empêcher que le polymorphisme s'applique sur une certaine méthode virtuelle d'une certaine classe dérivée. De plus et il y a quelques petites différences au niveau syntaxique :

- En C++, pour avoir accès au polymorphisme, on utilise le plus souvent des pointeurs de type classe de base sur lesquels on appelle des méthodes virtuelles. En C#, les pointeurs n'étant pas très populaires, on utilise la plupart du temps des références typées par une classe de base, référençant des objets d'implémentations dérivées. Notez qu'en C++ le polymorphisme est aussi accessible par l'intermédiaire de références, mais cette possibilité est moins utilisée que les pointeurs.
- En C++ une méthode virtuelle est déclarée avec le mot-clé `virtual` dans la première classe de base qui implémente cette méthode. Les implémentations de cette méthode dans les classes dérivées peuvent optionnellement réutiliser ce mot pour signaler que cette méthode est virtuelle. À la lecture du code il n'est donc pas toujours évident qu'une méthode est

virtuelle. Le langage C# résout ce problème. Une méthode virtuelle est toujours déclarée avec le mot-clé `virtual`, dans la première classe de base qui l'implémente. Dans chaque classe dérivée où la méthode virtuelle est redéfinie, il faut faire précéder la méthode du mot-clé `override`.

Une différence conceptuelle :

- La différence conceptuelle au niveau des méthodes virtuelles, entre C++ et C#, se situe dans le fait qu'en C#, le développeur peut redéfinir une méthode virtuelle tout en interdisant le polymorphisme de s'appliquer. Pour cela on doit utiliser le mot-clé `new` au lieu du mot-clé `override`. Si on ne met rien aucun de ces deux mots-clés, le compilateur produira seulement un avertissement du type : « vous devriez utiliser `new` » mais à l'exécution tout se passe comme si on avait utilisé `new`. Soyez vigilant !

## *La problématique*

En programmation objet, on est souvent confronté au problème suivant : on crée des objets, instances de plusieurs classes dérivées d'une classe de base, puis on veut leur appliquer un traitement de base, c'est-à-dire, un traitement défini dans la classe de base. Le problème est que ce traitement diffère selon la classe dérivée. Par exemple :

- On veut obtenir une description de tous les employés (traitement de base : obtenir une description d'un employé, quelle que soit sa catégorie).
- On veut dessiner toutes les figures géométriques (traitement de base : dessiner une figure géométrique, quel que soit le type de figure).
- On veut envoyer des données par l'intermédiaire d'un point de communication (traitement de base : envoyer des données, quel que soit le protocole de communication sous-jacent).

## *La solution : les méthodes virtuelles et le polymorphisme*

Il est très utile de rassembler les objets qui doivent subir le traitement de base, puis de leur faire subir le traitement de base à chacun (par exemple dans une boucle).

> Toute l'élégance de cette méthode vient du fait que le traitement de base s'applique sur un objet, sans connaître précisément sa classe, on ne le connaît que par une référence de type sa classe de base : c'est une illustration du polymorphisme.

Ce que nous avons appelé traitement de base, s'appelle *méthode virtuelle*. C'est une méthode définie à la fois dans la classe de base (précédée du mot clé `virtual`) et dans la classe dérivée (précédée du mot-clé `override`).

Il existe plusieurs corps pour cette méthode, un dans la classe de base et éventuellement un dans chacune des classes dérivées. Lors de l'appel de cette méthode sur une référence de type classe de base, le corps adéquat est choisi à l'exécution par le programme. La notion de plusieurs corps pour une même méthode se traduit par le mot *polymorphisme* (plusieurs formes).

Une classe dérivée n'est pas obligée de redéfinir le corps d'une méthode virtuelle de sa classe de base. Une classe de base qui contient au moins une méthode virtuelle est une *classe polymorphe*. En outre, il est possible de redéfinir le corps d'une méthode virtuelle avec l'utilisation conjointe des mots-clés `override sealed`. Dans ce cas, cela signifie que cette méthode virtuelle ne peut plus être redéfinie dans les classes dérivées de la classe dérivée. On parle de *méthode finalisée*.

## *Un exemple*

Voici un exemple où l'on a :

- Une classe de base `Employe` et deux classes dérivées `Technicien` et `Secretaire`
- Un traitement de base qui est d'afficher la description de l'employé. Ce traitement diffère selon la classe : Pour un objet de type `Employe` ce traitement se limite à afficher son nom. Pour un objet de type `Techicien` ce traitement exécute l'affichage de l'employé suivi du texte « *Fonction :Technicien* ». Pour un objet de type `Secretaire` ce traitement exécute l'affichage de l'employé suivi du texte « *Fonction :Secrétaire* ».

On peut illustrer ceci avec un diagramme de classe obtenu avec *Visual Studio 2005* :

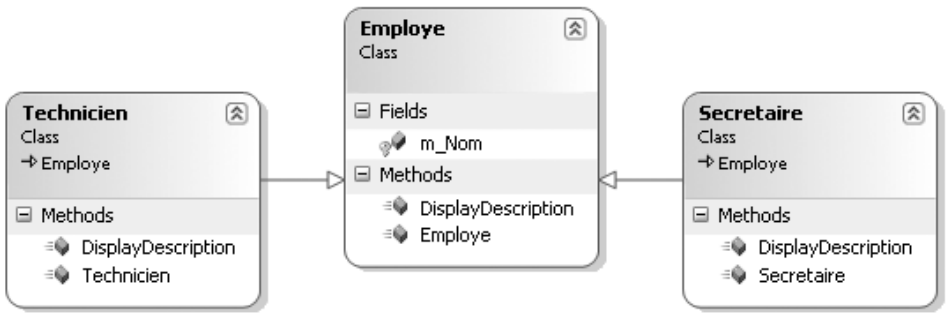


*Figure 12-4 : Diagramme de nos classes*

Le corps du programme consiste à créer trois employés référencés dans un tableau, dont les éléments sont des références de type `Employe`. Ensuite, on applique le traitement de base (l'affichage de la description) pour chaque employé référencé dans le tableau.

*Exemple 12-2 :*

```
public class Employe {
   // m_Nom peut etre accédé dans les méthodes des classes dérivées.
   protected string m_Nom ;
   public Employe(string nom) {m_Nom = nom;}
   public virtual void DisplayDescription() {
      System.Console.Write("Nom : {0}",m_Nom) ;
   }
}
class Secretaire : Employe {  // Secretaire hérite de Employe.
   public Secretaire(string nom):base(nom) {}
   public override void DisplayDescription() {
      // Appel de la méthode DisplayDescription() de Employe.
      base.DisplayDescription();
      System.Console.Write( "   Fonction : Secrétaire\n") ;
   }
}
```

```
class Technicien : Employe {  // Technicien hérite de Employe.
   public Technicien(string nom):base(nom) {}
   public override void DisplayDescription(){
      // Appel de la méthode DisplayDescription() de Employe.
      base.DisplayDescription();
      System.Console.Write( "  Fonction : Technicien\n") ;
   }
}
class Program {
   static void Main() {
      Employe[] tableau = new Employe[3] ;
      tableau[0] = new Technicien("Line") ;
      tableau[1] = new Secretaire("Lisanette") ;
      tableau[2] = new Secretaire("Anne-Mette") ;
      foreach( Employe employe in tableau )
         employe.DisplayDescription();
   }
}
```

Voici ce qu'affiche ce programme :

```
Nom : Line  Fonction : Technicien
Nom : Lisanette  Fonction : Secrétaire
Nom : Anne-Mette  Fonction : Secrétaire
```

La méthode `DisplayDescription()` est toujours appelée sur une référence de type `Employe`. Pourtant on voit bien que le programme exécute les méthodes `DisplayDescription()` de `Technicien` et `Secretaire`. C'est la magie du polymorphisme. Il évite un test de type fastidieux et facilite la maintenance. En effet lors de l'ajout d'une autre classe dérivée de `Employe` (une classe `Cadre` par exemple) il faudrait mettre à jour le test, alors qu'avec le polymorphisme, il n'y a absolument rien à faire pour que la fonction `Cadre::DisplayDescription()` soit appelée.

Les méthodes virtuelles `DisplayDescription()` appellent la méthode `DisplayDescription()` de la classe de base avec le mot-clé `base`. Pour chacune de vos classes dérivées, nous vous conseillons de vérifier que chaque méthode virtuelle appelle à un moment ou à un autre la méthode qui lui correspond dans la classe de base. Si ce n'est pas le cas il y a sûrement un problème de conception dans votre modèle objet.

## *Redéfinition d'une méthode et désactivation du polymorphisme*

C++ → C♯ Attention, la désactivation du polymorphisme n'est pas possible en C++ alors que la réécriture d'une méthode non virtuelle d'une classe de base dans une classe dérivée est permise.

C♯ C♯ permet de cacher une méthode (virtuelle ou non) d'une classe de base dans une classe dérivée, par la redéfinition avec le mot-clé `new` placé devant la méthode dans la classe dérivée.

Attention, dans ce cas le polymorphisme ne s'applique pas, y compris si la méthode est déclarée comme virtuelle dans la classe de base. Notez que si on ne met pas le mot-clé `new` le compilateur produira seulement un avertissement du type : « vous devriez mettre `new` » mais à l'exécution tout se passe comme si on avait mis `new`. On comprend alors, que l'utilisation explicite de `new` sert à clarifier le code.

Reprenons l'exemple précédent et observons l'effet de l'utilisation de new pour redéfinir une méthode virtuelle :

*Exemple 12-3 :*

```
public class Employe{
...
}
class Secretaire : Employe{  // Secretaire hérite de Employe.
...
}
class Technicien : Employe{  // Technicien hérite de Employe.
   public Technicien(string nom):base(nom) {}
   public new void DisplayDescription(){
      base.DisplayDescription() ;
      System.Console.Write( "   Fonction : Technicien\n") ;
   }
}
class Program {
   static void Main(){
      Employe[] tableau = new Employe[3] ;
      tableau[0] = new Technicien("Line") ;
      tableau[1] = new Secretaire("Lisanette") ;
      tableau[2] = new Secretaire("Anne-Mette") ;
      foreach( Employe employe in tableau )
         employe.DisplayDescription();
      // Appel de la méthode DisplayDescription() de Technicien
      ((Technicien)tableau[0]).DisplayDescription();
   }
}
```

Voici ce qu'affiche ce programme :

```
Nom : LineNom : Lisanette   Fonction : Secrétaire
Nom : Anne-Mette   Fonction : Secrétaire
Nom : Line   Fonction : Technicien
```

Remarquez que la méthode DisplayDescription() est appelée deux fois pour l'objet décrivant Line. Cependant la première fois, il s'agit de la méthode DisplayDescription() de Employe alors que la deuxième fois, c'est la méthode DisplayDescription() de Technicien. Dans le premier cas on voit bien que le polymorphisme a été désactivé. Dans le deuxième cas, on a explicitement transtypé notre référence de type Employe vers l'objet qui représente Line, en une référence de type Technicien. Il n'y a donc pas eu là aussi, d'application de polymorphisme. Le fait de transtyper explicitement une référence de type une classe de base vers une référence, de type classe dérivée se nomme « *downcast* ». Nous reviendrons un peu plus loin sur ce sujet.

# *L'abstraction*

C++ → C# Le concept d'abstraction est complètement similaire entre les langages C# et C++. Cependant des différences de syntaxe apparaissent, et même de vocabulaire.

Les méthodes virtuelles pures du langage C++ s'appellent, en langage C#, méthodes abstraites et se déclarent avec le mot-clé `abstract` en préfixe.

De plus il ne suffit plus qu'un classe abstraite ait au moins une méthode abstraite pour être abstraite. Il faut qu'elle soit déclarée avec le mot-clé `abstract` en préfixe. Contrairement au C++, en C# il peut donc y avoir des classes abstraites sans méthodes abstraites.

Malgré les apparences, le concept d'interface existe en C++ : rien n'empêche de déclarer une classe avec seulement des méthodes abstraites et sans aucun champ. Cependant C# met en évidence ce concept avec le mot-clé `interface`. La différence est qu'en C#, le fait d'utiliser le mot-clé `interface` oblige les développeurs à respecter des contraintes (pas d'ajout de champ, pas d'ajout de corps de méthode etc).

Les interfaces sont très importantes en C# puisqu'une classe peut implémenter plusieurs interfaces, ce qui permet de combler en partie le manque d'héritage d'implémentation multiple.

## *La problématique*

Dans l'exemple de la section précédente, afficher la description d'un employé (i.e afficher son nom) a un sens. Il est donc utile de mettre du code dans la méthode virtuelle `DisplayDescription()` de la classe de base `Employe`. Lorsqu'elle est instanciée, la classe `Employe` a quelque chose à afficher.

Il arrive que l'on n'ait pas de code à mettre dans la méthode virtuelle parce qu'il y a un manque d'information à ce niveau de l'arbre d'héritage.

Par exemple pour la classe `FigureGeometrique` il n'y a rien à mettre dans la méthode virtuelle `Dessine()`. En effet, à ce niveau de l'héritage on ne sait pas quel type de figure géométrique on instancie.

De même pour la classe `PointDeCommunication` (aussi définie plus haut) il n'y a rien à mettre dans la méthode `EnvoieUnStream()` puisque à ce niveau de l'héritage on ne connaît pas le protocole de communication sous-jacent.

On pourrait très bien déclarer une fonction virtuelle sans code à l'intérieur. Mais on sent bien que l'on a besoin de mécanismes plus performants pour déclarer proprement ces classes de base qui ont un manque d'information sur ce qu'elles représentent lorsqu'une de leurs classes dérivées est instanciée. Une telle classe de base veut imposer des opérations à ces classes dérivées, alors qu'elle même n'a pas assez d'informations pour implémenter ces opérations, même en partie.

## *La solution : les classes abstraites et les méthodes abstraites*

La solution s'appelle l'*abstraction*. Une *classe abstraite* est une classe qui doit déléguer complètement l'implémentation de certaines de ces méthodes à ses classes dérivées. On l'a vu, la raison pour laquelle ces méthodes ne peuvent être implémentées est que la classe abstraite a un manque d'information, mais souhaite imposer des opérations à ces classes dérivées.

Ces méthodes qui ne peuvent être implémentées s'appellent des *méthodes abstraites* (ou *virtuelles pures*). Elles se déclarent avec le mot-clé `abstract` en préfixe et doivent être contenues dans des classes abstraites, déclarées aussi avec le mot-clé `abstract` en préfixe.

Une méthode abstraite n'est qu'une méthode virtuelle particulière, et doit être implémentée dans les classes dérivées avec le mot-clé `override` en préfixe (ou `override sealed`). Si une classe

dérivée d'une classe abstraite n'implémente pas toutes les méthodes abstraites, elle est elle-même abstraite.

La conséquence immédiate et fondamentale de tout ceci est que : **Une classe abstraite n'est pas instanciable.**

Il ne peut y avoir d'objets instances d'une classe abstraite, mais une référence peut avoir pour type une classe abstraite. Une telle référence référence alors un objet d'une classe dérivée non abstraite et l'application du polymorphisme lors de l'appel des méthodes virtuelles et abstraites est alors automatiquement mise en œuvre. Le polymorphisme est encore plus évident lorsqu'il est utilisé sur des méthodes abstraites puisqu'on est certain que ce ne peut être le corps de la méthode abstraite qui est appelée, puisqu'il n'existe pas !

Remarquez qu'une méthode abstraite ne doit pas avoir une visibilité privée. En effet, les classes dérivées seraient dans l'impossibilité de l'implémenter.

## *Un exemple*

Voici un exemple où l'on a : une classe de base abstraite `FigureGeometrique`, deux classes dérivées `Cercle` et `Rectangle`, un traitement de base qui est de dessiner la figure.

Ce traitement diffère selon la classe. Il est abstrait pour la classe `FigureGeometrique` (i.e il n'est pas implémentable dans cette classe car il y a un manque d'informations concrètes sur la nature de la forme géométrique). Pour un objet de type `Cercle`, ce traitement exécute l'affichage d'un cercle en fonction du centre et du rayon. Pour un objet de type `Rectangle`, ce traitement exécute l'affichage d'un rectangle en fonction de trois de ses sommets.

*Exemple 12-4 :*

```
class Point{
   public Point (int x,int y){this.x = x;this.y = y;}
   int x ; int y ;
}
abstract class FigureGeometrique{
   // On vérifie qu'une méthode abstraite n'a pas de corps.
   public abstract void Dessine();
}
class Cercle : FigureGeometrique {
   private    Point    m_Centre ;
   private    double    m_Rayon ;
   public Cercle(Point centre, double rayon) {
      m_Centre = centre ;
      m_Rayon  = rayon ;
   }
   public override void Dessine (){
      // Dessine un Cercle à partir de son centre et de son rayon.
   }
}
class Rectangle : FigureGeometrique {
   private Point m_Sommet1 ;
   private Point m_Sommet2 ;
   private Point m_Sommet3 ;
```

```
   public Rectangle(Point s1, Point s2, Point s3 ) {
      m_Sommet1 = s1 ; m_Sommet2 = s2 ; m_Sommet3 = s3 ;
   }
   public override void Dessine (){
      // Dessine un Rectangle à partir de trois de ses sommets.
   }
}
class Program {
   static void Main() {
      FigureGeometrique[] tableau = new FigureGeometrique[3] ;
      tableau [0] = new Cercle(new Point(0,0),3.2) ;
      tableau [1] = new Rectangle(
                        new Point(0,0),new Point(0,2),new Point(1,2)) ;
      tableau [2] = new Cercle(new Point(1,1),4.1) ;
      // Le polymorphisme s'applique à l'appel
      // de la méthode abstraite Dessine().
      foreach(FigureGeometrique f in tableau )
         f.Dessine() ;
      // Erreur de compilation !
      // On ne peut instancier une classe abstraite !
      FigureGeometrique figure = new FigureGeometrique();
   }
}
```

Notez qu'une classe abstraite peut avoir des constructeurs, des champs et des propriétés.

## *Usage simultané des mots-clé abstract et override*

Il est possible de marquer une méthode simultanément avec ces deux mots-clés. Imaginez une classe C qui dérive d'une classe `B` qui elle-même dérive d'une classe `A` qui déclare une méthode `Foo()` comme abstraite. `B` peut elle-même être une classe abstraite et ne pas implémenter la méthode `Foo()`. Dans ce cas la déclaration de la méthode `B.Foo()` doit être précédée des mots-clés `abstract override`. Cela signifie que la classe `B` repousse l'implémentation de cette méthode à ces classes dérivées :

*Exemple 12-5 :*

```
abstract class A { public abstract void Foo();}
abstract class B : A { public abstract override void Foo();}
class C : B { public override void Foo() { /*...*/ } }
```

Comme l'illustre l'exemple suivant, l'association des mots clés `abstract` et `override` peut aussi permettre à `B` de forcer ses classes dérivées à réimplémenter la méthode `Foo()`, déjà implémentée par la classe `A` :

*Exemple 12-6 :*

```
class A { public virtual void Foo() { /*...*/ } }
abstract class B : A { public abstract override void Foo();}
// La classe C est forcée de réimplémenter la méthode Foo.
class C : B { public override void Foo() { /*...*/ } }
```

# *Les interfaces*

C++ → C# Une interface peut être vue comme une classe abstraite sans champ avec seulement des méthodes, des propriétés, des événements ou des indexeurs abstraits. Ce concept est tout à fait implémentable en C++, mais C# va plus loin en offrant la possibilité d'utiliser le mot-clé `interface` à la place du mot-clé `class`. Cette distinction est importante en C#. En effet une classe peut éventuellement dériver de plusieurs interfaces.

C# Il existe des classes abstraites très particulières. Ce sont celles qui n'ont que des méthodes, des propriétés, des événements ou des indexeurs abstraits. La théorie de la programmation objet les appelle *interfaces* ou bien *abstractions*. On dit qu'une classe implémente une interface au lieu de dire qu'une classe dérive d'une interface. On dit aussi qu'une classe qui implémente une interface est une *implémentation* de l'interface. Les structures peuvent aussi implémenter des interfaces mais cette possibilité est dangereuse comme nous l'expliquerons un peu plus loin.

Pour déclarer une interface en langage C#, il suffit de déclarer une classe avec le mot-clé `interface` au lieu du mot-clé `class`. Une interface ne peut avoir que quatre types de membres : des méthodes (des spécifications de méthodes), des propriétés, des événements et des indexeurs.

Les niveaux de visibilité ne peuvent s'appliquer aux méthodes d'une interface. C'est à la classe qui implémente l'interface d'en décider. En fait, on ne peut utiliser que les mots-clés `virtual` et `public` devant la déclaration d'un membre d'une interface. Les conséquences de ces règles d'utilisation sont détaillées dans les sections suivantes. Voici un point à souligner à propos des interfaces :

Une classe peut éventuellement implémenter plusieurs interfaces (en plus de dériver de sa classe de base). De même une interface peut dériver d'une ou plusieurs autres interfaces. On dit d'une telle interface qu'elle *étend* les autres interfaces.

Cette possibilité permet de rajouter vos propres comportements à une classe, mais aussi des comportements définis par des interfaces standard de l'architecture .NET. Par exemple l'interface `IEnumerator` permet à un objet d'être utilisé par la structure de contrôle de boucle `foreach`, comme s'il s'agissait d'un tableau. D'autres interfaces .NET utilisables dans vos classes sont présentées dans cet ouvrage. Nous vous conseillons de faire commencer le nom de vos interfaces par un « I » majuscule, à l'instar des interfaces standard.

Une référence peut être de type interface. Dans ce cas on ne peut accéder à l'objet qu'avec le comportement spécifique de l'interface, par exemple :

*Exemple 12-7 :*

```
interface IA { void f(int i) ; }
interface IB { void g(double d) ; }
class C : IA, IB {
   public void f(int i) { System.Console.WriteLine("f de C {0}", i) ; }
   public void g(double d){ System.Console.WriteLine("g de C {0}", d) ; }
```

```
}
class Program {
   static void Main() {
      // Une réference interface sur un objet.
      IA obj1 = new C() ;
      // Une réference interface sur un objet.
      IB obj2 = new C() ;
      // Récupération de l'objet d'après une réference interface.
      C _obj2 = (C)obj2 ;
      obj1.f(5) ;
   }
}
```

Contrairement aux classes et aux structures, les interfaces ne dérivent pas de la classe `Object`. Cependant comme seules les classes et les structures sont à même d'implémenter une interface, il est permis d'appeler une méthode de la classe `Object` sur une interface. L'exemple suivant montre qu'il est aussi permis de transtyper une référence de type interface en référence de type `Object`.

*Exemple 12-8 :*

```
interface I {}
class C : I {}
class Program {
   static void Main() {
      I c = new C() ;
      c.GetHashCode();
      object o = c;
   }
}
```

## *Obliger un client à utiliser une abstraction plutôt qu'une implémentation*

.NET offre une possibilité inédite par rapport aux langages C++ et Java. Vous pouvez forcer les clients de vos classes à utiliser une abstraction (i.e une référence de type une interface) plutôt qu'une implémentation (i.e une référence vers une instance d'une classe qui implémente l'interface). Cette possibilité se révèle extrêmement utile pour les concepteurs de bibliothèques de classes. Ils ont à leur disposition une technique qui fait partie intégrante du langage pour obliger le code de leurs clients à être découplé des implémentations. Or, le découplage des classes est un des principes fondamentaux de la POO avec la cohérence du code.

Voici un exemple. La méthode `fct2()` ne peut être appelée à partir d'une référence de type `C` mais peut être appelée à partir d'une référence de type `I`. Dans un cas réel, on aurait sûrement forcé l'utilisation de l'interface sur toutes les méthodes présentées par l'interface.

*Exemple 12-9 :*

```
interface I {
   void fct1() ;
```

```
   void fct2() ;
}
public class C : I {
   public void fct1() { System.Console.WriteLine("fct1 appelée") ; }
   void I.fct2() { System.Console.WriteLine("fct2 appelée") ; }
}
public class Program {
   public static void Main() {
      C refImpl = new C() ;
      I refAbst = (I)refImpl ;
      refAbst.fct1();  // Compilateur OK
      refAbst.fct2();  // Compilateur OK
      refImpl.fct1();  // Compilateur OK
      refImpl.fct2() ;  // Compilateur KO : Le message d'erreur est :
      // 'C' does not contain a definition for 'fct2'
   }
}
```

## *Conflits de noms de méthodes*

Puisqu'une classe peut implémenter plusieurs interfaces et dériver d'une classe, il se peut qu'il y ait conflit entre des méthodes ayant le même nom et la même signature. Pour résoudre ces conflits il faut utiliser la syntaxe décrite dans la section précédente :

*Exemple 12-10 :*

```
interface IA { void f(int i); }
interface IB { void f(int i); }
abstract class FooBase { public abstract void f(int i); }
class FooDeriv : FooBase, IA, IB {
   void IA.f(int i) { System.Console.WriteLine("IA.f({0})", i) ; }
   void IB.f(int i) { System.Console.WriteLine("IB.f({0})", i) ; }
   public override void f(int i) {
      System.Console.WriteLine("f({0})", i) ;
   }
}
class Program {
   static void Main() {
      FooDeriv refImpl = new FooDeriv() ;
      FooBase refAbst = (FooBase)refImpl ;
      IA refA = (IA)refImpl ;
      IB refB = (IB)refImpl ;
      refImpl.f(1) ;
      refAbst.f(2) ;
      refA.f(3) ;
      refB.f(4) ;
   }
}
```

Ce programme affiche :

```
f(1)
f(2)
IA.f(3)
IB.f(4)
```

Une même classe peut donc avoir plusieurs implémentations pour une même méthode. Telle ou telle implémentation sera exécutée en fonction du type de la référence sur laquelle la méthode est appelée. C'est une forme de polymorphisme.

## *Conflits d'extensions d'interfaces*

C++ → C# En C++, on parle d'héritage virtuel lorsqu'une classe hérite plusieurs fois de la même classe. Par exemple, supposez que la classe D dérive des classes C et B, qui elles mêmes dérivent de la classe A. La classe D dérive donc deux fois de la classe A. En utilisant la notion d'héritage virtuel, vous pouvez préciser si vous souhaitez qu'une instance de D soit constituée de deux instances de A ou d'une seule.

Le sujet de cette section se rapproche conceptuellement de la possibilité d'héritage virtuel du C++, bien qu'ici, on parle d'héritage d'abstraction multiple, et non d'héritage d'implémentation multiple.

C# Une classe a la possibilité d'implémenter plusieurs interfaces qui étendent la même interface. Cette possibilité est utile lorsqu'on a un diagramme UML qui ressemble à celui de la Figure 12-5.

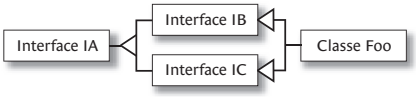


*Figure 12-5 : Conflit d'extension d'interfaces*

L'exemple suivant illustre cette possibilité et ses conséquences :

*Exemple 12-11 :*

```
interface IA { void fa() ; }
interface IB : IA { new void fa(); void fb() ; }
interface IC : IA { new void fa(); void fc() ; }
class Foo : IB, IC {
   void IA.fa() { System.Console.WriteLine("IA.fa") ; }
   void IB.fa() { System.Console.WriteLine("IB.fa") ; }
   void IC.fa() { System.Console.WriteLine("IC.fa") ; }
   public void fa() { System.Console.WriteLine("Foo.fa") ; }
   public void fb() { System.Console.WriteLine("Foo.fb") ; }
   public void fc() { System.Console.WriteLine("Foo.fc") ; }
}
class Program {
   static void Main() {
      Foo r = new Foo() ; r.fa() ;
      IA rA = r ;         rA.fa() ;
```

```
      IB rB = r ;         rB.fa() ;
      IC rC = r ;         rC.fa() ;
   }
}
```

Ce programme affiche :

```
Foo.fa
IA.fa
IB.fa
IC.fa
```

## *Réécriture de l'implémentation d'une interface*

Vous avez la possibilité de déclarer une méthode d'une interface implémentée dans une classe, comme virtuelle. Elle pourra ainsi être réécrite dans une classe dérivée de la classe qui implémente l'interface. Voici un exemple pour illustrer cette possibilité :

*Exemple 12-12 :*

```
interface I {
   void f(int i);
   void g(int i) ;
}
class FooBase : I {
   public virtual void f(int i) {
      System.Console.WriteLine("FooBase.f({0})", i) ;
   }
   public void g(int i) {
      System.Console.WriteLine("FooBase.g({0})", i) ;
   }
}
class FooDeriv : FooBase {
   public override void f(int i) {
      System.Console.WriteLine("FooDeriv.f({0})", i) ;
   }
}
class Program {
   static void Main() {
      FooBase  refB1 = new FooBase() ;
      I        refI1 = refB1 ;
      FooDeriv refD  = new FooDeriv() ;
      FooBase  refB2 = refD ;
      I        refI2 = refD ;
      refB1.f(1) ;
      refI1.f(2) ;
      refD.f(3) ;
      refB2.f(4) ;
      refI2.f(5) ;
   }
}
```

Ce programme affiche :

```
FooBase.f(1)
FooBase.f(2)
FooDeriv.f(3)
FooDeriv.f(4)
FooDeriv.f(5)
```

## *Implémentation d'une interface dans une structure*

Pour comprendre cette section il est nécessaire d'avoir assimilé ce que représentent les opérations de boxing et de unboxing décrit dans la section page 350.

La présente section a pour but de vous sensibiliser à un problème classique qui survient lorsqu'une structure implémente une interface. Si vous essayez d'accéder aux membres de l'interface implémentés par la structure à partir d'une référence vers l'interface vous n'obtiendrez certainement pas le résultat escompté. En effet, lors du transtypage de la structure vers l'interface, une opération de boxing est implicitement réalisée car l'interface z besoin d'une référence. Pour vous en convaincre, analysez l'exemple suivant :

*Exemple 12-13 :*

```
interface I {
   void SetState(int i) ;
   int GetState() ;
}
struct Struct : I {
   private int i ;
   public void SetState(int i) { this.i = i ; }
   public int GetState() { return i ; }
}
class Program {
   static void Main() {
      Struct s = new Struct() ;
      // Ici un boxing de la structure est réalisé implicitement.
      I i = (I)s;
      s.SetState(10) ;
      i.SetState(20) ;
      System.Console.WriteLine("Retour de s.GetState():"+ s.GetState()) ;
      System.Console.WriteLine("Retour de i.GetState():"+ i.GetState()) ;
   }
}
```

Ce programme affiche :

```
Retour de s.GetState():10
Retour de i.GetState():20
```

En règle générale il vaut mieux ne pas implémenter d'interface dans les structures. Si vous n'avez pas d'autres possibilités, il vaut mieux appeler les méthodes de l'interface à partir de l'implémentation, contrairement à ce qui est conseillé pour les classes.

Pour les plus sceptiques, l'analyse avec `ildasm.exe` du code IL généré pour la méthode `Main()` montre clairement qu'il y a une opération de boxing réalisée. Notez que les deux autres opérations de boxing sont nécessaires pour pouvoir afficher l'état l'entier retourné par la méthode `GetState()` :

```
.method private hidebysig static void  Main() cil managed {
  .entrypoint
  // Code size       86 (0x56)
  .maxstack  2
  .locals ([0] valuetype Struct s,
           [1] class IInterface I)
  IL_0000:  ldloca.s   s
  IL_0002:  initobj    Struct
  IL_0008:  ldloc.0
  IL_0009:  box        Struct  // <- c'est ici que l'opération de boxing
                               //    de la structure a implicitement lieu.
  IL_000e:  stloc.1
  IL_000f:  ldloca.s   s
  IL_0011:  ldc.i4.s   10
  IL_0013:  call       instance void Struct::SetState(int32)
  IL_0018:  ldloc.1
  IL_0019:  ldc.i4.s   20
  IL_001b:  callvirt   instance void IInterface::SetState(int32)
  IL_0020:  ldstr      "Retour de s.GetState():"
  IL_0025:  ldloca.s   S
  IL_0027:  call       instance int32 Struct::GetState()
  IL_002c:  box        [mscorlib]System.Int32
  IL_0031:  call       string
                         [mscorlib]System.String::Concat(object,object)
  IL_0036:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_003b:  ldstr      "Retour de i.GetState():"
  IL_0040:  ldloc.1
  IL_0041:  callvirt   instance int32 I::GetState()
  IL_0046:  box        [mscorlib]System.Int32
  IL_004b:  call       string
                         [mscorlib]System.String::Concat(object,object)
  IL_0050:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0055:  ret
} // end of method Program::Main
```

# *Propriétés, événements et indexeurs virtuels et abstraits*

Les propriétés, les événements et les indexeurs d'une classe ont la possibilité d'être virtuels ou abstraits. En fait, l'idée sous-jacente est que ce sont les accesseurs de ces membres, considérés alors comme des méthodes, qui peuvent être virtuels ou abstraits. Rappelons que les accesseurs possibles pour une propriété ou un indexeurs sont `get` et `set` tandis que les accesseurs possibles pour un événement sont `add` et `remove`.

Comme les méthodes virtuelles et abstraites les propriétés, les événements et les indexeurs virtuels et abstraits peuvent être :

- (Re)définies dans une classe dérivée avec le mot-clé `override`. Dans ce cas le polymorphisme s'applique sur les accesseurs.
- (Re)définies dans une classe dérivée avec les mots-clés `override sealed`. Dans ce cas le polymorphisme s'applique sur les accesseurs, et le membre concerné ne peut être redéfini dans les classes dérivées de la classe dérivée.
- Redéfinies dans une classe dérivée avec le mot-clé `new`. Dans ce cas le polymorphisme ne s'appliquera pas sur les accesseurs.

Comme pour les méthodes abstraites, pour contenir une propriété, un événement ou un indexeur abstrait, une classe doit être abstraite. De plus comme les méthodes abstraites, un tel membre abstrait ne doit pas avoir une visibilité privée. Voici un exemple avec des propriétés virtuelles et abstraites :

*Exemple 12-14 :*

```
abstract class FooBase {
   protected int valA = 0 ;
   public virtual int Prop1 {
      get { return valA ; }
      set { valA = value ; }
   }
   public virtual int Prop2 {
      get { return 43 ; }
   }
   public abstract int Prop3 {
      get ;
      set ;
   }
}
class FooDeriv : FooBase {
   private int valB = 0 ;
   public override int Prop1 {
      get { return base.Prop1 * 2 ; }
      set { base.Prop1 = value * 2 ; }
   }
   public override sealed int Prop2 {
      get { return valA > valB ? valA : valB ; }
   }
   public override int Prop3 {
      get { return valA + valB ; }
      set { valA = value - valB ; }
   }
}
```

Il n'est pas nécessaire de faire de la classe `FooDeriv` une classe abstraite, puisqu'elle n'a pas de membres abstraits. Notez l'utilisation du mot-clé `base` pour appeler les accesseurs définis dans la classe de base.

# Les opérateurs is et as

C++ → C# La possibilité de permettre l'évaluation ou le transtypage du type d'une expression à l'exécution (permis en C# avec les opérateurs `is` et `as`) est implémentée en C++ avec l'opérateur `typeid(expression)` qui renvoie un objet de type `type_info`.

Cette possibilité rentre dans le cadre plus général du RTTI (RunTime Type Information en anglais, information de type à l'exécution en français).

Le RTTI est à la fois plus facile à utiliser en C# qu'en C++, et incomparablement plus complet, grâce aux métadonnées de type et au mécanisme de réflexion, présentés en page 233.

## L'opérateur is

L'opérateur `is` sert à déterminer à l'exécution si une expression peut être transtypée (castée) dans un type donné. Cet opérateur retourne un booléen. Son opérande de gauche est une expression et son opérande de droite un type.

Concrètement un objet de classe `A` dérivant d'une classe `B` et implémentant les interfaces `I1`, `I2`, ...`In`, peut être utilisé par l'intermédiaire :

- D'une référence de type `A`, `B`, `I1`, `I2`, ...`In`.
- D'une référence de type, une classe de base située dans la hiérarchie des classes de base de `B`.
- D'une référence de type, une interface supportée par une classe de base située dans la hiérarchie des classes de base de `A`.

Si la référence est nulle l'opérateur `is` retourne `false`. Voici un exemple d'utilisation du mot-clé `is` :

*Exemple 12-15 :*

```
using System ;
interface IA { void f(int i) ; }
interface IB { void g(int i) ; }
abstract class FooBase { public abstract void h(int i) ; }
class FooDeriv : FooBase, IA {
   public void f(int i) { /*...*/ }
   public override void h(int i){ /*...*/ }
}
class Program {
   static void Main() {
      IA refA = new FooDeriv() ;
      IB refB = null ;
      FooBase refAbst = null ;
      FooDeriv refC = null ;
      // Ici, le transtypage peut se faire.
      if ( refA is FooBase ){
         refAbst = (FooBase)refA ;
         // utilise refAbst...
      }
```

```
      // Ici, le transtypage peut se faire.
      if ( refA is FooDeriv ){
         refC = (FooDeriv)refA ;
         // utilise refC...
      }
      // Ici le transtypage ne peut pas se faire,
      if ( refA is IB ){
         refB = (IB)refA ;
         // utilise refB...
      }
      // Cette expression est toujours vraie si IA n'est pas nulle.
      // Le compilateur l'interpète comme : if( refA != null )
      if ( refA is IA ){ /*...*/ }
   }
}
```

Faites attention car bien souvent l'opérateur `is` est utilisé à tort, dans des situations où le polymorphisme aurait évité bien des tests.

## *L'opérateur as*

Après avoir déterminé à l'exécution si une expression peut être transtypée (*castée*) dans un type donné avec l'opérateur `is`, on réalise effectivement le transtypage la plupart du temps. L'opérateur `as` permet d'effectuer ces deux étapes d'un seul coup. Si le transtypage ne peut avoir lieu, la référence nulle est retournée. Il faut donc toujours tester la référence retournée.

L'avantage d'utiliser l'opérateur `as` au lieu de l'opérateur `is` lorsque ceci est possible est double : le code est plus lisible, les performances sont meilleures. Voici la méthode `Main()` de l'exemple de la section précédente, réécrite en utilisant l'opérateur `as` :

*Exemple 12-16 :*

```
...
class Program {
   static void Main() {
      IA refA = new FooDeriv() ;
      IB refB = null ;
      FooBase refAbst = null ;
      FooDeriv refDeriv = null ;
      // Ici, le transtypage peut se faire.
      refAbst = refA as FooBase ;
      if ( refAbst != null ) {
         // utilise refAbst...
      }
      // Ici, le transtypage peut se faire.
      refDeriv = refA as FooDeriv ;
      if ( refDeriv != null ) {
         // utilise refC...
      }
```

```
      // Ici, le transtypage ne peut pas se faire.
      refB = refA as IB ;
      if ( refB != null ) {
         // utilise refB...
      }
   }
}
```

# *Techniques de réutilisation de code*

L'héritage de classe n'est pas la seule solution pour réutiliser du code. En fait, j'ai rédigé un article disponible sur le site dotnetguru.org pour exprimer mon avis à ce sujet : *Faut il interdire l'héritage d'implémentation dans les langages objets ?* http://www.dotnetguru.org/articles/dossiers/heritageimpl/FragileBaseClass.htm.

Le fil directeur de cet article est que l'héritage d'implémentation est très souvent utilisé à mauvais escient. J'ai constaté que lorsque les développeurs sont formés, on leur présente ce mécanisme comme la pierre angulaire de la POO. La POO englobe d'autres principes tout aussi importants qui sont principalement l'encapsulation, la composition d'objet, l'abstraction avec les interfaces et la généricité. Mon article montre qu'avec un peu d'astuce, on peut utiliser conjointement les interfaces et la composition d'objet pour réutiliser du code d'une manière bien plus efficace qu'avec l'héritage d'implémentation.

Le chapitre suivant traite de la généricité. Ce mécanisme introduit avec C# 2.0 se révèle être un puissant outil pour réutiliser du code. Nous verrons qu'il est particulièrement adapté à l'écriture d'algorithmes génériques tels que les implémentations de collections qui ont besoin de stocker des objets instances d'une même classe, sans nécessairement cibler une classe en particulier.

Depuis quelques années, il y a une certaine effervescence autour de la *programmation orientée aspect* (*AOP* pour *Aspect Oriented Programming* en anglais). J'ai collaborer à la rédaction d'un article sur le sujet téléchargeable à l'URL : http://www.dotnetguru.org/articles/dossiers/aop/quid/AOP15.htm. L'AOP est une technique de réutilisation du code implémentant les aspects d'un logiciel (synchronisation, sécurité, persistance etc). À l'heure actuelle, *AspectDNG* constitue l'outil le plus avancé pour faire de l'*AOP* en .NET (*home page* : http://sourceforge.net/projects/aspectdng/).

Enfin, nous précisons que la version 3.0 de C# introduira un nouveau mécanisme de réutilisation de code permettant en quelques sortes d'étendre des classes avec des méthodes définies dans d'autres classes.

# 13

# *La généricité*

Sans conteste la *généricité* constitue la fonctionnalité phare de .NET 2005 au niveau des langages. Après avoir exposé en quoi consiste la généricité, nous examinerons les implications du support de la généricité, au niveau du langage C#2, du CLR et du *framework*. Sachez d'emblée que les types et les méthodes génériques sont *CLS compliant* et que par conséquent, ils sont supportés par tous les langages ciblant le CLR 2.0.

## *Un problème de C#1 et sa résolution par les types génériques de C#2*

### *Le problème du typage des éléments d'une collection en C#1*

Supposons que nous ayons à implémenter une classe `Stack` (pile en français) qui permet d'empiler et de dépiler des éléments. Pour simplifier, nous considérons que la pile ne peut contenir plus qu'un certain nombre d'éléments ce qui nous permet d'utiliser en interne un tableau C#. Voici une implémentation de la classe `Stack` qui satisfait ces contraintes :

*Exemple 13-1 :*

```
class Stack{
   private object[] m_ItemsArray ;
   private int m_Index = 0 ;
   public const int MAX_SIZE = 100 ;
   public Stack() { m_ItemsArray = new object[MAX_SIZE] ; }
   public object Pop() {
      if (m_Index ==0 )
         throw new System.InvalidOperationException(
          "Impossible de dépiler un élément d'une pile vide.") ;
```

```
      return m_ItemsArray[--m_Index] ;
   }
   public void Push( object item ) {
      if(m_Index == MAX_SIZE)
         throw new System.StackOverflowException(
          "Impossible d'empiler un élément sur une pile pleine.") ;
      m_ItemsArray[m_Index++] = item ;
   }
}
```

Cette implémentation souffre de trois défauts majeurs.

- Premièrement, les clients de la classe Stack doivent transtyper explicitement tout élément obtenu à partir de la pile. Par exemple :

```
...
Stack stack = new Stack() ;
stack.Push(1234) ;
int number = (int)stack.Pop() ;
...
```

- Un deuxième problème moins flagrant se situe au niveau des performances. Il faut être conscient que lorsque l'on utilise notre classe Stack avec des éléments de type valeur, nous réalisons implicitement une opération de boxing à l'insertion d'un élément et une opération de unboxing à la récupération d'un élément. Ce phénomène est mis en évidence par la version IL du client ci-dessus :

```
L_0000: newobj instance void Stack::.ctor()
L_0005: stloc.0
L_0006: ldloc.0
L_0007: ldc.i4 1234
L_000c: box int32
L_0011: callvirt instance void Stack::Push(object)
L_0016: nop
L_0017: ldloc.0
L_0018: callvirt instance object Stack::Pop()
L_001d: unbox int32
L_0022: ldind.i4
L_0023: stloc.1
L_0024: ret
```

- Enfin, un troisième problème vient du fait que l'on peut empiler des éléments de types différents dans une même instance de la classe Stack. Or, en général nous souhaitons avoir des piles d'éléments qui partagent un même type. Cette possibilité peut facilement mener à des problèmes de transtypages qui ne sont découverts qu'à l'exécution comme dans l'exemple ci-dessous :

```
...
Stack stack = new Stack() ;
stack.Push("1234");
int number = (int)stack.Pop() ; // Provoque une exception de type
```

```
                                    // InvalidCastException.
...
```

Lorsqu'un problème de transtypage n'est pas détecté à la compilation mais qu'il provoque une exception à l'exécution, on dit que le code n'est pas *type-safe*. Or, dans le développement logiciel comme dans toute discipline, plus une erreur est détectée tôt dans le processus de production moins elle est nuisible. Il faut donc dans la mesure du possible avoir du code type-safe puisque celui-ci permet la détection d'erreurs au plus tôt, lors de la compilation.

Il est possible d'implémenter notre concept de pile d'une manière *type-safe*. Nous pourrions en effet décider d'implémenter une classe `StackOfInt` pour décrire une pile contenant des entiers, une classe `StackOfSring` pour décrire une pile contenant des chaînes de caractères etc.

*Exemple 13-2 :*

```
class StackOfInt{
   private int[] m_ItemsArray ;
   private int m_Index = 0 ;
   public const int MAX_SIZE = 100 ;
   public StackOfInt(){m_ItemsArray = new int[MAX_SIZE];}
   public int Pop() { /*...*/  return -1 ; }
   public void Push(int item) { /*...*/ }
}
class StackOfString{
   private string[] m_ItemsArray ;
   private int m_Index = 0 ;
   public const int MAX_SIZE = 100 ;
   public StackOfString(){m_ItemsArray = new string[MAX_SIZE];}
   public string Pop() {/*...*/ return null ; }
   public void Push(string item) {/*...*/}
}
```

Bien qu'elle soit *type-safe* et qu'elle résolve à la fois le problème de transtypage et le problème de performance cette solution n'est clairement pas satisfaisante. Elle implique de la duplication de code puisque la logique d'une pile est implémentée par plusieurs classes. Les conséquences sont plus de code à maintenir et donc une baisse de la productivité.

## *Résolution élégante du problème à l'aide d'une classe générique de C♯2*

C♯2 permet de résoudre élégamment le problème de la section précédente grâce à l'introduction des *types génériques*. Concrètement, nous pouvons implémenter une liste d'éléments de type `T` en laissant la liberté aux clients de spécifier le type `T` lorsqu'ils instancient la classe. Par exemple :

*Exemple 13-3 :*

```
class Stack<T>{
   private T[] m_ItemsArray ;
   private int m_Index = 0 ;
   public const int MAX_SIZE = 100 ;
```

```
   public Stack(){ m_ItemsArray = new T[MAX_SIZE] ; }
   public T Pop(){
      if (m_Index ==0 )
         throw new System.InvalidOperationException(
          "Impossible de dépiler un élément d'une pile vide.") ;
      return m_ItemsArray[--m_Index] ;
   }
   public void Push(T item) {
      if(m_Index == MAX_SIZE)
         throw new System.StackOverflowException(
          "Impossible d'empiler un élément sur une pile pleine.") ;
      m_ItemsArray[m_Index++] = item ;
   }
}
class Program{
   static void Main(){
      Stack<int> stack = new Stack<int>();
      stack.Push(1234) ;
      int number = stack.Pop() ; // Plus besoin de casting.
      stack.Push(5678) ;
      string sNumber = stack.Pop() ;   // Erreur de compilation :
           // Cannot implicitly convert type 'int' to 'string'.
   }
}
```

Cette solution ne souffre d'aucun des problèmes vus précédemment :

- Le client n'a plus besoin de transtyper un élément récupéré de la pile.
- Cette solution est performante car elle n'entraîne aucune opération de boxing/unboxing.
- Le client écrit du code *type-safe*. Il n'a pas la possibilité d'avoir à l'exécution une pile d'éléments de types différents. Dans notre exemple, le compilateur interdit toute insertion ou récupération d'un élément d'un type différent que `int` ou qui n'est pas implicitement convertible en `int`.
- Il n'y a aucune duplication de code.

Comprenez bien que dans notre exemple la classe générique est `Stack<T>` alors que `T` est le type qui paramètre notre classe générique. On dit que `T` est un *type paramètre*. On utilise parfois le terme *polymorphisme paramétré* (*parametric polymorphism* en anglais) pour désigner la généricité. En effet, notre classe `Stack<T>` peut prendre plusieurs formes (`Stack<int>`, `Stack<string>` etc). Elle est donc polymorphe et paramétrée par un type. Attention, il ne faut pas confondre ceci avec le *polymorphisme* des langages objets qui permet de manipuler différentes formes d'objets (i.e des objets instances de classes différentes) au travers d'une même interface.

En bref, la classe `Stack<T>` représente n'importe quelle pile alors qu'une pile d'objets est une pile de n'importe quoi.

# *Vue d'ensemble de la généricité de C♯2*

## *Possibilité pour un type d'être générique sur plusieurs types*

Il peut être utile de paramétrer un type par plusieurs types. C♯2 présente cette possibilité. Par exemple, comme le montre l'exemple ci dessous, il est possible d'implémenter une classe dictionnaire qui laisse la possibilité aux clients de choisir le type des clés et le type des valeurs :

```
class DictionaryEntry<K,V>{
   public K Key ;
   public V Value ;
}
class Dictionary<K,V>{
   private DictionaryEntry<K,V>[] m_ItemsArray ;
   public void Insert(DictionaryEntry<K,V> entry) {...}
   public V Get(K key) {...}
   ...
}
```

## *Types génériques ouverts et fermés*

Un *type générique* (parfois aussi nommé *type construit*) est un type paramétré par un ou plusieurs autres types. Par exemple `Stack<T>`, `Stack<int>`, `Dictionary<K,V>`, `Dictionary<int,V>`, `Dictionary<int,string>` et `Stack<Stack<T>>` sont des types génériques.

Un *type générique fermé* (parfois aussi nommé *type construit fermé*) est un type générique pour lequel tous les types paramètres sont précisés : Par exemple `Stack<int>`, `Dictionary<int,string>` et `Stack<Stack<int>>` sont des types génériques fermés.

Un *type générique ouvert* (parfois aussi nommé *type construit ouvert*) est un type générique pour lequel au moins un type paramètre n'est pas précisé : Par exemple `Stack<T>`, `Dictionary<int,V>`, `Dictionary<K,V>`, et `Stack<Stack<T>>` sont des types génériques ouverts.

Un type générique se compile en un seul type dans son assemblage. Concrètement, si l'on analyse l'assemblage qui contient le type générique ouvert `Stack<T>`, on s'aperçoit que la compilation n'a produit qu'une seule classe indépendamment du fait qu'un client peut par exemple utiliser les types génériques fermés `Stack<int>`, `Stack<bool>`, `Stack<double>`, `Stack<string>`, `Stack<object>` et `Stack<IDispose>`.

En revanche, à l'exécution, le CLR crée et utilise plusieurs versions de la classe `Stack<T>`. Plus précisément, le CLR utilise une même version de `Stack<T>` commune à tous les types paramètres références et une version de `Stack<T>` pour chaque type paramètre valeur.

## *La généricité de .NET vs. le mécanisme de templates de C++*

C♯ → C++ Ceux qui connaissent le C++ auront certainement rapproché les types génériques de C♯ aux *templates* de C++. Bien que ces fonctionnalités soient conceptuellement proches, la présente section exhibe une différence fondamentale :

- Les types génériques fermés engendrés par les templates C++ sont produits par le compilateur C++ et sont contenus dans le composant produit de la compilation.

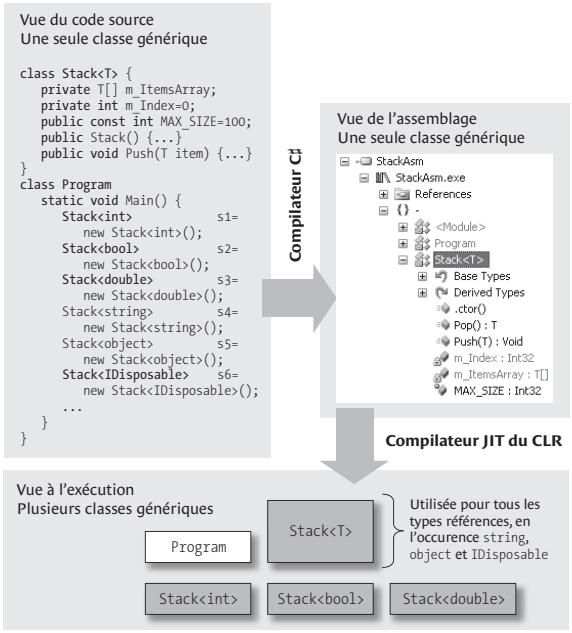


*Figure 13-1 : Interprétation d'un type générique selon le contexte*

- Les types génériques fermés engendrés par la généricité .NET sont produits à l'exécution par le compilateur JIT et le type générique sous-jacent n'est présent qu'en une seule version dans l'assemblage produit de la compilation.

Autrement dit, la notion de type générique ouvert existe en C#/.NET au niveau du code source, du composant et du runtime alors qu'en C++, elle n'existe qu'au niveau du code source.

Cette remarque souligne clairement un atout à la généricité de C# puisque la taille des composants .NET est d'autant réduite. Cela n'est pas négligeable puisque le phénomène de gonflement de la taille des composants C++, connu sous le nom de *code-bloat*, peut être parfois très pénalisant (sans compter l'avalanche d'avertissements produite par certains compilateurs C++). En outre le modèle de programmation par composant proposé par .NET est plus puissant avec cette implémentation de la généricité puisqu'un type générique ouvert peut être fermé par un type d'un autre composant.

Il peut y avoir cependant du *code-bloat* en .NET dans une moindre mesure. En effet, les types génériques fermés créés à l'exécution par le CLR ne sont jamais collectés ni par le ramasse-miettes ni par une autre entité du CLR. Ils résident dans leurs domaines d'application jusqu'à ce que celui-ci soit détruit. Dans certains cas rares résolubles en déchargeant à la main un domaine d'application, il peut donc y avoir un encombrement de la mémoire. Un bon point pour la généricité en .NET est qu'un type générique fermé n'est effectivement créé que le plus tard possible, lorsqu'il va être utilisé pour la première fois. De plus, il faut bien être conscient que le nombre de classes construites à l'exécution est forcément borné par le nombre de classes génériques fermées utilisées dans le code source.

Un problème similaire plus gênant survient lorsque l'on utilise l'outil `ngen.exe` pour améliorer les performances globales en effectuant le travail du compilateur JIT avant l'exécution. Dans ce cas, tous les types génériques fermés mentionnés dans votre code sources seront créés. L'outil `ngen.exe` est d'ailleurs incapable de distinguer si certains types génériques fermés mentionnés dans le code source ne seront jamais utilisés.

## *Visibilité d'un type générique*

La visibilité d'un type générique est l'intersection de la visibilité du type générique avec celles de ses types paramètres. Si les visibilités des types `C`, `T1`, `T2` et `T3` sont toutes égales à `public` alors la visibilité du type `C<T1,T2,T3>` est `public` ; mais si la visibilité d'un seul de ces types est `private`, alors la visibilité du type `C<T1,T2,T3>` est `private`.

Le lecteur astucieux subodore déjà que l'on peut obtenir un type générique avec une visibilité jusqu'ici inconnue en C♯ mais connue du CLR qui est `protected and internal` (visible seulement dans les classes dérivées situées dans le même assemblage, voir page 418). Sachez cependant qu'un tel type est forcément construit à l'exécution par le CLR, et cela n'entraînant donc aucune incohérence dans la langage C♯.

*Exemple 13-4 :*

```
internal class ClassInternal { }
public class ClassFoo{
   protected class ClassProtected { }
   public class ClassPublic<U,V> { }
   // Le compilateur vérifie que le type
   // ClassPublic<ClassInternal,ClassProtected> n'est pas utilisé
   // ailleurs que dans cette classe et dans ses classes dérivée
   // définie dans le même composant, mais vous ne pouvez fournir
   // une autre visibilité que private pour ce champ.
   private ClassPublic<ClassInternal,ClassProtected> foo ;
}
```

## *Structures et interfaces génériques*

En plus des classes génériques, C♯2 permet de définir des structures et des interfaces génériques. Ces possibilités n'ajoutent pas de remarques particulières mis à part le fait qu'un type ne peut implémenter plusieurs fois la même interface générique avec des types paramètres différents. Concrètement, le programme suivant ne compile pas :

*Exemple 13-5 :*

```
interface I<T> { void Fct() ; }
// Erreur de compilation :
// 'C<U,V>' cannot implement both 'I<U>' and 'I<V>' because they
// may unify for some type parameter substitutions.
class C<U, V> : I<U>, I<V>{
   void I<U>.Fct() { }
   void I<V>.Fct() { }
}
```

## *Possibilité de créer des alias sur le nom d'un type générique fermé*

La directive `using` peut être utilisée pour créer un alias sur le nom d'un type générique fermé. La portée d'une telle directive est le fichier courant si elle est utilisée hors de tous espace de noms, sinon la portée est l'intersection entre le fichier courant et l'espace de noms dans lequel l'alias est défini. Par exemple :

```
using Annuaire = Dictionary<TelephoneNumber, string>;
class TelephoneNumber { }
class Dictionary<K, V>{ }
...
Annuaire annuaire = new Annuaire();
```

# *Possibilité de contraindre un type paramètre*

C#2 présente la possibilité d'imposer des *contraintes* sur un type paramètre d'un type générique. Sans cette possibilité, la généricité de C#2 ne serait pratiquement pas exploitable. En effet, on ne peut pratiquement rien faire avec un type paramètre sur lequel on ne connaît rien. On ne sait même pas s'il est instanciable (puisqu'il peut prendre la forme d'une interface ou d'une classe abstraite). De plus, on ne peut pas appeler une méthode particulière sur une instance d'un tel type, on ne peut pas comparer les instances d'un tel type etc.

Pour pouvoir utiliser un type paramètre au sein d'un type générique, vous pouvez lui apposer une ou plusieurs contraintes parmi trois sortes de contraintes :

- La contrainte d'avoir un constructeur par défaut.
- La contrainte d'implémenter une certaine interface ou (non exclusif) de dériver d'une certaine classe.
- La contrainte d'être un type valeur ou (exclusif) un type référence.

C# → C++ Le mécanisme de template de C++ n'a pas besoin de contraintes pour exploiter les types paramètres puisque les types paramètres sont forcément résolus au moment de la compilation. Dans ce cas, toute tentative d'utilisation d'un membre absent est donc détectée à la compilation.

## *La contrainte du constructeur par défaut*

Si vous souhaitez pouvoir instancier un type paramètre au sein d'un type générique vous n'avez pas d'autre choix que de lui apposer une contrainte du constructeur par défaut. Voici un exemple qui illustre la syntaxe :

*Exemple 13-6 :*

```
class Factory<U> where U : new() {
   public static U GetNew() { return new U() ; }
}
class Program {
   static void Main(){
      int i = Factory<int>.GetNew() ;
      object obj = Factory<object>.GetNew() ;
```

```
      // Ici i vaut 0 et obj est une instance de object.
   }
}
```

## *Contraintes de dérivation*

Si vous souhaitez utiliser certains membres des instances d'un type paramètre au sein d'un type générique, vous devez lui apposer une contrainte de dérivation. Voici un exemple qui illustre la syntaxe :

*Exemple 13-7 :*

```
interface ICustomInterface { int Fct() ; }
class C<U> where U : ICustomInterface {
   public int AutreFct(U u) { return u.Fct(); }
}
```

Vous pouvez apposer plusieurs contraintes d'implémentation d'interfaces et une contrainte de dérivation d'une classe de base sur un même type paramètre. Le cas échéant, la classe de base doit apparaître en premier dans la liste des types. Vous pouvez aussi utiliser conjointement la contrainte du constructeur par défaut avec une ou plusieurs contraintes de dérivations. Dans ce cas la contrainte du constructeur par défaut doit apparaître en dernier :

*Exemple 13-8 :*

```
interface ICustomInterface1 { int Fct1() ; }
interface ICustomInterface2 { string Fct2() ; }
class BaseClass{}
class C<U>
   where U : BaseClass, ICustomInterface1,  ICustomInterface2, new() {
   public string Fct(U u) { return u.Fct2() ; }
}
```

Vous ne pouvez pas utiliser une classe `sealed` ou une des classes `System.Object`, `System.Array`, `System.Delegate`, `System.Enum` ou `System.ValueType` comme classe de base pour un type paramètre.

Vous ne pouvez pas non plus utiliser les membres statiques de `T` comme ceci :

*Exemple 13-9 :*

```
class BaseClass { public static void Fct(){} }
class C<T> where T : BaseClass {
   void F(){
      // Erreur de compilation : 'T' is a 'type parameter',
      // which is not valid in the given context.
      T.Fct();
   }
}
```

Un type utilisé dans une contrainte de dérivation peut être un type générique ouvert ou fermé. Illustrons cette possibilité avec l'interface `System.IComparable<T>`. Rappelons que les types qui implémentent cette interface peuvent voir leurs instances comparées à une instance de type `T`.

*Exemple 13-10 :*

```
class C1<U> where U : System.IComparable<int> {
   public bool Egaux(U u,int i) { return u.Equals(i) ; }
}
class C2<U> where U : System.IComparable<U> {
   public int Compare(U u1,U u2) { return u1.CompareTo(u2) ; }
}
class C3<U,V> where U : System.IComparable<V> {
   public int Compare(U u, V v) { return u.CompareTo(v) ; }
}
class C4<U, V> where U : System.IComparable<V>, System.IComparable<int>
{   public int Compare(U u, int i) { return u.CompareTo(i) ; }
}
```

Notez qu'un type utilisé dans une contrainte de dérivation doit avoir une visibilité égale ou supérieure à celle du type générique qui contient le type paramètre concerné. Par exemple :

*Exemple 13-11 :*

```
internal class BaseClass{}
// Erreur de compilation : Inconsistent accessibility :
// constraint type 'BaseClass' is less accessible than 'C<T>'
public class C<T> where T : BaseClass{}
```

Pour pouvoir être exploitées dans un type générique, certaines fonctionnalités peuvent vous obliger à apposer certaines contraintes de dérivation. Par exemple si vous souhaitez utiliser un type paramètre `T` dans une clause `catch`, vous devez contraindre `T` à dériver de la classe `System.Exception` ou d'une de ses classes dérivées. De même si vous souhaitez utiliser le mot-clé `using` pour disposer automatiquement une instance d'un type paramètre, celui-ci doit être contraint d'implémenter l'interface `System.IDisposable`. Enfin, si vous souhaitez utiliser le mot-clé `foreach` pour énumérer les éléments d'une instance d'un type paramètre, celui-ci doit être contraint d'implémenter une des deux interfaces `System.Collections.IEnumerable` ou `System.Collections.Generic.IEnumerable<T>`.

Notons enfin que dans le cas particulier où `T` est contraint d'implémenter une interface et `T` est un type valeur, l'appel d'un membre de l'interface sur une instance de `T` ne provoque pas de boxing. L'exemple suivant met en évidence ce phénomène :

*Exemple 13-12 :*

```
interface ICompteur{
   void Increment() ;
   int Val{get;}
}
struct Compteur : ICompteur {
   private int i ;
   public void Increment() { i++ ; }
   public int Val { get { return i ; } }
}
class C<T> where T : ICompteur, new() {
   public void Fct(){
```

```
      T t = new T() ;
      t.Increment() ;   // Modifie l'état de t.
      System.Console.WriteLine( t.Val.ToString() ) ;
      // Modifie l'état d'une copie boxée de t.
      (t as ICompteur).Increment() ;
      System.Console.WriteLine(t.Val.ToString()) ;
   }
}
class Program {
   static void Main() {
      C<Compteur> c = new C<Compteur>() ;
      c.Fct() ;
   }
}
```

Ce programme affiche :

```
1
1
```

## *La contrainte type valeur/type référence*

La contrainte type valeur/type référence permet de contraindre un type paramètre à être un type valeur ou un type référence. Cette contrainte, qui doit être utilisée en premier dans la liste des contraintes sur un type paramètre donné, utilise les mots-clés `struct` pour contraindre un type valeur et `class` pour contraindre un type référence. Attention, cette syntaxe peut prêter à confusion puisque les classes représentent un sous ensemble des types références (il y a aussi les interfaces) et les structures représentent un sous ensemble des types valeurs (il y a aussi les énumérations). Cette contrainte peut être utile dans certains cas particuliers où l'on souhaite utiliser des tests de nullité de références (une instance de type valeur ne peut jamais être nulle) ou lorsque l'on veut s'assurer qu'un type paramètre utilisé avec le mot-clé `lock` est de type référence.

*Exemple 13-13 :*

```
class C<U> where U : class, new () {
   U u = new U() ;
   void Fct(){ lock(u){ } }
}
```

# *Les membres d'un type générique*

## *Surcharge de méthode*

Les propriétés, les constructeurs, les méthodes et les indexeurs peuvent être surchargés dans une classe générique. Cependant il peut y avoir ambiguïté lorsqu'une certaine combinaison des types paramètres amène plusieurs surcharges à avoir une même signature. Dans ce cas, la préférence ira à la surcharge qui a la signature avec le moins de types paramètres. Si une telle méthode ne peut être trouvée, alors le compilateur émet une erreur au niveau de l'appel ambiguë. Voici un exemple pour clarifier ces règles :

*Exemple 13-14 :*

```
interface I1<T> {}
interface I2<T> {}
class C1<U> {
   public void Fct1(U u){} // Cette fct ne peut être appelée si U est int.
   public void Fct1(int i){}
   public void Fct2(U u1, U u2){}       // Pas d'ambiguité.
   public void Fct2(int i, string s){}
   public void Fct3(I1<U> a){}          // Pas d'ambiguité.
   public void Fct3(I2<U> a){}
   public void Fct4(U a){}              // Pas d'ambiguité.
   public void Fct4(U[] a){}
}
class C2<U,V> {
   public void Fct5(U u, V v){}     // Possibilité d'ambiguité si
   public void Fct5(V v, U u){}     // le type U = le type V.
   public void Fct6(U u, V v){}     // Possibilité d'ambiguité si
   public void Fct6(V v, U u){}     // le type U = le type V != int .
   public void Fct6(int u, V v){}
   public void Fct7(int u, V v){}   // Possibilité d'ambiguité si
   public void Fct7(U u, int v){}   // le type U = le type V = int.
   public void Fct8(U u, I1<V> v){} // Possibilité d'ambiguité
   public void Fct8(I1<V> v, U u){} // par exemple pour c2<I1<int>,int>.
   public void Fct9(U u1, I1<V> v2){} // Pas d'ambiguité.
   public void Fct9(V v1, U u2){}
   public void Fct10(ref U u){}       // Pas d'ambiguité.
   public void Fct10(out V v){ v = default(V) ; }
}
class Program {
   static void Main(){
      C1<int> a = new C1<int>() ;
      a.Fct1(34) ; // Appelle   Fct1(int i)
      C2<int, int> b = new C2<int, int>() ;
      b.Fct5(13, 14) ;  // Erreur de compilation : This call is ambiguous.
      b.Fct6(13, 14) ;  // Appelle Fct6(int u, V v)
      b.Fct7(13, 14) ;  // Erreur de compilation : This call is ambiguous.
      C2<I1<int>,int> c = new C2<I1<int>,int>() ;
      c.Fct8(null,null) ; //Erreur de compilation:This call is ambiguous.
   }
}
```

## *Les champs statiques*

Lorsqu'un type générique contient un champ statique, celui-ci existe à l'exécution en autant de versions qu'il y a de types génériques fermés fabriqués à partir du type générique concerné. Cette règle s'applique indépendamment du fait que le type du champ statique est fonction d'un type paramètre ou pas. Cette règle s'applique aussi indépendamment du fait que les types paramètres des types génériques fermés sont des types valeurs ou références. Cette dernière remarque est

pertinente car le fait que les types génériques fermés ayant des types paramètres références se partagent la même implémentation à l'exécution amène à se poser la question. Tout ceci est illustré par l'exemple suivant :

*Exemple 13-15 :*

```
using System ;
class C<T> {
   private static int m_NInst = 0;
   public C() { m_NInst++ ; }
   public int NInst { get { return m_NInst ; } }
}
class Program {
   static void Main() {
      C<int>    c1 = new C<int>() ;
      C<int>    c2 = new C<int>() ;
      C<int>    c3 = new C<int>() ;
      C<string> c4 = new C<string>() ;
      C<string> c5 = new C<string>() ;
      C<object> c6 = new C<object>() ;
      Console.WriteLine( "NInst C<int>   : " + c1.NInst.ToString() ) ;
      Console.WriteLine( "NInst C<string> : " + c4.NInst.ToString() ) ;
      Console.WriteLine( "NInst C<object> : " + c6.NInst.ToString() ) ;
   }
}
```

Ce programme affiche :

```
NInst C<int>   : 3
NInst C<string> : 2
NInst C<object> : 1
```

## *Les méthodes statiques*

Un type générique peut avoir des méthodes statiques. Dans ce cas il est obligatoire de résoudre les types paramètres lors de l'invocation d'une telle méthode. Par exemple :

*Exemple 13-16 :*

```
class C<T> {
   private static T t ;
   public static void ChangeState(T t_){ t = t_ ; }
}
class Program {
   static void Main() {
      C<int>.ChangeState(5);
   }
}
```

La méthode statique `Main()`, point d'entrée d'un programme, ne peut être dans une classe générique.

## Le constructeur statique

Si un type générique contient un constructeur statique, celui-ci est appelé par le CLR à chaque création d'un de ses types génériques fermés. Nous pouvons exploiter cette propriété pour ajouter nos propres contraintes sur les types paramètres. Par exemple, on ne peut pas strictement contraindre un type paramètre à ne pas être le type `int`. On peut donc profiter du constructeur statique pour vérifier une telle contrainte comme ceci :

*Exemple 13-17 :*

```
using System ;
class C<T> {
   static C() {
      int a=0;
      if( ((object) default(T) != null) && a is T )
         throw new ArgumentException("Ne pas utiliser le type C<int>.") ;
   }
}
```

Notez le test de la non nullité de la valeur par défaut de `T`. En effet, l'expression `(a is T)` est vrai lorsque `T` est le type `object` et lorsque `T` est le type `int`. Pour éliminer le premier cas, nous comptons sur le fait que l'expression `(object)defaut(object)` renvoie la valeur nulle.

## Surcharge des opérateurs

Bien que cela puisse mener à du code peu lisible, un type générique peut surcharger les opérateurs. Il n'y a pas de remarques particulières concernant les opérateurs arithmétiques et les opérateurs de comparaisons.

En revanche, lorsque l'on définit un opérateur de conversion (i.e opérateur de transtypage) d'un type source `Src` vers un type destination `Dest`, le compilateur ne doit pas pouvoir trouver de relation d'héritage entre les deux types au moment où le type générique est compilé. Par exemple :

*Exemple 13-18 :*

```
class C<T>{}
class D<T> : C<T>{
   public static implicit operator C<int>(D<T> val) {}  // OK
   // Erreur de compilation :  'D<T>.implicit operator C<T>(D<T>)' :
   // user-defined conversion to/from base class.
   public static implicit operator C<T>(D<T> val) {}
}
class Program{
   static void Main() {
      D<int> dd = new D<int>() ;  // OK
   }
}
```

Une conséquence du fait que l'on peut redéfinir certains opérateurs de conversion dans un type générique est qu'il devient possible de redéfinir certains opérateurs de conversions de types prédéfinis. Dans l'exemple suivant, si le type paramétré `U` est le type `objet` nous redéfinissons l'opérateur implicite de conversion de `D<object>` vers `object` :

```
class D<U> {
   public static implicit operator U(D<U> val) { return default(U) ; }
}
```

Dans ce cas deux règles sont appliquées par le CLR :

- Si une conversion implicite prédéfinie existe du type `Src` vers le type `Dest`, alors toute redéfinition (implicite ou explicite) de cette conversion est ignorée.
- Si une conversion explicite prédéfinie existe du type `Src` vers le type `Dest`, alors toute redéfinition de cette conversion est ignorée. En revanche, les redéfinitions implicites de la conversion du type `Src` vers le type `Dest` sont utilisées.

## *Les types encapsulés*

Un type encapsulé dans un type générique est implicitement un type générique. Les types paramètres du type générique encapsulant peuvent être librement utilisés au sein du type encapsulé. Un type encapsulé dans un type générique a la possibilité d'avoir ses propres types paramètres. Dans ce cas il y aura un type encapsulé générique fermé construit par le CLR pour chaque combinaison différente utilisée de l'ensemble des types paramètres.

*Exemple 13-19 :*

```
using System ;
class Outer<U>{
   static Outer(){Console.WriteLine("Hello du .cctor de Outer.");}
   public class Inner<V>{
      static Inner(){Console.WriteLine("Hello du .cctor de Inner.");}
   }
}
class Program{
   static void Main() {
      Outer<string>.Inner<int> a = new Outer<string>.Inner<int>();
      Outer<int>.Inner<int> b = new Outer<int>.Inner<int>();
   }
}
```

Ce programme affiche :

```
Hello du .cctor de Inner.
Hello du .cctor de Inner.
```

# *Les opérateurs et les types génériques*

## *Utilisation des opérateurs d'égalité, d'inégalité et de comparaison avec une instance d'un type paramètre*

Les opérateurs d'égalité et d'inégalité ne peuvent être utilisé avec une instance ou une référence d'un type paramètre `T` que dans les cas suivants :

- Si T a une contrainte de dérivation d'une classe ou T a une contrainte de type référence, alors les opérateurs d'égalité et d'inégalité peuvent être utilisés entre une référence de type T et n'importe qu'elle référence.
- Si T n'a pas de contrainte de type valeur, alors les opérateurs d'égalité et d'inégalité peuvent être utilisés entre une référence de type T et la référence null. Si T prend la forme d'un type valeur, le test d'égalité sera faux et le test d'inégalité sera vrai.

Exposons ces règles au moyen du programme suivant :

*Exemple 13-20 :*

```
class C<T,U,V> where T : class   where V :struct {
   public void Fct1( T t , U u , V v , object o, int i) {
      if (t == o) { } // OK
      if (u == o) { } // Erreur de compilation
      if (v == o) { } // Erreur de compilation
      if (v == i) { } // Erreur de compilation
      if (u == null) { } // OK
      if (v == null) { } // Erreur de compilation
   }
   public void Fct2(T t1, U u1, V v1, T t2, U u2, V v2) {
      if (t1 == t2) { } // OK
      if (u1 == u2) { } // Erreur de compilation
      if (v1 == v2) { } // Erreur de compilation
   }
}
```

Les opérateurs de comparaisons ne peuvent jamais être utilisés avec une instance ou une référence d'un type paramètre T.

## *L'opérateur typeof*

L'opérateur typeof utilisé sur un type paramètre retourne l'instance du type Type correspondant à la valeur courante du type paramètre.

L'opérateur typeof utilisé sur un type générique retourne l'instance du type Type correspondant à l'ensemble des valeurs courantes des types paramètres.

Ce comportement n'est pas flagrant puisque la propriété Name des types retournés n'affiche pas les noms des types paramètres.

*Exemple 13-21 :*

```
class C<T>{
   public static void PrintTypes(){
      System.Console.WriteLine( typeof(T).Name ) ;
      System.Console.WriteLine( typeof(C<T>).Name ) ;
      System.Console.WriteLine( typeof(C<C<T>>).Name ) ;
      if(typeof(C<T>) != typeof(C<C<T>>))
         System.Console.WriteLine("Malgré un nom similaire ce ne sont" +
                                  " pas les mêmes instances de Type.") ;
   }
```

```
}
class Program {
   static void Main() {
      C<string>.PrintTypes() ;
      C<int>.PrintTypes() ;
   }
}
```

Ce programme affiche :

```
String
C‘1
C‘1
Malgré un nom similaire ce ne sont pas les mêmes instances de Type.
Int32
C‘1
C‘1
Malgré un nom similaire ce ne sont pas les mêmes instances de Type.
```

Il n'en est pas de même si l'on utilise la propriété `FullName` :

*Exemple 13-22 :*

```
...
   public static void PrintTypes(){
      System.Console.WriteLine( typeof(C<T>).FullName ) ;
      System.Console.WriteLine( typeof(C<C<T>>).FullName ) ;
   }
...
```

Ce programme affiche :

```
C‘1[[System.String, mscorlib, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089]]
C‘1[[C‘1[[System.String, mscorlib, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089]], AsmTest, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null]]
C‘1[[System.Int32, mscorlib, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089]]
C‘1[[C‘1[[System.Int32, mscorlib, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089]], AsmTest, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null]]
```

## *Les mots clés params et lock*

Un type paramètre peut être utilisé comme type pour un paramètre `params` de la signature d'une méthode d'un type générique.

Le mot clé `lock` peut être utilisé avec une variable d'un type paramètre. Cette possibilité présente un danger lorsque le type paramètre est un type valeur. Il faut bien être conscient que dans ce cas, le mot clé `lock` n'aura aucun effet. Il est d'ailleurs assez surprenant que le compilateur ne force pas un type paramètre utilisé dans une clause `lock` à avoir une contrainte qui le force à être de type référence.

## *L'opérateur default*

Dans l'exemple de la pile, nous avons considéré l'opération `Pop()` sur une pile vide comme une erreur d'utilisation de la classe `Stack<T>` de la part du client (i.e une violation du contrat présenté par une pile). Nous aurions pu affaiblir le contrat et considérer cette opération comme un événement possible. Dans le premier cas, lancer une exception est le traitement adapté. Dans le second, il serait plus judicieux de retourner un élément vide que le client interprétera comme : il n'y a plus d'élément dans ma pile. Cependant nous ne connaissons rien du type `T` de l'élément à retourner. Si `T` est un type référence nous souhaiterions retourner une référence nulle alors que si `T` est le type `int` nous souhaiterions peut être retourner 0. Le mot clé `default` de C♯2 permet d'obtenir la valeur par défaut d'un type, i.e la référence nulle pour un type référence ou un block de mémoire de la taille adéquate mis à 0 pour un type valeur.

*Exemple 13-23 :*

```
class Stack<T>{
   ...
   public T Pop(){
      if (m_Index == 0)
         return default(T);
      return m_ItemsArray[--m_Index] ;
   }
   ...
}
```

La notion de type nullable constitue une manière plus élégante de définir la valeur par défaut d'un type valeur.

# *Le transtypage (casting) et la généricité*

## *Les règles de base*

Dans la suite, nous supposerons que `T` est un type paramètre. Le compilateur C♯2 accepte de :

- Transtyper implicitement une instance d'un type `T` (si `T` est de type valeur sinon une référence de type `T`) vers une référence de type `objet`. Si `T` est de type valeur, il y a une opération de boxing.
- Transtyper explicitement une référence de type `objet` vers une instance d'un type `T`. Si `T` est de type valeur, il y a une opération de unboxing.
- Transtyper explicitement une instance d'un type `T` vers une référence de type une interface quelconque. Si `T` est de type valeur, il y a une opération de boxing.
- Transtyper explicitement une référence de type une interface quelconque vers une instance d'un type `T`. Si `T` est de type valeur, il y a une opération de boxing.
- Dans les trois derniers cas, si le transtypage est impossible, une exception de type `InvalidCastException` est lancée.

D'autres règles de transtypage s'ajoutent si l'on utilise des contraintes de dérivations :

- Si T est contraint d'implémenter l'interface I, vous pouvez transtyper implicitement une instance de T en I ou en toute interface implémentée par I et vice versa. Si T est de type valeur, il y a une opération de boxing (ou de unboxing).
- Si T est contraint de dériver de la classe C, vous pouvez transtyper implicitement une instance de T en C ou en toute sous-classe de C et vice versa. Si une conversion propriétaire implicite existe de C vers un type A alors le compilateur accepte une conversion implicite de T vers A. Si une conversion propriétaire explicite existe de A vers C alors le compilateur accepte une conversion explicite de A vers T.

## *Transtypage entre tableaux*

Si T est un type paramètre d'une classe générique et si T à la contrainte de dériver de C alors le compilateur C#2 accepte de :

- Transtyper implicitement un tableau de T en un tableau de C. Autrement dit, le compilateur C#2 accepte de transtyper implicitement une référence de type T[] vers une référence de type C[]. On dit que les tableaux de C# acceptent la *covariance* sur leurs éléments.
- Transtyper explicitement un tableau de C en un tableau de T. Autrement dit, le compilateur C#2 accepte de transtyper explicitement une référence de type C[] vers une référence de type T []. On dit que les tableaux de C# acceptent la *contravariance* sur leurs éléments.

Ces deux règles sont illustrées par l'exemple suivant :

*Exemple 13-24 :*

```
class C { }
class ClassGenerique<T> where T : C {
   T[] arrOfT = new T[10] ;
   public void Fct(){
      C[] arrOfC = arrOfT;
      T[] arrOfT2 = (T[]) arrOfC;
   }
}
```

Il n'y a pas de règles équivalentes si T est contraint d'implémenter une interface I. En outre, la covariance et la contravariance ne sont pas supportées sur les types paramètres d'une classe générique. Autrement dit, si la classe D dérive de la classe B, il n'existe pas de conversion implicite ou explicite entre une référence de type List<D> et une référence de type List<B>.

## *Les opérateurs is et as*

Pour éviter une exception de type InvalidCastException lorsque vous n'êtes pas certain d'une conversion de type impliquant un type paramètre T, il est conseillé d'utiliser l'opérateur is pour tester si la conversion est possible et l'opérateur as pour tenter de réaliser la conversion. Rappelons que l'opérateur as retourne la référence null si la conversion est impossible. Par exemple :

*Exemple 13-25 :*

```
using System.Collections.Generic ;
class C<T> {
   public void Fct(T t){
      int i = t as int ; // Erreur de compilation :
                         // The as operator must be used with a reference type.
      string s = t as string ;
      if( s!= null ) { /*...*/ }
      if( t is IEnumerable<int> ){
         IEnumerable<int> enumerable = t as IEnumerable<int> ;
         foreach( int j in enumerable) { /*...*/ }
      }
   }
}
```

# *L'héritage et la généricité*

## *Les différents cas*

Une classe non générique peut dériver d'une classe générique. Dans ce cas tous les types paramètres doivent être résolus :

```
class B<T> {...}
class D : B<double> {...}
```

Une classe générique peut dériver d'une classe générique. Dans ce cas il est optionnel de résoudre tous les paramètres. En revanche il est obligatoire de rappeler toutes les contraintes sur les types paramètres non résolus. Par exemple :

```
class B<T> where T : struct { }
class D1<T> : B<T> where T : struct  { }
class D2<T> : B<int> { } // Maladroit T est ici un type paramétre
                         // différent.
class D3<U,V> : B<int> { }
```

Enfin, sachez qu'une classe générique peut dériver d'une classe non générique.

## *Redéfinition d'une méthode virtuelle d'un type générique*

Une classe générique de base peut avoir des méthodes abstraites ou virtuelles qui utilisent ou non les types paramètres dans leur signature. Dans ce cas, le compilateur oblige les réécritures de telles méthodes dans les classes dérivées à utiliser les types paramètres adéquates. Par exemple :

*Exemple 13-26 :*

```
abstract class  B<T> {
   public abstract T Fct(T t) ;
}
class D1 : B<string>{
   public override string Fct( string t ) { return "hello" ; }
```

```
}
class D2<T> : B<T>{
   public override T Fct(T t) { return default (T) ; }
}
// Erreur de compilation :
// does not implement inherited abstract member 'B<U>.Fct(U)'
class D3<T, U> : B<U> {
   // Erreur de compilation : no suitable method found to override
   public override T Fct(T t) { return default(T) ; }
}
```

On profite de l'exemple pour souligner le fait qu'une classe générique peut aussi être abstraite. Cet exemple montre aussi le genre d'erreur de compilation que l'on obtient lorsque l'on nomme maladroitement les types paramètres.

Il est intéressant de noter que les types paramètres d'une classe générique dérivée peuvent être utilisés dans le corps d'une méthode virtuelle réécrite, même si la classe de base n'est pas générique.

*Exemple 13-27 :*

```
class B  {
   public virtual void Fct() { }
}
class D<T> : B where T : new(){
   public override void Fct() {
      T t = new T();
   }
}
```

Toutes les règles énoncées dans la présente section restent valables pour l'implémentation d'interfaces éventuellement génériques, par des classes ou des structures éventuellement génériques.

# *Les méthodes génériques*

## *Introduction*

Qu'elle soit définie dans un type générique ou non, qu'elle soit statique ou non, une méthode a la possibilité de définir ses propres types paramètres. À chaque invocation d'une telle méthode un type doit être fourni pour chaque type paramètre. On parle de *méthode générique*.

Les types paramètres propres à une méthode ne sont utilisables que dans le scope de la méthode (i.e valeur de retour + liste des arguments + corps de la méthode). Dans la classe `C2<T>` de l'exemple suivant, il n'y a pas de corrélation entre le type paramètre `U` de la méthode `Fct<U>()` et le type paramètre `U` de la méthode `FctStatic<U>()`.

Un type paramètre d'une méthode peut avoir le même nom qu'un type paramètre de la classe qui définie la méthode. Dans ce cas, le type paramètre de la classe est caché dans le scope de la méthode. Dans la méthode `C3<T>.Fct<T>()` de l'exemple suivant, le type paramètre `T` défini par la méthode cache le type paramètre `T` défini par la classe. Cette pratique est plutôt maladroite et le compilateur produit un avertissement lorsqu'il la détecte.

*Exemple 13-28 :*

```
class C1 {
   public U Fct<U>(U u) { return u ; }
}
class C2<T> {
   public U Fct<U>(U u) { return u ; }
   public static U FctStatic<U>(U u) { return u ; }
}
class C3<T> {
   // Avertissement de compilation : Type parameter ' T' has same
   // name as type parameter from outer type 'C3<T>'.
   public T Fct<T>(T t) { return t ; }
}
class Program {
   static void Main() {
      C1 c1 = new C1() ;
      c1.Fct<double>(3.4);
      C2<int> c2 = new C2<int>() ;
      c2.Fct<double>(3.4);
      c2.Fct<string>("hello");
      C3<int> c3 = new C3<int>() ;
      c3.Fct<double>(3.4) ;
   }
}
```

Cette possibilité n'est pas utilisable ni sur les opérateurs, ni sur les méthodes `extern` ni sur les méthodes particulières que constituent les accesseurs des propriétés, des indexeurs et des événements.

## *Méthodes génériques et contraintes*

Une méthode générique peut définir toutes sortes de contraintes pour chacun de ses types paramètres. La syntaxe est identique à celle de la définition de contraintes sur un type générique.

*Exemple 13-29 :*

```
class C {
   public int Fct<U>(U u) where U : class, System.IComparable<U> ,new(){
      if (u == null) return 0 ;
      U unew = new U() ;
      return u.CompareTo(unew) ;
   }
}
```

Bien évidemment, une méthode générique ne peut redéfinir l'ensemble des contraintes d'un type paramètre défini par sa classe.

## *Méthodes virtuelles génériques*

Les méthodes abstraites, virtuelles et d'interface peuvent être génériques. Dans ce cas, les réécritures de telles méthodes ne sont pas obligées de respecter le nom des types paramètres. Dans le

cas d'une réécriture d'une méthode générique virtuelle ou abstraite qui a des contraintes sur ses types paramètres, vous ne devez pas réécrire l'ensemble des contraintes. Dans le cas d'une implémentation d'une méthode d'interface qui a des contraintes sur ses types paramètres, vous devez réécrire l'ensemble des contraintes. Ces règles sont illustrées par l'exemple suivant qui compile sans erreurs ni avertissements :

*Exemple 13-30 :*

```
using System ;
abstract class B {
   public virtual A Fct1<A, C>(A a, C c) { return a ; }
   public abstract int Fct2<U>(U u) where U:class,IComparable<U>,new() ;
}
class D1 : B {
   public override X Fct1<X, Y>(X x, Y y) { return x ; }
   public override int Fct2<U>(U u)  { return 0 ; }
}
interface I {
   A Fct1<A, C>(A a, C b) ;
   int Fct2<U>(U u) where U : class, IComparable<U>, new() ;
}
class D2 : I {
   public X Fct1<X, Y>(X x, Y y) { return x ; }
   public int Fct2<U>(U u) where U : class, IComparable<U>, new()
   { return 0 ; }
}
```

## *Inférence des types paramètres selon les types des paramètres d'une méthode générique*

Lors de l'invocation d'une méthode générique, le compilateur C#2 a la possibilité d'inférer les types paramètres d'une méthode générique à partir des types des paramètres fournis. Le fait de fournir explicitement des types pour les types paramètres prévaut sur les règles d'inférences.

Les règles d'inférences ne tiennent pas comptes du type de la valeur de retour. En revanche le compilateur est capable d'inférer un type paramètre à partir du type des éléments d'un tableau. Le programme suivant illustre tout ceci :

*Exemple 13-31 :*

```
class C {
    public static U Fct1<U>() { return default(U) ; }
    public static void Fct2<U>(U u) { return ; }
    public static U Fct3<U>(U u) { return default(U) ; }
    public static void Fct4<U>(U u1, U u2) { return ; }
    public static void Fct5<U>(U[] arrayOfU) { return ; }
}
class Program {
   static void Main() {
      // Erreur de compilation : The type arguments for method
```

```
      // 'C.Fct1<U>()' cannot be inferred from the usage.
      string s = C.Fct1() ;
      // Erreur de compilation : Cannot implicitly convert type
      // 'System.IDisposable' to 'string'.
      string s = C.Fct1<System.IDisposable>() ;
      s = C.Fct1<string>() ; // OK
      C.Fct2("hello") ; // Infère : le type paramètre U est string.
      // Erreur de compilation : The type arguments for
      // method 'C.Fct2<U>(U)' cannot be inferred from the usage.
      C.Fct2(null) ;
      int i = C.Fct3(6) ; // Infère : le type paramètre U est int.
      double d = C.Fct3(6) ; // ATTENTION : Infère : le type paramètre
                             // U est int et non pas double.
      // Erreur de compilation : Cannot implicitly convert 'int'
      // to 'System.IDisposable'.
      System.IDisposable dispose = C.Fct3(6) ;
      // Infère le type paramètre U est string.
      C.Fct4("hello", "bonjour") ;
      // Erreur de compilation : The type arguments for method
      // 'C.Fct4<U>(U,U)' cannot be inferred from the usage.
      C.Fct4(5, "bonjour") ;
      C.Fct5(new int[6]) ; // Infère : le type paramètre U est int.
   }
}
```

## *Ambiguïté dans la grammaire de C♯2*

On trouve une ambiguïté dans la grammaire de C♯2 car les caractères inférieur « < » et supérieur « > » peuvent dans certains cas très précis être interprétés à la fois comme la définition d'une liste de types paramètres et deux utilisations de l'opérateur de comparaison. Ce cas extrême est illustré par l'exemple suivant :

*Exemple 13-32 :*

```
class C<U,V> {
    public static void Fct1() {
        int U = 6 ;
        int V = 7 ;
        int Fct2 = 9 ;
        Fct3(Fct2 < U, V > (20)) ; // Appelle Fct3(int)
        Fct3(Fct2 < U, V > 20) ;   // Appelle Fct3(bool,bool)
    }
    public static int Fct2<A, B>(int i) { return 0;}
    public static void Fct3(int i) { return ; }
    public static void Fct3(bool b1, bool b2) { return ; }
}
```

La règle est que lorsque le compilateur rencontre un tel dilemne, il analyse le caractère situé immédiatement après « > ». Si celui-ci est dans la liste suivante, alors le compilateur infère une liste de type paramètre :

```
( ) ] > : ; , . ?
```

# *Les délégués, les événements et la généricité*

## *Introduction*

Comme tous les types encapsulés, une délégation (i.e une classe dont les instances sont des délégués) peut exploiter les types paramètres du type qui l'encapsule :

*Exemple 13-33 :*

```
class C<T> {
    public delegate T GenericDelegate(T t);
    public static T Fct(T t) { return t ; }
}
class Program {
   static void Main() {
       C<string>.GenericDelegate genericDelegate = C<string>.Fct ;
       string s = genericDelegate("hello") ;
   }
}
```

Une délégation peut aussi définir ses propres types paramètres ainsi que leurs contraintes :

*Exemple 13-34 :*

```
public delegate U GenericDelegate<U>(U u) where U : class ;
class C<T> {
   public static T Fct(T t) { return t ; }
}
class Program {
   static void Main() {
      GenericDelegate<string> genericDelegate = C<string>.Fct ;
      string s = genericDelegate( "hello" ) ;
   }
}
```

## *Délégués génériques et méthodes génériques*

Lors d'une affectation d'une méthode générique à un délégué générique, le compilateur C#2 est capable d'inférer les types paramètres de la méthode générique à partir des types paramètres du délégué générique. Cette possibilité est illustrée par l'exemple suivant :

*Exemple 13-35 :*

```
delegate void GenericDelegateA<U>(U u) ;
delegate void GenericDelegateB(int i) ;
delegate U    GenericDelegateC<U>() ;
class Program {
   static void Fct1<T>(T t) { return ; }
```

```
   static T    Fct2<T>() { return default(T) ; }
   static void Main() {
      GenericDelegateA<string> d1 = Fct1 ; // Le compilateur infére
                                           // Fct1<string>.
      GenericDelegateB d2 = Fct1 ; // Le compilateur infére Fct1<int>.
      GenericDelegateC<string> d3 = Fct2<string> ; // OK mais pas
                                                   // d'inférence.
      // Erreur de compilation : The type arguments for
      // method 'Program.Fct2<T>()' cannot be inferred from the usage.
      GenericDelegateC<string> d4 = Fct2 ;
   }
}
```

Comme l'illustre cet exemple, il n'y a jamais d'inférence sur les types paramètres d'un délégué générique.

## *Contravariance, covariance, délégués et généricité*

Nous exposons ici une nouvelle fonctionnalité des délégués qui va nous être utile par la suite. En C#2, les délégués supportent la *contravariance* sur leurs arguments et la *covariance* sur leur type de retour. Cette possibilité est exposée ci dessous :

*Exemple 13-36 :*

```
class Base { }
class Derived : Base { }
delegate Base DelegateType ( Derived d ) ;
class Program{
    static Derived Handler ( Base b ) { return b as Derived ; }
    static void Main() {
        // Remarquez que la signature de la méthode 'Handler()' ne
        // satisfait pas la signature de la délégation 'DelegateType'.
        DelegateType delegateInstance = Handler;
        Base b = delegateInstance( new Derived() ) ;
    }
}
```

Il est légitime que ce programme compile. Réfléchissez en terme de contrat :

- La méthode `Handler(Base)` a un contrat moins contraignant sur ses entrées que celui proposé par la délégation `DelegateType(Derived)`. Une instance de `DelegateType` peut donc référencer la méthode `Handler()` sans risque de *downcasting* illégal. C'est la contravariance.
- La méthode `Derived Handler()` a un contrat plus contraignant sur ses sorties que celui proposé par la délégation `Base DelegateType()`. Là aussi, une instance de `DelegateType` peut donc référencer la méthode `Handler()` sans risque de *downcasting* illégal. C'est la covariance.

Supposons maintenant que ces possibilités de covariance et de contravariance n'existent pas et supposons que nous souhaitons appeler la méthode `Derived Handler(Base)` par l'intermédiaire d'un délégué de type `Base delegate(Derived)`. Nous pourrions tout à fait utiliser un délégué générique comme ceci :

*Exemple 13-37 :*

```
class Base { }
class Derived : Base { }
delegate B DelegateType<B,D>(D d) ;
class Program {
   static Derived Handler(Base b){return b as Derived;}
   static void Main() {
      DelegateType<Base, Derived> delegateInstance = Handler ;
// La référence en entrée est implictement castée de Derived vers Base.
// La référence en sortie est implictement castée de Derived vers Base.
      Base b = delegateInstance( new Derived() ) ;
   }
}
```

## *Événements et délégués génériques*

Les délégués génériques peuvent être utiles pour éviter de définir de multiples délégués pour typer les événements. L'exemple suivant montre qu'avec une seule délégation générique, vous pouvez typer tous les événements qui prennent en paramètre un « sender » et un argument :

*Exemple 13-38 :*

```
delegate void GenericEventHandler<U,V>(U sender, V arg) ;
class Publisher {
   public event GenericEventHandler<Publisher,System.EventArgs> Event ;
   public void TriggerEvent() { Event(this, System.EventArgs.Empty) ; }
}
class Subscriber {
   public void EventHandler(Publisher sender, System.EventArgs arg){}
}
class Program {
   static void Main() {
      Publisher publisher = new Publisher() ;
      Subscriber subscriber = new Subscriber() ;
      publisher.Event += subscriber.EventHandler ;
   }
}
```

# *Réflexion, attribut, IL et généricité*

## *Évolution de la classe System.Type*

Rappelons qu'une instance de `System.Type` s'obtient soit avec l'opérateur `typeof` soit en appelant la méthode `object.GetType()`. En .NET 2005, une instance de `System.Type` peut référencer un type générique ouvert ou fermé.

*Exemple 13-39 :*

```
using System ;
using System.Collections.Generic ;
class Program {
    static void Main() {
        List<int> list = new List<int>() ;
        Type type1 = list.GetType() ;
        Type type2 = typeof(List<int>) ;
        Type type3 = typeof(List<double>) ;
        // type4 représente un type générique ouvert
        Type type4 = type3.GetGenericTypeDefinition();
        System.Diagnostics.Debug.Assert(type1 == type2) ;
        System.Diagnostics.Debug.Assert(type1 != type3) ;
        System.Diagnostics.Debug.Assert(type3 != type4) ;
    }
}
```

La classe System.Type supporte de nouvelles méthodes et propriétés dédiées à la généricité :

```
public abstract class Type : System.Reflection.MemberInfo,
                             System.Runtime.InteropServices._Type,
                             System.Reflection.IReflect
{
   // Utilisé lors de la construction d'un type avec l'API Emit
   // pour ajouter des types generics paramètres.
   public virtual System.Type MakeGenericType(
                                params System.Type[] typeArgs) ;
   // Obtient les types paramètres qu'ils soient ouverts ou fermés.
   public virtual System.Type[] GetGenericArguments() ;
   // Obtient la forme ouverte d'un type générique
   public virtual System.Type GetGenericTypeDefinition() ;
   // Retourne true si contient des types paramètres non précisés.
   public virtual bool IsGenericTypeDefinition { get ; }
   // Retourne true si appelée sur un type générique ouvert.
   // Contrairement à IsGenericTypeDefinition la recherche d'un type
   // non précisé se fait récursivement sur les tous les types
   // paramètres précisés.
   public virtual bool ContainsGenericParameters { get ; }
   // Retourne true si appelée sur un type qui est un type paramètre
   // d'un type générique ou d'une méthode générique.
   public virtual bool IsGenericParameter { get ; }
   //-------------------------------------------------------------
   // Les membres suivants ne peuvent être appelée que sur les types
   // pour lesquels IsGenericParameter est true
   // Obtient la position (0 based) d'un type paramètre ouvert.
   public virtual int GenericParameterPosition { get ; }
   // Obtient la méthode générique qui déclare le type paramètre
   // ouvert, null si non déclarée dans une méthode générique.
   public virtual System.Reflection.MethodBase DeclaringMethod { get ; }
```

```
   // Obtient les contraintes de dérivations d'un type paramètre ouvert.
   public virtual System.Type[] GetGenericParameterConstraints() ;
   // Obtient les contraintes autres que celles de dérivation.
   public virtual System.GenericParameterAttributes
              GenericParameterAttributes { get ; }
   ...
}
```

Voici un exemple d'utilisation de cette classe pour retrouver la définition d'un type générique :

*Exemple 13-40 :*

```
using System ;
using System.Reflection ;
class Program {
   static void WriteTypeConstraints(Type type ){
      string[] results = new string[type.GetGenericArguments().Length] ;
      foreach (Type t in type.GetGenericArguments()) {
         if ( t.IsGenericParameter ) {
            int pos = t.GenericParameterPosition ;
            Type[] derivConstraints =
                        t.GetGenericParameterConstraints() ;
            MethodBase methodBase = t.DeclaringMethod ;
            GenericParameterAttributes attributes =
                        t.GenericParameterAttributes ;
            results[pos] = "   where " + t.Name + ":" ;
            if ((GenericParameterAttributes.ReferenceTypeConstraint &
                 attributes) != 0 ) {
               results[pos] += "class," ;
            }
            if((GenericParameterAttributes.
                NotNullableValueTypeConstraint & attributes) != 0 ) {
               results[pos] += "struct," ;
            }
            foreach (Type derivConstraint in derivConstraints) {
               results[pos] += derivConstraint.Name + "," ;
            }
            if ((GenericParameterAttributes.
                 DefaultConstructorConstraint & attributes) != 0 ) {
               results[pos] += "new()" ;
            }
         }
      }
      Console.WriteLine(type.Name) ;
      foreach (string result in results)
         if (result != null)
            Console.WriteLine(result) ;
   }
   class Bar{}
   class Foo : Bar, IDisposable{ public void Dispose() {} }
```

```
   class C<U, V>
       where U : Bar, IDisposable, new()
       where V : struct {}
   static void Main() {
      WriteTypeConstraints( typeof(C<Foo,int>) ) ;
      WriteTypeConstraints(
              typeof(C<Foo,int>).GetGenericTypeDefinition() ) ;
   }
}
```

Ce programme affiche :

```
C‘2
C‘2
   where U:Bar,IDisposable,new()
   where V:struct,ValueType,new()
```

On s'aperçoit que la contrainte d'être un type valeur oblige le compilateur à ajouter les contraintes de dérivation de `ValueType` et d'implémentation d'un constructeur par défaut.

## *Évolution des classes System.Reflection.MethodBase et System.Reflection.MethodInfo*

Les classes `System.Reflection.MethodBase` et `System.Reflection.MethodInfo` (qui dérive de `MethodBase`) ont évolué de façon à supporter le concept de méthode générique. Voici les nouveaux membres :

```
public abstract class MethodBase :
       System.Reflection.MemberInfo,
       System.Runtime.InteropServices._MethodBase {
   // Retourne un tableau de types contenant les types paramètres.
   public virtual System.Type[] GetGenericArguments() ;
   // Retourne true si contient des types paramètres ouverts.
   public virtual bool IsGenericMethodDefinition { get ; }
   // Retourne true si appelée sur une méthode générique ouverte.
   // Contrairement à IsGenericMethodDefinition la recherche d'un type
   // non précisé se fait récursivement sur les tous les types
   // paramètres précisés.
   public virtual bool ContainsGenericParameters { get ; }
   ...
}
public abstract class MethodInfo :
       System.Reflection.MemberBase,
       System.Runtime.InteropServices._MethodInfo {
   // Obtient une méthode générique fermée à partir d'une méthode
   // générique ouverte, en résolvant les types paramètres à partir
   // de typeArgs.
   public virtual System.Reflection.MethodInfo MakeGenericMethod(
                          params System.Type[] typeArgs) ;
   // Obtient une méthode générique ouverte à partir d'une méthode
```

```
   // générique fermée.
   public virtual System.Reflection.MethodInfo
                                        GetGenericMethodDefinition() ;
   ...
}
```

Le programme suivant montre comment se lier tardivement à une méthode générique et comment l'invoquer après avoir résolu les types paramètres :

*Exemple 13-41 :*

```
using System ;
using System.Reflection ;
class Program{
   public class Bar{}
   public class Foo : Bar, IDisposable{ public void Dispose() { } }
   public static void Fct<U, V>()
       where U : Bar, IDisposable, new()
       where V : struct {
      Console.WriteLine(typeof(U).Name);
      Console.WriteLine(typeof(V).Name);
   }
   static void Main() {
      Type typeProgram = typeof(Program) ;
      MethodInfo methodGenericOpen = typeProgram.GetMethod("Fct",
                     BindingFlags.Static | BindingFlags.Public) ;
      //  Résoud les types paramètres.
      MethodInfo methodGenericClosed =
         methodGenericOpen.MakeGenericMethod (
            new Type[] { typeof(Foo), typeof(int) } ) ;
      System.Diagnostics.Debug.Assert (
             methodGenericClosed.GetGenericMethodDefinition() ==
             methodGenericOpen ) ;
      methodGenericClosed.Invoke (
             null,  //  null car méthode statique -> pas d'instance
             new object[0]) ; // new object[0] car pas de paramètres
   }
}
```

Ce programme affiche :

```
Foo
Int32
```

## *Les attributs et la généricité*

Un attribut qui peut marquer une méthode quelconque peut aussi marquer une méthode générique.

Un attribut qui peut tagger un type quelconque peut aussi tagger un type générique.

L'énumération `System.AttributeTargets` a la nouvelle valeur `GenericParameter` qui permet de préciser qu'un attribut peut tagger un type paramètre. Par exemple :

*Exemple 13-42 :*

```
[System.AttributeUsage(System.AttributeTargets.GenericParameter)]
public class A : System.Attribute{}
class C<[A]U, V> { }
```

Une classe d'attribut ne peut être une classe générique.

Une classe générique ne peut dériver directement ou indirectement de la classe `System.Attribute`.

Une classe d'attribut peut utiliser des types génériques et définir des méthodes génériques :

*Exemple 13-43 :*

```
class C<U, V> { }
public class A : System.Attribute{
   void Fct1(C<int, string> c) { }
   void Fct2<X>() { }
}
```

## *La généricité et le langage IL*

Le support de la généricité implique des changements au niveau du CLR mais aussi, au niveau du langage IL, au niveau du CTS et au niveau des métadonnées contenues dans les assemblages.

Dans les corps des méthodes d'un type générique ou dans le scope d'une méthode générique, le langage IL utilise la notation `!x` pour nommer un type paramètre situé en position `x` (0 indexée) dans la liste des types paramètres.

De nouvelles instructions IL ont été rajoutée tel que `stelem.any` ou `ldelem.any` pour l'accès aux éléments d'un tableau d'éléments d'un type paramètre (elles viennent ainsi compléter la famille d'instructions `stelem.i2`, `ldelem.i2`, `stelem.i4`, `ldelem.i4`, `stelem.ref`, `ldelem.ref`...). Certaines instructions IL telles que `stloc.x` ou `ldloc.x` (qui permettent de manipuler les variables locales) ne tenaient déjà pas compte du type des valeurs manipulées. Elles étaient donc prêtes pour la généricité et seule leur interprétation par le compilateur JIT a évolué.

Seulement deux tables de métadonnées sont rajoutées dans la liste des tables de métadonnées d'un assemblage contenant des types ou des méthodes génériques :

- La table `GenericParam` permet de décrire les types paramètres.
- La table `GenericParamConstraint` permet de décrire les contraintes de dérivation.

Les contraintes type valeur/référence et constructeur par défaut ne sont pas stockées dans un attribut ou dans une table de métadonnées quelconque. Elles sont tout simplement contenues dans le nom des types paramètres dans le nom du type ou de la méthode générique. Ainsi les classes suivantes...

```
class C1<T> where T : new() {...}
class C2<T> where T : class {...}
class C3<T> where T : struct {...}
```

...sont nommées en IL :

```
... C1<(.ctor) T> {...}
... C2<(class) T> {...}
... C3<(value type, .ctor, [mscorlib]System.ValueType) T> {...}
```

# *La généricité et le framework .NET*

## *La sérialisation et la généricité*

Il est possible de sérialiser et désérialiser une instance d'un type générique. Dans ce cas, il est obligatoire que la liste des types paramètres du type générique de l'objet sérialisé soit strictement identique à la liste des types paramètres du type générique de l'objet désérialisé. Par exemple :

*Exemple 13-13 :*

```
using System ;
using System.Runtime.Serialization ;
using System.Runtime.Serialization.Formatters.Binary ;
using System.IO ;
[Serializable]
public class C<T>{
   private T m_t ;
   public T t { get { return m_t ; } set { m_t = value ; } }
}
class Program{
   static void Main() {
      C<int> objIn = new C<int>() ;
      objIn.t = 691 ;
      IFormatter formatter = new BinaryFormatter() ;
      Stream stream = new FileStream("obj.bin", FileMode.Create,
                                     FileAccess.ReadWrite) ;
      formatter.Serialize(stream, objIn) ;
      stream.Seek(0, SeekOrigin.Begin) ;
      C<int> objOut = (C<int>)formatter.Deserialize(stream) ;
      // Ici, objOut.t est égale à 691.
      // Cette ligne provoque l'envoi d'une exception de type
      // SerializationException.
      C<long> objOut2 = (C<long>)formatter.Deserialize(stream) ;
      stream.Close() ;
   }
}
```

## *.NET Remoting et la généricité*

Il est possible de consommer une instance d'un type générique fermé avec la technologie .NET Remoting, que vous soyez en mode CAO ou WKO :

```
// définition de la classe générique utilisable avec .NET Remoting
public class Serveur<T> : MarshalByRefObject{}
...
   // côté serveur
   RemotingConfiguration.RegisterActivatedServiceType(
      typeof(Serveur<int>)) ;
   RemotingConfiguration.RegisterWellKnownServiceType(
      typeof(Serveur<string>), "MonService",
```

```
      WellKnownObjectMode.SingleCall) ;
...
   // côté client
   RemotingConfiguration.RegisterActivatedClientType(
      typeof(Serveur<int>), url) ;
   RemotingConfiguration.RegisterWellKnownClientType(
      typeof(Serveur<string>), url) ;
```

Si vous souhaitez utiliser les fichiers de configuration coté client ou coté serveur, il faut impérativement préciser les types paramètres utilisés :

```
// côté serveur
   <service>
      <activated
       type="ServeurAssembly.Serveur[[System.Int32]],ServeurAssembly"/>
   </service>
// côté client
   <client url="...">
      <activated
       type="ServeurAssembly.Serveur[[System.Int32]],ServeurAssembly"/>
   </client>
```

La syntaxe avec *doubles crochets* (*double square brackets*) permet de préciser une liste de type paramètres :

```
type="ServeurAssembly.Serveur[[System.Int32],[System.String]],
      ServeurAssembly"
```

La classe `System.Activator` supporte aussi la généricité. Sachez juste que lorsque vous utilisez conjointement cette classe avec un type générique, vous ne pouvez pas utiliser les surcharges des méthodes `CreateInstance()` et `CreateInstanceFrom()` dans lesquelles vous devez préciser les noms des types dans des chaînes de caractères.

## *Les collections et la généricité*

L'ensemble des collections du *framework* .NET fait l'objet du chapitre 15. Cet ensemble a été complètement revu en tenant compte des bénéfices du support de la généricité. L'espace de noms `System.Collections` est supporté pour des raisons de compatibilité. Il n'y a plus aucune raison de préférer un type de `System.Collections` à un type de `System.Collections.Generic`. En page 595 nous exposons un tableau de correspondance entre les types des deux espaces de noms `System.Collections` et `System.Collections.Generic`.

## *Les domaines ne supportant pas la généricité*

Les notions de services web et de composants servis (i.e COM+, Entreprise Services) ne supportent pas le concept de type générique car ni les standards de définitions de service web, ni la technologie COM ne supporte la généricité.

# 14

# *Les mécanismes utilisables dans C♯*

Nous allons voir que C♯ permet de suspendre ponctuellement la gestion du code par le CLR pour permettre aux développeurs des accès mémoires directs à l'aide de pointeurs. Ainsi avec C♯, vous pouvez réaliser d'une manière standard certaines optimisations qui jusqu'ici n'étaient possibles que dans les environnements non gérés tels que C++. Ces optimisations concernent par exemple le traitement de données volumineuses en mémoire, tel que les bitmaps.

À l'instar des langages C++ et Java, C♯ propose un mécanisme de gestion d'exceptions simple et classique mais puissant.

Nous verrons enfin que le langage C♯2 ajoute deux possibilités syntaxiques proches du concept de programmation fonctionnelle qui, dans certains cas précis, améliorent grandement la lisibilité du code.

## *Les pointeurs et les zones de code non vérifiable*

C♯ → C++ C++ ne connaît pas la notion de gestion de code. Cela est un atout de C++, puisque ceci permet d'utiliser les pointeurs, donc d'écrire du code très proche de la machine, très optimisé.

Cela est aussi un désavantage de C++, puisque l'utilisation des pointeurs est pénible et dangereuse et allonge donc considérablement les durées de développement et de maintenance.

C♯ Avant la plateforme .NET, 100% du code exécuté sous les systèmes d'exploitation *Windows* était non géré. Concrètement, les exécutables contenaient directement le code compatible avec les instructions machines d'un type de processeur (i.e du code en langage machine). L'introduction du mode d'exécution géré avec l'architecture .NET est une révolution. En effet, les principales causes de bug difficilement identifiables sont systématiquement détectées et signalées, voire résolues, par le CLR. Parmi ces causes citons :

- Les débordements d'accès aux éléments d'un tableau. (Maintenant géré dynamiquement par le CLR).

- Les *fuites de mémoire* (*memory leak* en anglais). (Maintenant géré pour la plupart, par le ramasse-miettes).
- L'utilisation d'un pointeur invalide. Cette contrainte est résolue d'une manière radicale : LA MANIPULATION DE POINTEURS EST INTERDITE EN MODE GÉRÉ.

Cependant, lors de la présentation du CTS en page 342 nous avons montré que le CLR sait manipuler trois sortes de pointeurs :

- Les *pointeurs gérés*. Ces pointeurs peuvent pointer vers une donnée contenue dans le tas des objets gérés par le ramasse-miettes. Ces pointeurs ne peuvent pas être utilisés explicitement par du code C#. Ils sont cependant exploités implicitement par le compilateur C# lorsque ce dernier compile des méthodes avec des arguments `out` et `ref`.
- Les *pointeurs non gérés de fonction*. La section en page 272 expose l'utilisation de ces pointeurs.
- Les *pointeurs non gérés*. Ces pointeurs peuvent pointer vers toute donnée contenue dans la zone d'adressage utilisateur du processus. Le langage C# permet d'utiliser cette sorte de pointeur dans un mode d'exécution spécialement prévu à cet effet : le mode d'exécution *non vérifiable* (parfois nommé mode d'exécution *non protégé*). Le code IL émis par le compilateur C# correspondant aux zones où vous manipulez des pointeurs non gérés contient des instructions IL spéciales. Leur effet sur la mémoire du processus ne peut être vérifié par le compilateur JIT du CLR (d'où le terme zone de code non vérifiable). En conséquence, un individu mal intentionné peut profiter des zones de code non vérifiables pour effectuer des actions malicieuses. Pour pallier cette faiblesse, à l'exécution le CLR ne se permet d'exécuter du code non vérifiable que si ce code a la méta permission CAS `SkipVerification`.

Puisqu'il permet de manipuler directement la mémoire du processus par l'intermédiaire de pointeurs non gérés, le code non vérifiable est particulièrement utile pour optimiser certains traitements de données volumineuses stockées dans des structures. Un exemple d'optimisation de traitement d'image, grâce aux pointeurs, est disponible en page 699.

## *Option de compilation pour le mode d'exécution non vérifiable*

Le code non vérifiable doit être utilisé en toute connaissance de cause. Aussi, vous devez obligatoirement fournir l'option de compilation `/unsafe` au compilateur `csc.exe` pour lui indiquer que vous êtes bien conscient que le code que vous demandez de compiler contient des zones qui seront non vérifiables par le compilateur JIT. *Visual Studio* présente la propriété de projet *Build* ► *Allow unsafe code* pour savoir s'il doit utiliser cette option de compilation.

## *Déclaration d'une zone de code non vérifiable*

En C# le mot-clé `unsafe` annonce au compilateur une zone de code non vérifiable. Il peut être utilisé dans trois situations :

- Devant la déclaration d'une classe ou d'une structure. Dans ce cas le code de toutes les méthodes (statiques ou non) de la classe peut utiliser des pointeurs.
- Devant la déclaration d'une méthode (statique ou non). Dans ce cas les pointeurs peuvent être utilisés dans tout le corps de la méthode.

- À l'intérieur du corps d'une méthode (statique ou non). Dans ce cas les pointeurs peuvent être utilisés dans le bloc de code signalé. Par exemple :

```
unsafe{
   ...
}
```

Précisons que si une méthode accepte au moins un pointeur en argument ou en retour, il faut que la méthode (ou sa classe) soit non vérifiable, mais aussi que toutes les zones de codes appelant cette méthode soient non vérifiables.

# *Manipulation des pointeurs en C♯*

C++ → C♯ La syntaxe et l'utilisation des pointeurs sont identiques en C♯ et en C++, mis à part les points particuliers suivants : en C♯, la déclaration `int *p1,p2;` fait que `p1` est un pointeur sur un entier et `p2` est un entier.

Seuls certains types peuvent être pointés.

En C♯ il est nécessaire d'épingler en mémoire les objets auxquels on veut accéder avec des pointeurs.

C♯ Chaque objet, qu'il soit de type valeur ou référence, admet une adresse mémoire qui le localise physiquement dans le processus. Cette adresse n'est pas nécessairement constante au cours de la vie de l'objet puisque le ramasse-miettes peut déplacer physiquement les objets qui sont stockés sur le tas.

## *Les types que l'on peut pointer*

Pour certains types, il existe un type dual, le type pointeur non géré du type concerné. Une variable de type pointeur est en fait l'adresse d'une variable du type concernée. L'ensemble des types qui autorisent l'utilisation de pointeurs se limite à tous les types valeur, mis à part les structures ayant au moins un champ de type référence. En conséquence, seules les instances des types suivants peuvent être pointées :

- les types primitifs,
- les énumérations,
- les structures n'ayant aucun champ de type référence,
- les pointeurs.

## *Déclaration des pointeurs*

Un pointeur peut ne pointer sur rien du tout. Dans ce cas il est **extrêmement important** que sa valeur soit nulle (0). En effet la majorité des bugs dus aux pointeurs viennent de pointeurs non nuls, qui ne pointent sur aucune donnée valide.

La déclaration d'un pointeur sur le type `FooType` se fait comme suit :

```
FooType * pointeur ;
```

Par exemple :

```
long * pUnEntier = 0 ;
```

Notez que la déclaration ...

```
int * p1,p2 ;
```

...fait que `p1` est un pointeur sur un entier et `p2` est un entier.

## *Opérateurs d'indirection et de déréférencement*

En C♯, on peut obtenir un pointeur sur une variable en utilisant l'*opérateur d'indirection* &. Par exemple :

```
long unEntier = 98 ;
long * pUnEntier = &unEntier;
```

On peut accéder à l'objet pointé par l'*opérateur de déréférencement* *. Par exemple :

```
long unEntier = 98 ;
long * pUnEntier = &unEntier ;
long unAutreEntier = *pUnEntier ;
// unAutreEntier vaut ici 98
```

## *L'opérateur sizeof*

L'opérateur `sizeof` permet d'obtenir la taille, en octets, d'un type valeur. Cet opérateur ne peut être utilisé qu'en mode non vérifiable. Par exemple :

```
int i = sizeof(int)  // i vaut 4
int j = sizeof(double)  // j vaut 8
```

## *Arithmétique des pointeurs*

Un pointeur sur un type `T` peut être modifié au moyen des opérateurs unaires « `++` » et « `--` ». L'opérateur binaire « `-` » peut aussi être utilisé avec les pointeurs.

- L'opérateur « `++` » incrémente l'adresse de `sizeof(T)` octets.
- L'opérateur « `--` » décrémente l'adresse de `sizeof(T)` octets.
- L'opérateur « `-` » utilisé entre deux pointeurs de même type `T`, retourne une valeur de type `long`. Cette valeur est égale à la différence d'octets entre les deux pointeurs divisée par `sizeof(T)`.

Les opérateurs de comparaison peuvent être utilisés sur deux pointeurs de même type ou de types différents. Rappelons la liste des opérateurs de comparaison :

```
==      !=      <       >       <=      >=
```

## *Casting de pointeurs*

Les pointeurs en C♯ ne dérivent pas de la classe `Object`. Les opérations de boxing et unboxing (voir page 350) n'existent donc pas avec les pointeurs. Cependant les pointeurs supportent à la fois le transtypage (casting) explicite et implicite.

Les *transtypages implicites* se font de n'importe quel type pointeur vers le type de pointeur `void*`.

Les *transtypages explicites* se font de :

- N'importe quel type de pointeur vers n'importe quel type de pointeur.
- N'importe quel type de pointeur vers un des types `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` (attention, nous ne parlons pas ici des type `sbyte*`, `byte*` ,`short*` etc).
- Un des types `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` vers n'importe quel type de pointeur.

## *Pointeurs de pointeurs*

Notons la possibilité d'utiliser un pointeur vers un pointeur (bien que celle-ci soit quasi inutile en C♯). On appelle ceci un *double pointeur*. Par exemple :

```
long unLong = 98 ;
long * pUnLong = &unLong ;
long ** ppUnLong = &pUnLong ;
```

Il est très important d'avoir une convention de nommage des pointeurs et des doubles pointeurs. En général, pour le nom d'un pointeur on a un « `p` » minuscule comme premier caractère, et « `pp` » pour un double pointeur.

## *Nécessité d'épingler les objets en mémoire*

C++ → C♯ La notion d'épinglage d'un objet est absolument étrangère au C++, puisqu'elle découle du fait que la plateforme .NET confie la gestion du tas géré à un ramasse-miettes.

C♯ En page 119 nous expliquons que le ramasse-miettes se réserve la possibilité de déplacer physiquement les objets dont il a la responsabilité. Les objets gérés par le ramasse-miettes sont de type référence alors que les objets pointés sont de type valeur. Si un pointeur pointe vers un champ de type valeur d'une instance d'un type référence, il y a potentiellement un problème, car l'instance de type référence peut être déplacée à tout moment par le ramasse-miettes. Le compilateur oblige les développeurs à utiliser le mot-clé `fixed` afin de signaler au ramasse-miettes de ne pas déplacer les instances de type référence qui ont un champ de type valeur pointé. La syntaxe du mot-clé `fixed` est la suivante :

*Exemple 14-1 :*

```
class Article { public long Prix = 0;}
unsafe class Program {
   unsafe public static void Main() {
      Article article = new Article() ;
      fixed (long* pPrix = &article.Prix){
         // Ici pPrix peut être utilisé et l'objet 'article' ne
         // peut pas être bougé par le ramasse-miettes.
```

```
      }
      //Ici pPrix n'existe plus et l'objet 'article' n'est plus épinglé.
   }
}
```

Si nous n'avions pas utilisé le mot-clé `fixed` dans cet exemple, le compilateur aurait produit une erreur car il sait détecter que l'objet qui sera référencé par la référence `article` sera susceptible d'être déplacé à l'exécution.

On peut épingler plusieurs objets de même type, dans une même clause `fixed`. Si l'on a besoin d'épingler des objets de types différents il faut utiliser des clauses `fixed` imbriquées.

Il faut épingler des objets le moins souvent possible, et pour le moins longtemps possible. En effet, lorsque des objets sont épinglés, le travail du ramasse-miettes est considérablement moins efficace.

Les variables de type valeur déclarées en tant que variables locales dans une méthode (statique ou non) n'ont pas besoin d'être épinglés puisqu'elles ne sont pas prises en compte par le ramasse-miettes. En conséquence il ne faut épingler que les objets de types « pointables », définis en tant que champs (statiques ou non) d'une classe (et non d'une structure).

En anglais le fait d'épingler un objet se dit « *to pin an object* ».

## *Les pointeurs et les tableaux*

C++ → C# Les notions de pointeurs et de tableaux sont assez proches en C++, du fait que les éléments d'un tableau sont contigus en mémoire. En C# cette similitude existe aussi.

C# En C#, les éléments d'un tableau d'éléments d'un « type pointable » peuvent être accédés au moyen de pointeurs. Nous rappelons qu'un tableau est une instance de la classe `System.Array` et en tant que tel, est stocké sur le tas géré par le ramasse-miettes. Voici un exemple qui nous montre à la fois la syntaxe mais aussi, un dépassement de limite (qui n'est pas détecté ni à la compilation ni à l'exécution !) dû à l'utilisation des pointeurs :

*Exemple 14-2 :*

```
using System ;
public class Program {
   unsafe public static void Main(){
      // Création d'un tableau à quatre éléments.
      int [] tableau = new int[4] ;
      for(int i=0 ; i < 4 ; i++)
         tableau[i] = i*i ;
      Console.WriteLine("Affichage de 6 éléments du tableau oups !!") ;
      fixed( int *ptr = tableau )
         for( int j = 0 ; j< 6 ; j++ )
            Console.WriteLine( *(ptr+j) );
      Console.WriteLine("Affichage de tous les éléments du tableau:") ;
      foreach(int k in tableau)
         Console.WriteLine(k) ;
   }
}
```

Voici l'affichage :

```
Affichage de 6 éléments du tableau oups !!
0
1
4
9
0
2042318948
Affichage de tous les éléments du tableau:
0
1
4
9
```

Notez qu'il est nécessaire d'épingler seulement le tableau et non pas chacun des éléments du tableau. Ceci confirme le fait qu'à l'exécution les éléments de type valeur d'un tableau sont stockés d'une manière contiguë en mémoire.

## *Les tableaux fixés de taille fixe*

C# 2.0 permet de déclarer un champ de type tableau de taille fixe d'éléments de types primitifs au sein d'une structure. Pour cela il suffit de déclarer le tableau avec le mot clé `fixed` et la structure avec le mot clé `unsafe`. Le type du champ n'est alors pas `System.Array` mais un pointeur vers le type primitif (i.e le champ `TableauFixe` est de type `int*` dans l'exemple suivant) :

*Exemple 14-3 :*

```
unsafe struct Foo {
   public fixed int TableauFixe[10];
   public int Overflow ;
}
unsafe class Program {
   unsafe public static void Main() {
      Foo foo = new Foo() ;
      foo.Overflow = -1 ;
      System.Console.WriteLine(foo.Overflow) ;
      foo.TableauFixe[10] = 99999 ;
      System.Console.WriteLine(foo.Overflow) ;
   }
}
```

Cet exemple affiche ceci :

```
-1
99999
```

Comprenez bien que l'accès `TableauFixe[10]` référence un onzième élément du tableau puisque les éléments sont zéro indexés. Ainsi, nous affectons en fait la valeur 99999 à l'entier `Overflow`.

## *Réservation de mémoire sur la pile avec stackalloc*

C++ → C♯ C♯ permet, avec une nouvelle syntaxe, d'allouer un tableau d'éléments de' type pointable' sur la pile. Le résultat est le même qu'une allocation statique d'un tableau en C++.

C♯ C♯ permet d'allouer un tableau d'éléments de 'type pointable' sur la pile. Le mot-clé `stackalloc` est utilisé à cette fin, avec la syntaxe suivante :

*Exemple 14-4 :*

```
public class Program {
   unsafe public static void Main(){
      int * tableau = stackalloc int[100];
      for( int i = 0 ; i< 100 ; i++ )
         tableau[i] = i*i ;
   }
}
```

Aucun des éléments du tableau n'est initialisé. La responsabilité de l'initialisation incombe au développeur. S'il n'y a pas assez de mémoire sur la pile, l'exception `System.StackOverflowException` est lancée.

La taille de la pile est relativement petite, et on ne peut y allouer des tableaux de plus de quelques milliers d'éléments. La désallocation d'un tel tableau se fait implicitement lorsque la méthode retourne.

## *Chaîne de caractères et pointeurs*

Le compilateur C♯ accepte l'obtention d'un pointeur de type `char` à partir d'une instance de la classe `System.String`. Vous pouvez exploiter cette possibilité pour vous affranchir du caractère immuable des chaînes de caractères gérées. L'immuabilité des chaînes de caractères gérées permet de faciliter grandement la manipulation de chaînes de caractères. Néanmoins, ceci peut nuire aux performances. La classe `System.StringBuiler` ne présente pas toujours une solution convenable aussi il peut être utile de pouvoir aller directement modifier les caractères d'une chaînes. L'exemple suivant montre comment exploiter cette possibilité pour écrire une méthode qui met en majuscule les caractères d'une chaîne :

*Exemple 14-5 :*

```
public class Program {
   static unsafe void ToUpper(string str) {
      fixed (char* pfixed = str)
         for (char* p = pfixed ; *p != 0 ; p++)
            *p = char.ToUpper(*p);
   }
   static void Main() {
      string str = "Bonjour" ;
      System.Console.WriteLine(str) ;
      ToUpper(str) ;
      System.Console.WriteLine(str) ;
   }
}
```

# *Les exceptions et le traitement des erreurs*

## *La problématique : Gérer toutes les erreurs dans un programme*

Les applications doivent faire face à des situations exceptionnelles indépendantes du programmeur. Par exemple :

- Accéder à un fichier qui n'existe pas ou plus.
- Faire face à une demande d'allocation de mémoire alors qu'il n'y en a plus.
- Accéder à un serveur qui n'est plus disponible.
- Accéder à une ressource sans en avoir les droits.
- Saisie d'un paramètre invalide par l'utilisateur (une date de naissance en l'an 3000 par exemple).

Ces situations, qui ne sont pas des bugs mais que l'on peut appeler erreurs, engendrent cependant un arrêt du programme, à moins qu'elles ne soient traitées. Pour les traiter on peut tester les codes d'erreur retournés par les fonctions, mais ceci présente deux inconvénients :

- Le code devient lourd, puisque chaque appel à une fonction est suivi de nombreux tests. Les tests ne sont pas centralisés. Cela viole le principe de cohérence du code, très important en architecture logicielle.
- Le programmeur doit prévoir toutes les situations possibles dès la conception du programme. Il doit aussi définir les réactions du programme et les traitements à effectuer pour chaque type d'erreur. Il ne peut pas simplement factoriser plusieurs types d'erreur en un seul traitement.

En fait ces inconvénients sont majeurs, et il a fallu trouver une solution à ce problème : c'est la gestion des *exceptions*.

## *Principe de la gestion des exceptions*

C++ → C♯ Au niveau de la gestion des exceptions, aucun principe majeur ne change par rapport à C++. En revanche, nous verrons que de nombreux détails ont évolué notamment parce que les exceptions .NET sont complètement gérées par le CLR et non par le système d'exploitation.

C♯ Voici les étapes dans la gestion d'une exception :

- Une erreur survient.
- Le CLR, le code du *framework* .NET ou notre propre code construit un objet qui contient, éventuellement, des paramètres descriptifs de l'erreur. Le détail de l'implémentation d'un tel objet sera présenté un peu plus loin.
- L'exception est lancée. Elle est paramétrée par l'objet.
- Deux possibilités peuvent alors survenir à ce moment :
  - Un gestionnaire d'exception rattrape l'exception. Il l'analyse et a la possibilité d'exécuter du code, par exemple pour sauver des données ou avertir l'utilisateur.
  - Aucun gestionnaire d'exception ne rattrape l'exception. Le programme se termine.

Voici un exemple où une exception de type `System.DivideByZeroException` est lancée : (notez que la division par zéro de nombres réels ne déclenche pas de lancement d'une exception).

*Exemple 14-6 :*

```
public class Program {
   public static void Main() {
      int i = 1 ;
      int j = 0 ;
      int k = i/j ;
   }
}
```

Le programme s'arrête puisque l'exception n'est pas rattrapée et la fenêtre de la Figure 14-1 s'affiche (que l'assemblage ait été compilé en mode Debug ou Release), vous proposant de déboguer votre programme :

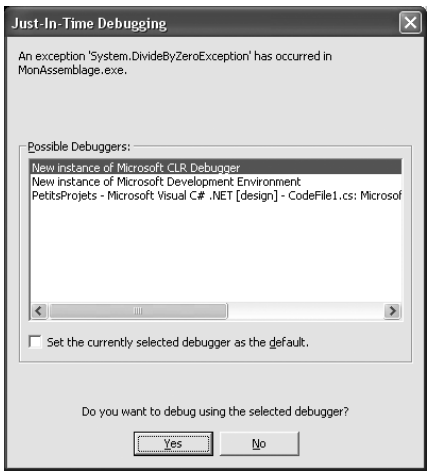


*Figure 14-1 : Conséquence d'une exception non rattrapée*

Une fois que vous êtes en mode Debug *Visual Studio 2005* présente un assistant très pratique pour visualiser les données relatives à l'exception :

Voici le même exemple où l'exception est rattrapée par un gestionnaire d'exception :

*Exemple 14-7 :*

```
using System ;
public class Program {
   public static void Main(){
      try {
        int i = 1 ;
        int j = 0 ;
```

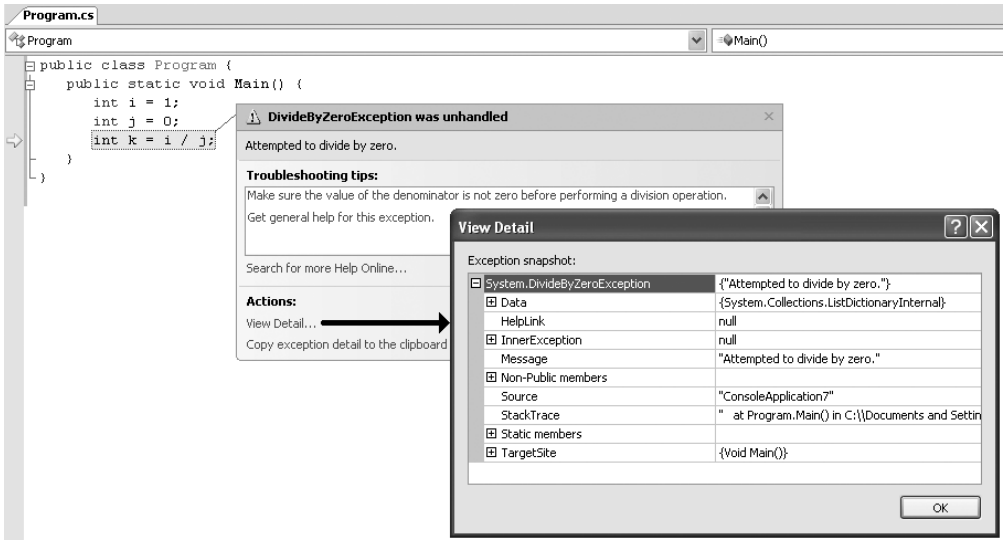


*Figure 14-2 : Assistant de la gestion des exceptions*

```
int k = i/j ;
      } catch( System.DivideByZeroException ) {
        Console.WriteLine("Une division entière par zéro a eu lieu !") ;
      }
   }
}
```

L'exception étant rattrapée, le programme ne s'arrête pas et affiche sur la console :

```
Une division entière par zéro a eu lieu !
```

La syntaxe pour définir un gestionnaire d'exceptions ne fait donc appel qu'à deux mots-clés : `try`, `catch`. Cette syntaxe est la même que celle d'autres langages comme C++, ADA ou Java. Nous précisons que l'on peut imbriquer les blocs `try/catch`.

# *Objet associé à une exception et lancement de vos propres exceptions*

C++ → C# À l'instar du C++, en C#, une exception est représentée par un objet. L'exception peut être lancée avec la syntaxe `throw new objet`, qui existe aussi en C++. En C++, on avait la possibilité de lancer une exception sans la matérialiser par un objet. En C# il est obligatoire d'allouer un nouvel objet pour chaque exception lancée explicitement avec `throw` (sauf si le lancement s'effectue dans une clause `catch`, auquel cas l'exception couramment traitée est renvoyée).

En C#, cet objet est forcément instance d'une classe dérivée de la classe `System.Exception`. Rappelons que C++ accepte n'importe quel type d'objet.

De plus, en tant qu'objet C#, ce dernier est obligatoirement alloué dynamiquement (ce qui n'est pas le cas en C++).

C# Une exception est toujours représentée par un objet. Cet objet contient en général des informations de description relatives au problème qui a provoqué l'exception. Cet objet est obligatoirement une instance d'une classe qui dérive de la classe `System.Exception`. Il existe deux types de classes dérivant de `System.Exception` :

- Celles fournies par C#/.NET qui sont lancée par le système, mais peuvent aussi être lancées dans vos propres méthodes (par exemple `System.DivideByZeroException` vue précédemment).
- Celles que vous définissez vous-même et qui ne peuvent être lancées que dans vos propres méthodes.

## *La classe System.Exception*

La classe `System.Exception` contient des propriétés que vous pouvez utiliser pour vos propres classes :

- `public string Message{get;}`
  Cette chaîne de caractères doit contenir un message descriptif de l'exception. Cette propriété est accessible en lecture seule, mais vous pouvez l'initialiser en appelant le constructeur de `System.Exception` acceptant une chaîne de caractères en paramètre (la propriété prend alors la valeur de l'argument de ce constructeur).
- `public string Source{get;set;}`
  Cette chaîne de caractères contient le nom de l'objet ou de l'application qui a généré l'erreur.
- `public string HelpLink{get;set;}`
  Cette chaîne de caractères contient une référence vers une page d'explication de l'exception. Vous pouvez vous en servir si vous mettez en ligne des informations sur vos propres exceptions.
- `public Exception InnerException{get;}`
  Cette propriété référence une exception. Elle est utilisée lorsqu'une exception rattrapée provoque le lancement d'une nouvelle exception. La nouvelle exception référence l'exception rattrapée au moyen de cette propriété.

Il est conseillé d'utiliser ces propriétés mais ce n'est pas une obligation. Par exemple, dans le cas des exceptions qui instancient la classe `System.DivideByZeroException` :

- La propriété `Message` contient la chaîne de caractères `"Attempted to divide by zero."`.
- La propriété `Source` est une chaîne de caractères égale au nom de l'assemblage d'où l'exception est lancée.
- La propriété `HelpLink` est une chaîne de caractères vide.

D'autres propriétés intéressantes et initialisées automatiquement par le CLR, sont disponibles dans cette classe :

- `public string StackTrace{get;}`
  Contient la représentation des appels de méthodes sur la pile au moment où l'exception a été lancée (la méthode la plus récente apparaît en premier). Il se peut que cette chaîne ne contienne pas ce qui logiquement devrait y être, à cause des optimisations du compilateur qui parfois modifie la structure du code. Si le programme a été compilé en mode Debug, cette chaîne contient aussi le numéro de la ligne et le nom du fichier source de l'instruction qui a lancé l'exception.
- `public MethodBase TargetSite{get;}`
  Retourne un objet de type `MethodBase` qui référence la méthode qui a lancée l'exception.

## *Définition de vos propres classes d'exceptions*

Comme nous l'avons mentionné, nous pouvons fabriquer nos propres exceptions à l'aide de classes qui dérivent de la classe `System.Exception`. En fait, il est plutôt conseillé de faire dériver vos propres classes d'exceptions de la classe `System.ApplicationException` (qui elle-même dérive directement de la classe `Exception`). Voici par exemple la définition d'une exception qui pourrait être utilisée lorsqu'un argument entier d'une fonction sort d'un intervalle.

*Exemple 14-8 :*

```
using System ;
public class ExceptionArgEntierHorsLimite : ApplicationException {
   public ExceptionArgEntierHorsLimite (int argVal,int inf,int sup):
      base(string.Format(
         "L'argument {0} est hors de l'intervalle [{1},{2}]",
         argVal, inf, sup)){}
}
```

Nous aurions éventuellement pu sauver les trois valeurs entières dans trois champs.

## *Lancement d'exceptions dans vos propres méthodes*

Vous avez la possibilité de lancer une exception (propriétaire ou non) avec le mot-clé C# `throw`. Il est obligatoire de représenter une exception par un objet, sauf si l'exception est lancée dans une clause `catch`, auquel cas l'exception couramment traitée est renvoyée. Voici un exemple d'utilisation de `throw` avec notre propre classe d'exception :

*Exemple 14-9 :*

```
using System ;
public class ExceptionArgEntierHorsLimite : ApplicationException{
   public ExceptionArgEntierHorsLimite (int argVal,int inf,int sup):
      base(string.Format(
         "L'argument {0} est hors de l'intervalle [{1},{2}]",
         argVal, inf, sup)){}
}
class Program {
   static void f( int i ){
      // Supposons que i doive être entre 10 et 50 (inclus).
      if( i<10 || i>50 )
```

```
            throw new ExceptionArgEntierHorsLimite (i,10,50) ;
         // Ici, nous sommes certains que i est dans le bon intervalle
      }
      public static void Main() {
         try{
            f(60) ;
         }
         catch( ExceptionArgEntierHorsLimite e ){
            Console.WriteLine( "Exception : " + e.Message ) ;
            Console.WriteLine( "Etat de la pile lors du lancement:"
                               + e.StackTrace) ;
         }
      }
}
```

L'exécution de ce programme, lorsque ce dernier est compilé en mode Debug, affiche ceci :

```
Exception : L'argument 60 est hors de l'intervalle [10,50]
Etat de la pile lors du lancement :
  at Program.f(Int32 i) in d:\mes documents\visual studio projects
   \test_exception\program.cs:line 11
  at Program.Main() in d:\mes documents\visual studio projects
   \test_exception\program.cs:line 17
```

L'exécution de ce programme, lorsque ce dernier est compilé en mode Release, affiche ceci :

```
Exception : L'argument 60 est hors de l'intervalle [10,50]
Etat de la pile lors du lancement :
  at Program.f(Int32 i)
  at Program.Main()
```

Vous pouvez aussi lancer des exceptions définies par le *framework* .NET. Par exemple, on aurait pu utiliser la classe `System.ArgumentOutOfRangeException` comme ceci :

*Exemple 14-10 :*

```
public class Program {
   static void f(int i) {
      if( i<10 || i>50 )
         throw new System.ArgumentOutOfRangeException("i") ;
   }
   public static void Main() {
      try{
         f(60) ;
      }
      catch(System.ArgumentOutOfRangeException e){
         System.Console.WriteLine("Exception : " + e.Message) ;
      }
   }
}
```

Ce programme affiche ceci :

```
Exception: Specified argument was out of the range of valid values.
Parameter name: i
```

## *Pas d'exceptions contrôlées en C#*

Les développeurs connaissant le langage Java peuvent s'étonner de l'absence d'*exceptions contrôlées* (aussi appelée *exception vérifiée* ou *checked exception* en anglais) en C#. Dans l'article à l'URL `http://www.artima.com/intv/handcuffsP.html` *Anders Hejlsberg* un des concepteurs principaux de C#, expose deux problèmes potentiels engendrés par les exceptions contrôlées. Ces problèmes surviennent lorsqu'un grand nombre d'API est appelé et lorsque l'on doit maintenir une nouvelle version d'une application. En l'absence de solutions ne présentant pas ces problèmes, les concepteurs du langage C# ont préféré ne pas fournir de mécanisme pour contrôler les exceptions.

# *Le gestionnaire d'exceptions et la clause finally*

C++ → C# Les gestionnaires d'exception du langage C# sont assez similaires à ceux du langage C++ mis à part les points suivants.

En C#, on n'utilise pas la syntaxe `catch(...)` pour rattraper toutes les exceptions mais la syntaxe `catch(Exception e)`. L'avantage est que l'on a ainsi un accès à l'exception.

La clause `finally` de C# n'existe pas en C++. Elle remplace le fait qu'en C++ on libère souvent les ressources critiques dans des destructeurs d'objets statiquement alloués. Ceci n'est pas possible en C# puisqu'on ne peut coder les destructeurs des types valeur. Un autre problème est qu'en C# le moment de l'appel d'un destructeur est indéterminé.

## *Remarques sur la clause catch*

C# Un gestionnaire d'exception peut contenir une ou plusieurs clauses `catch`, et/ou une clause `finally`.

Dans le cas où vous avez plusieurs clauses `catch`, les types des exceptions des clauses doivent être différents deux à deux. De plus C# testera si le type de l'exception à rattraper correspond au type d'exception précisé dans la clause `catch`, de la première clause `catch` à la dernière. Au plus une clause `catch` sera exécutée.

Notez qu'une exception lancée, instance d'une classe dérivée `D` de la classe de base `B`, matche à la fois la clause `catch(D d)` et la clause `catch(B b)`. Les conséquences de cette remarque sont les suivantes :

- Dans le même gestionnaire d'exception, le compilateur interdit les clauses `catch` de classes dérivées de `B`, après la définition d'une clause `catch(B b)`. En effet les clauses `catch` des classes dérivées de `B` n'auraient aucune chance d'être exécutées.
- La clause `catch(System.Exception)` rattrape toutes les exceptions puisque toutes les classes d'exception dérivent de la classe `System.Exception`.
- Vous avez la possibilité d'avoir une clause `catch` vide. Vous n'avez qu'à écrire le mot-clé `catch` suivit directement de l'accolade ouvrante du bloc `catch`. Ce type de clause est équivalent à une clause `catch(System.Exception)`.

Si un gestionnaire d'exception a une clause `catch` vide celle-ci est forcément en dernière position. Même remarque pour une clause `catch(System.Exception)`. Si ces deux clauses sont présentes, la clause `catch` vide doit être en dernière position.

## *La clause finally*

Si la clause `finally` est présente, elle se trouve obligatoirement après toutes les clauses `catch`. La clause `finally` est un bloc d'instructions exécuté dans tous les cas possibles qui sont :

- Aucune exception n'a été lancée dans le bloc `try`.
- Une exception a été lancée dans le bloc `try` et a été rattrapée par un gestionnaire d'exception `catch`.
- Une exception a été lancée dans le bloc `try` et a été rattrapée par un gestionnaire d'exception `catch`. Une exception est alors levée lors de l'exécution de ce gestionnaire d'exception.
- Une exception a été lancée dans le bloc `try` et n'a pas été rattrapée par le gestionnaire d'exception.

La clause `finally` est en général utilisée pour libérer des ressources critiques (une connexion avec une base de données, un fichier ouvert, etc) indépendamment du fait qu'une exception ait été lancée ou pas. Notez que si une exception est lancée dans un bloc `finally` (ce qui est déconseillé) alors qu'une exception lancée n'a pas pu être rattrapée par le gestionnaire d'exception courant, elle remplace cette dernière. Par exemple :

*Exemple 14-11 :*

```
using System ;
public class Program {
   public static void Main(){
      try {
         try {
            throw new ArgumentOutOfRangeException();
         }
         catch(DivideByZeroException e){
            Console.WriteLine("Gestionnaire 1:") ;
            Console.WriteLine("Exception: "+e.Message) ;
         }
         finally {
            Console.WriteLine("finally 1") ;
            throw new DivideByZeroException();
         }
      }
      catch(Exception e) {
         Console.WriteLine("Gestionnaire 2:") ;
         Console.WriteLine("Exception: "+e.Message) ;
      }
      finally {
         Console.WriteLine("finally 2") ;
      }
   }
}
```

Ce programme affiche :

```
finally 1
Gestionnaire 2:
Exception: Attempted to divide by zero.
finally 2
```

## *Améliorer la sémantique d'une exception*

Lorsqu'une exception remonte les différents appels de méthodes imbriquées, elle a tendance à perdre de sa signification car elle traverse plusieurs couches de code. Il est donc souvent nécessaire de rattraper l'exception pour en relancer une nouvelle, dont le contenu dépendra de l'exception rattrapée. Pour ne rien perdre de l'exception initiale, vous avez la possibilité de la référencer directement dans la nouvelle exception au moyen de la propriété `InnerException` de la classe `Sytem.Exception`.

# *Exceptions lancées dans un constructeur ou dans la méthode Finalize()*

C++ → C# En C# une exception lancée dans un constructeur non statique est traitée exactement comme toutes les autres exceptions, et l'objet n'est pas créé. On va s'intéresser au comportement du CLR lorsqu'une exception est lancée à partir des constructeurs statiques.

En C#, une exception lancée dans la méthode `Finalize()` et qui est non rattrapée dans le constructeur, ne provoque pas de problèmes majeurs, comme c'est le cas en C++.

## *Exception lancée dans un constructeur d'instance*

C# Une exception lancée dans un constructeur d'instance, est traitée exactement comme toutes les exceptions et l'objet n'est pas créé. Cependant il reste déconseillé de lancer une exception dans un constructeur. Voici un petit programme pour illustrer ceci :

*Exemple 14-12 :*

```
using System ;
public class Article {
   public Article(){ i=3 ; throw new ArgumentOutOfRangeException(); }
   public Article(int j){ i=j ; }
   public int i=0 ;
}
public class Program {
   public static void Main() {
      Article article = new Article( 2 ) ;
      try{
         article = new Article() ;
      }
      catch(Exception e) {
         Console.WriteLine("Exception: "+e.Message) ;
```

```
      }
      Console.WriteLine(article.i) ;
   }
}
```

Ce programme affiche :

```
Exception: Specified argument was out of the range of valid values.
2
```

Comprenez bien que l'allocation d'une seconde instance de `Article`, dans le bloc `try`, a échoué du fait qu'une exception a été lancée sans être rattrapée, dans le constructeur.

## *Exception lancée dans un constructeur statique ou lors de l'initialisation d'un champ statique*

L'appel des constructeurs statiques et l'initialisation des champs statiques est non déterministe, c'est-à-dire que les règles qui régissent le moment de leurs appels dépendent de l'implémentation du langage et ne sont en général pas documentées. Plus de détails à ce sujet sont disponibles page 431. Cependant, une exception peut être lancée dans ces méthodes et non rattrapée. Dans ce cas, tout se passe comme si une exception de type `TypeInitializationException` est lancée à l'endroit (non déterminé *a priori*) du programme qui a déclenché l'appel au constructeur statique. En général cet endroit est la première instanciation de cette classe ou le premier accès à un de ses membres statiques. L'exception qui a été lancée dans le constructeur statique est maintenant référencée par le champ `Exception.InnerException` de l'exception de type `TypeInitializationException`. Ce comportement est automatiquement pris en charge par le CLR. Voici un exemple pour illustrer ceci :

*Exemple 14-13 :*

```
using System ;
public class Article {
   static  Article() {
     throw new ArgumentOutOfRangeException() ;
   }
}
public class Program {
   public static void Main() {
     try{
       Article article = new Article() ;
     }
     catch(Exception e) {
       Console.WriteLine("Exception: " + e.Message) ;
       Console.WriteLine("Inner Exception: "+e.InnerException.Message) ;
     }
   }
}
```

Ce programme affiche :

```
Exception: The type initializer for "Article" threw an exception.
Inner Exception: Specified argument was out of the range of valid values.
```

### *Exception lancée dans la méthode Finalize()*

Une exception lancée dans la méthode `Finalize()` et non rattrapée dans cette méthode a pour effet de faire sortir immédiatement le thread dédié à l'exécution des finaliseurs de cette méthode. Cependant la mémoire allouée à l'objet est quand même rendue au système par le ramasse-miettes.

Si l'exception a été lancée dans la méthode `Finalize()` d'une classe dérivée et non rattrapée, le finaliseur de la classe de base est quand même automatiquement exécuté.

Il est très fortement déconseillé de lancer une exception dans la méthode `Finalize()`.

## *Le CLR et la gestion des exceptions*

Il faut bien comprendre que le CLR s'occupe complètement de la gestion des exceptions .NET. Il a notamment les responsabilités suivantes :

- Le CLR a la responsabilité de trouver le *gestionnaire d'exception* adéquat (i.e le bloc `catch()` adéquat) en remontant la liste d'appels imbriqués de méthodes qui se trouve dans la pile. Pendant cette recherche, le CLR sauve plusieurs informations dans l'objet représentant l'exception. Parmi ces informations, on trouve la pile d'appels, qui représente l'enchaînement des appels de méthodes. Ainsi le développeur a à sa disposition ces informations pour l'aider à comprendre les causes de l'exception.
- Lorsque l'exception est lancée dans un environnement distribué, le CLR se charge de la « sérialiser » et de la propager au domaine d'application contenant le client. Soyez conscient qu'il y a cependant quelques précautions à prendre si vous souhaitez faire transiter par .NET Remoting vos propres types exceptions (plus d'informations à ce sujet sont disponibles sur cette page `http://www.thinktecture.com/Resources/RemotingFAQ/CustomExceptions.html`).
- Le CLR a parfois la responsabilité de rattraper une exception pour en relancer une autre, considérée comme plus significative. Par exemple nous avons vu dans la section précédente que le CLR propage une exception de type `TypeInitializationException` quel que soit le type de l'exception qui a été lancée dans le constructeur statique. Notez que dans ce genre de cas, l'exception initialement lancée est stockée dans la propriété `InnerException` de l'exception. C'est aussi le cas des exceptions lancées à partir d'une méthode appelée avec un lien tardif explicite.
- En page 285 nous expliquons que le CLR transforme les codes d'erreur `HRESULT` retournés par les méthodes des objets COM en exception gérée. De même, lorsqu'un objet géré est considéré comme un objet COM, le CLR produit un code d'erreur `HRESULT` à partir d'une exception gérée qui remonte dans le code natif. Nous verrons dans la prochaine section comment le CLR se comporte avec le système d'exception natif de *Windows*.
- En page 92nous expliquons que la classe `AppDomain` présente l'événement d'instance `UnhandledException`. L'exemple suivant montre que ces événements peuvent être exploités pour exécuter du code au moment où le CLR réalise que le processus courant va crasher parce qu'une exception n'a pas été rattrapée. Nous précisons en commentaire le genre d'action à faire avant de terminer le processus :

*Exemple 14-14 :*

```
using System ;
using System.Threading ;
public class Program {
   public static void Fct() {
      try {
         Console.WriteLine("Fct() Bloc try.") ;
         throw new Exception("Hello from Fct()...") ;
      } finally {
         Console.WriteLine("Fct() Bloc finally.") ;
      }
   }
   public static void Main() {
      Console.WriteLine("Thread{0}: Hello world...",
                        Thread.CurrentThread.ManagedThreadId ) ;
      AppDomain currentDomain = AppDomain.CurrentDomain;
      currentDomain.UnhandledException += UnhandledExceptionHandler;
      try{
         Console.WriteLine("Main() Bloc try.") ;
         Fct() ;
      }
      catch{
         Console.WriteLine("Main() Bloc catch.") ;
      }
      throw new Exception ("Bonjour...") ;
   }
   static void UnhandledExceptionHandler(
                object s, UnhandledExceptionEventArgs e) {
      Console.WriteLine("Thread{0}: UnhandledExceptionHandler: {1}",
                        Thread.CurrentThread.ManagedThreadId ,
                       (e.ExceptionObject as Exception).Message) ;
      //  a) Sauvegarder un rapport d'erreur.
      //  b) Proposer à l'utilisateur de sauvegarder l'état courant.
      //  c) Proposer à l'utilisateur d'envoyer automatiquement le
      //     rapport d'erreur à l'équipe de développement.
   }
}
```

- Ce programme affiche ceci avant de crasher :

```
Thread1: Hello world...
Main() Bloc try.
Fct() Bloc try.
Fct() Bloc finally.
Main() Bloc catch.
Thread1: UnhandledExceptionHandler: Bonjour...
```

- En page 125 nous exposons des techniques propres au CLR permettant d'améliorer la fiabilité d'une application susceptible de faire face à des exceptions asynchrones.

## *Exceptions non gérées*

Typiquement, le thread *Windows* sous jacent à un thread géré traverse du code géré (compilé en code natif) et du code natif. Lorsqu'une exception gérée survient dans du code géré le CLR fait en sorte de trouver le bloc `catch` adéquat dans le code géré. Les zones de code natif éventuelles qui se situent entre le déclenchement de l'exception et la clause `catch` gérée trouvée sont alors dépilées normalement.

*Windows* présente un mécanisme d'exception nommé *Structured Exception Handling* (*SEH*) pour gérer les exception natives qui surviennent. Une description détaillée de ce mécanisme est disponible dans cet article de *Matt Pietrek* `http://www.microsoft.com/msj/0197/exception/exception.aspx`.

Le CLR est capable de détecter lorsqu'une exception native remonte dans du code géré. Dans cette situation il produit alors une exception gérée dont le type dépend du code de l'exception native :

| Code de l'exception native | Type de l'exception géré |
|---|---|
| `STATUS_FLOAT_INEXACT_RESULT` | `System.ArithmeticException` |
| `STATUS_FLOAT_INVALID_OPERATION` | |
| `STATUS_FLOAT_STACK_CHECK` | |
| `STATUS_FLOAT_UNDERFLOW` | |
| `STATUS_FLOAT_OVERFLOW` | `System.OverflowException` |
| `STATUS_INTEGER_OVERFLOW` | |
| `STATUS_FLOAT_DIVIDE_BY_ZERO` | `System.DivideByZeroException` |
| `STATUS_INTEGER_DIVIDE_BY_ZERO` | |
| `STATUS_FLOAT_DENORMAL_OPERAND` | `System.FormatException` |
| `STATUS_ACCESS_VIOLATION` | `System.NullReferenceException` |
| `STATUS_ARRAY_BOUNDS_EXCEEDED` | `System.IndexOutOfRangeException` |
| `STATUS_NO_MEMORY` | `System.OutOfMemoryException` |
| `STATUS_STACK_OVERFLOW` | `System.StackOverflowException` |
| Tous les autres code | `System.Runtime.InteropServices.SEHException` |

Le mécanisme SEH fonctionne avec un modèle de filtres d'exceptions enregistrés auprès des threads. Ces filtres permettent de préciser à *Windows* des fonctions natives à appeler lorsqu'une exception native est détectée sur un thread. Le CLR enregistre son propre filtre sur chaque thread exécutant du code géré. La fonction native enregistrée a la responsabilité de déclencher l'événement `AppDomain.UnhandledException`. Si un thread n'a pas été créé par le CLR (i.e s'il

n'a pas été créé à partir d'un appel à la méthode `Thread.Start()`) il y a toujours un risque que les filtres d'exceptions enregistrés sur ce thread par d'autres composants viennent perturber le comportement du filtre du CLR. Aussi, soyez conscient que dans cette situation l'événement `UnhandledException` ne sera pas nécessairement déclenché.

# *Les exceptions et l'environnement Visual Studio*

Lorsque l'environnement *Visual Studio* débogue une application, il offre la possibilité de suspendre l'exécution en donnant la main au débogueur lorsqu'une exception (gérée ou non) est lancée ou lorsqu'elle n'est pas rattrapée. Ce paramétrage se fait en fonction du type de l'exception. Pour accéder à cette fonctionnalité, il faut aller dans le menu *Debug Exceptions...*. La liste des exceptions est divisée en cinq catégories :

- **C++ Exceptions** : liste les exceptions C++ non gérées.
- **Common Language Runtime Exceptions** : liste les exceptions gérées (trier par espaces de noms).
- **Managed Debugging Assistants** : liste des événements problématiques connus du CLR qui se traduisent parfois par une exception gérée. Ainsi, si vous souhaitez confirmer qu'une telle exception est la conséquence d'un tel événement ou si vous souhaitez être informé d'un tel événement lorsqu'il ne se traduit pas par une exception vous devez vous servir de cette liste. Chacun de ces événements est expliqué dans les **MSDN** dans l'article **Diagnosing Errors with Managed Debugging Assistants**.
- **Native Run-Time Checks** : liste des exceptions critiques qui peuvent survenir dans un programme C/C++.
- **Win32 Exceptions** : liste les code d'exceptions SEH.

*Visual Studio* vous permet notamment d'ajouter vos propres types d'exceptions dans une de ces listes.

# *Conseils d'utilisation des exceptions*

## *Quand lancer une exception ?*

Le mécanisme d'exception est en général bien compris mais mal utilisé. **Le principe de base est qu'une application qui fonctionne en mode normal ne devrait pas lancer d'exception.** Cela repousse le problème sur la définition des situations anormales. Il y en a de trois types :

- Celles qui surviennent à cause d'un problème d'environnement d'exécution et qui peuvent être réglées par une modification de cet environnement (absence d'un fichier de configuration, mauvais mot de passe, réseau indisponible, permissions restreintes etc). On parle d'*exception applicative*.
- Celles qui surviennent à cause d'un problème d'environnement d'exécution qui ne peut être résolu. Par exemple, les applications gourmandes en mémoire telles que *SQL Server 2005* peuvent être limitées par les 2 ou 3Go d'espace utilisateur d'un processus *Windows* 32 bits. On parle d'*exception asynchrone* du fait qu'une telle exception ne dépend pas de la sémantique de la portion de code qui l'a levé. Pour gérer ce genre de problème vous devez avoir recours à des mécanismes complexes présentés en page 125. Cela revient à considérer

ces situations anormales comme normales ! **Soyez conscient que seuls les gros serveurs qui poussent le système dans ses retranchement en utilisant au mieux toute la mémoire disponible devraient rencontrer des exceptions asynchrones et donc avoir recours à ce type de mécanisme.**

- Celles qui surviennent à cause d'un bug et qui ne peuvent être réglées que par une nouvelle version corrigeant ce bug.

## *Que faire lorsque l'on rattrape une exception ?*

Lorsque vous rattrapez une exception vous pouvez envisager 3 scénarios :

- Soit vous avez à faire à un réel problème mais vous pouvez y remédier en corrigeant les conditions qui l'ont engendré. À cette fin, vous pouvez avoir besoin de nouvelles informations (mot de passe invalide ▶ redemander à l'utilisateur...).
- Soit vous avez à faire à un problème que vous ne pouvez résoudre à ce niveau. Dans ce cas la seule bonne attitude est de relancer l'exception. Il se peut qu'il n'y ait plus de gestionnaire d'exception approprié et dans ce cas vous déléguez la décision à prendre à l'hôte du moteur d'exécution. Dans les applications consoles ou fenêtrées ce dernier décide de faire tomber le processus tout entier. Notez que vous pouvez avoir recours à l'événement `AppDomain.UnhandledException` déclenché dans cette situation de façon à maîtriser la terminaison du processus. Vous pouvez alors profiter de cette « dernière chance » pour sauver des données (à la *Word*) qui sans cela seraient définitivement perdues. Dans une application ASP.NET un mécanisme de traitement des erreurs décrit en page 904 est mis en place.
- En théorie un troisième scénario est envisageable. Il se peut que l'exception reçue représente une fausse alerte. En pratique, ce cas n'arrive jamais.

Il ne faut pas rattraper une exception pour la loguer puis la relancer. Pour loguer les exceptions et le code qu'elles ont traversées, nous vous conseillons d'avoir recours à des techniques non intrusives telles que l'utilisation des événements spécialisés de la classe `AppDomain` ou l'analyse des appels de méthodes sur la pile au moment où l'exception a été lancée.

Il ne faut pas nettoyer les ressources que l'on a alloué lorsque l'on rattrape une exception. D'ailleurs, soyez conscient qu'en général, seules les ressources non gérées sont susceptibles de vous poser des problèmes (fuite de mémoire etc). Ce genre de code de libération de ressources doit être placé dans des clauses `finally` ou dans des méthode `Dispose()`. En C#, les clauses `finally` sont souvent implicitement encapsulées dans une clause `using` qui agit sur des objets qui implémentent l'interface `IDisposable`.

## *Quand rattraper une exception ?*

Pour un type d'exception donné, se poser cette question revient à se demander à quelle profondeur de méthode ce type d'exception doit être rattrapé et qu'est ce qui doit être fait. Nous entendons par profondeur d'une méthode, le nombre d'appels imbriqués à partir du point d'entré (en général la méthode `Main()`). Ainsi la méthode représentant le point d'entrée est la moins profonde. Les réponses à ces deux questions dépendent de la sémantique d'une exception. Demandez-vous pour chaque type d'exception quel est le niveau de votre code le plus adapté pour pouvoir corriger les conditions qui l'ont déclenchées et reprendre l'exécution ou pour terminer « proprement » l'application.

En général, plus la méthode est profonde, moins elle doit rattraper d'exceptions propriétaires. La raison est que les exceptions propriétaires ont souvent une signification dans le métier de votre application. Ainsi si vous développez une bibliothèque de classes, il faut laisser remonter les exceptions significatives pour les applications clientes de la bibliothèque.

### *Exceptions vs. code retour*

Vous pourriez être tenté d'utiliser les exceptions à la place des codes de retour de vos méthodes, pour signaler un éventuel problème. Il faut être vigilant car l'utilisation des exceptions souffre des deux inconvénients majeurs suivants :

- Le code est très peu lisible. En effet, pour comprendre le code il faut faire à la main le travail du CLR qui consiste à remonter les appels jusqu'à trouver un gestionnaire d'exception. Même si vous séparez proprement votre code en couches d'appel, le code reste peu lisible.
- La gestion d'une exception par le CLR est beaucoup plus coûteuse que l'analyse d'un code de retour.

La règle fondamentale énoncée en début de section peut aussi vous aider à décider : une application qui fonctionne dans des conditions normales d'exécution ne lance pas d'exception.

### *Ne pas sous estimer les bugs dont les conséquences sont rattrapées*

Une utilisation abusive des exceptions arrive lorsque l'on se dit que, puisqu'on rattrape toutes les exceptions, celles provoquées par d'éventuels bugs seront rattrapées aussi. On se dit qu'elles ne provoqueront pas de plantage du programme. Ce raisonnement ne tient pas compte du fait que les principales nuisances des bugs sont celles qui passent inaperçues, comme des résultats indéterminés, inattendus et faux.

## *Les méthodes anonymes*

La section courante ainsi que la section suivante présentent deux possibilités ajoutées à la version 2 du langage C#. Contrairement aux génériques, ces deux possibilités n'impliquent aucune nouvelles instructions IL. Toute la magie se situe au niveau du compilateur.

Chacune de ces deux sections aura la même structure : une présentation « classique » de la fonctionnalité suivie de l'analyse du travail du compilateur pour enfin commenter des utilisations avancées.

### *Introduction aux méthodes anonymes de C#2*

Commençons par un premier exemple de réécriture de code C#1 au moyen des *méthodes anonymes* de C#2. Voici du code compilable en C#1 qui montre comment référencer une méthode à l'aide d'un délégué :

*Exemple 14-15 :*

```
class Program {
   delegate void DelegateType() ;
   static DelegateType GetMethod(){
      return new DelegateType(MethodBody) ;
```

```
   }
   static void MethodBody() {
      System.Console.WriteLine("Hello") ;
   }
   static void Main() {
      DelegateType delegateInstance = GetMethod() ;
      delegateInstance() ;
      delegateInstance() ;
   }
}
```

Voici le même code réécrit en C#2 à l'aide d'une méthode anonyme :

*Exemple 14-16 :*

```
class Program {
   delegate void DelegateType() ;
   static DelegateType GetMethod() {
      return delegate() { System.Console.WriteLine("Hello") ; };
   }
   static void Main() {
      DelegateType delegateInstance = GetMethod() ;
      delegateInstance() ;
      delegateInstance() ;
   }
}
```

Plusieurs remarques s'imposent :

- Le mot-clé `delegate` a une nouvelle utilisation en C#2. Il est utilisé au sein du corps d'une méthode pour indiquer au compilateur qu'il doit s'attendre à trouver le corps d'une méthode anonyme.
- On constate que l'on peut assigner une méthode anonyme à un délégué.
- On comprend le nom de méthode anonyme pour cette nouvelle fonctionnalité : La méthode définie au sein de `GetMethod()` n'a effectivement pas de nom. On peut cependant l'invoquer car elle est référencée par un délégué.

Notez aussi que la syntaxe d'assignement de plusieurs méthodes à un délégué avec l'opérateur += est utilisable avec les méthodes anonymes :

*Exemple 14-17 :*

```
using System ;
class Program{
   delegate void DelegateType() ;
   static void Main(){
      DelegateType delegateInstance = delegate() {
         Console.WriteLine("Hello") ; };
      delegateInstance += delegate() { Console.WriteLine("Bonjour") ; };
      delegateInstance() ;
   }
}
```

Comme l'on peut s'en douter ce programme affiche :

```
Hello
Bonjour
```

## *Méthodes anonymes et arguments*

Comme le montre l'exemple suivant, une méthode anonyme peut avoir des arguments. Tous les types sont applicables, y compris les possibilités de passer des arguments par référence avec le mot-clé `ref` et des arguments de sortie avec le mot-clé `out` :

*Exemple 14-18 :*

```
class Program {
   delegate int DelegateType(int valTypeParam, string refTypeParam,
                             ref int refParam, out int outParam) ;
   static DelegateType GetMethod() {
      return delegate( int valTypeParam , string refTypeParam,
                       ref int refParam , out int outParam     ) {
         System.Console.WriteLine(
            "Hello valParam:{0} refTypeParam:{1}",
            valTypeParam, refTypeParam) ;
         refParam++ ;
         outParam = 9 ;
         return valTypeParam ;
      } ; // Fin du corps de la méthode anonyme.
   }
   static void Main() {
      DelegateType delegateInstance = GetMethod() ;
      int refVar = 5 ;
      int outVar ;
      int i = delegateInstance(1, "un", ref refVar, out outVar);
      int j = delegateInstance(2, "deux", ref refVar, out outVar);
      System.Console.WriteLine("i:{0} j:{1} refVar:{2} outVar:{3}",
                        i, j, refVar, outVar) ;
   }
}
```

Ce programme affiche :

```
Hello valParam:1 refTypeParam:un
Hello valParam:2 refTypeParam:deux
i:1 j:2 refVar:7 outVar:9
```

Soyez conscient que le type de la valeur de retour n'est pas défini par la méthode anonyme, mais par le type du délégué auquel elle est assignée. Celui-ci est toujours défini car le compilateur oblige à assigner toutes méthodes anonymes à un délégué dont le type est connu à la compilation. On en conclut qu'on ne peut assigner une méthode anonyme à un délégué référencé par une référence de type `System.Delegate`.

La syntaxe C♯ avec le mot clé `param` n'est pas utilisable dans la liste d'arguments d'une méthode anonyme. En effet, en interne le mot clé `param` oblige le compilateur à marquer la méthode

concernée avec l'attribut `ParamArrayAttribute`. Or les méthodes anonymes ne peuvent supporter d'attribut.

*Exemple 14-19 :*

```
using System ;
class Program {
   delegate void DelegateType( params int[] arr ) ;
   static DelegateType GetMethod() {
      // erreur de compilation : param is not valid in this context.
      return delegate(params int[] arr){ Console.WriteLine("Hello");} ;
   }
}
```

## *Une subtilité syntaxique*

Lorsque le mot-clé `delegate` est utilisé sans parenthèse lors de la définition d'une méthode anonyme sans arguments d'entrée ni de retour, celle-ci peut être assignée à n'importe quel type de délégué. Bien entendu, le corps de la méthode anonyme ne peut alors pas utiliser les paramètres d'entrée. En outre, une telle méthode anonyme ne peut rien retourner. En conséquence, cette syntaxe n'est pas utilisable si la signature du délégué cible retourne une valeur ou prend un paramètre `out` :

*Exemple 14-20 :*

```
using System ;
class Program{
   delegate void DelegateType(int valTypeParam, string refTypeParam,
                              ref int refParam);
   static void Main(){
      DelegateType delegateInstance = delegate {
         Console.WriteLine("Hello") ; } ;
      int refVar = 5 ;
      delegateInstance(1, "un", ref refVar) ;
      delegateInstance(2, "deux", ref refVar) ;
   }
}
```

## *Les méthodes anonymes et la généricité*

Une méthode anonyme peut avoir des arguments de type générique. Par exemple :

*Exemple 14-21 :*

```
class Foo<T>{
   delegate void DelegateType(T t) ;
   internal void Fct(T t){
      DelegateType delegateInstance = delegate(T arg){
         System.Console.WriteLine("Hello arg:{0}",arg.ToString());} ;
      delegateInstance(t) ;
   }
```

```
}
class Program{
   static void Main(){
      Foo<double> inst = new Foo <double>() ;
      inst. Fct(5.5) ;
   }
}
```

En C#2, les délégations peuvent admettrent des arguments génériques. Un délégué instance d'une telle délégation peut référencer une méthode anonyme. Il faut alors résoudre les types génériques lors de la définition de la méthode anonyme, comme le montre l'exemple suivant :

*Exemple 14-22 :*

```
class Program{
   delegate void DelegateType<T>( T t );
   static void Main() {
      DelegateType<double> delegateInstance = delegate(double arg) {
         System.Console.WriteLine( "Hello arg:{0}" , arg.ToString()) ;
      } ;
      delegateInstance(5.5) ;
   }
}
```

## *Exemples simples d'utilisation des méthodes anonymes*

L'utilisation de méthodes anonymes est particulièrement adaptée à la définition de « petites » méthodes destinées à être invoquées au moyen d'un délégué. Par exemple, vous pouvez définir au moyen d'une méthode anonyme la procédure sur laquelle démarrera un nouveau thread :

*Exemple 14-23 :*

```
using System.Threading ;
class Program{
   static void Main(){
      Thread thread = new Thread( delegate() {
        System.Console.WriteLine("ManagedThreadId:{0} Hello",
                                 Thread.CurrentThread.ManagedThreadId ) ;
      } ) ;
      thread.Start() ;
      System.Console.WriteLine("ManagedThreadId:{0} Bonjour",
                               Thread.CurrentThread.ManagedThreadId ) ;
   }
}
```

Ce programme affiche :

```
ManagedThreadId:1 Bonjour
ManagedThreadId:3 Hello
```

Vous pouvez aussi définir au moyen d'une méthode anonyme toute sorte de *call-back*, comme la réponse aux événements des contrôles *Windows Forms* :

*Exemple 14-24 :*

```
public class FooForm : System.Windows.Forms.Form{
   System.Windows.Forms.Button m_Button ;
   public FooForm(){
      InitializeComponent() ;
      m_Button.Click += delegate(object sender, System.EventArgs args){
         System.Windows.Forms.MessageBox.Show("m_Button Clicked") ;
      } ;
   }
   void InitializeComponent()  {/*...*/}
}
```

En apparence les méthodes anonymes constituent une possibilité simple ajoutée à C#2. En apparence seulement : entrons dans les arcanes du compilateur C#2 pour saisir toute la puissance de ce mécanisme.

# *Le compilateur C#2 et les méthodes anonymes*

## *Le cas simple*

Comme l'on peut s'en douter, lorsque le compilateur C#2 rencontre une méthode anonyme, il crée une méthode dans la classe de la méthode qui l'encapsule :

*Exemple 14-25 :*

```
class Program {
   delegate void DelegateType() ;
   static void Main() {
      DelegateType delegateInstance = delegate() {
         System.Console.WriteLine("Hello") ; } ;
      delegateInstance() ;
   }
}
```

Ainsi le compilateur C#2 produit l'assemblage suivant à partir du code précédent (l'assemblage est visualisé avec l'outil *Reflector* décrit en page 24) :

On vérifie bien qu'une méthode privée, statique et nommée `<Main>b__0()` a été créée dans la classe encapsulante (i.e la classe `Program`) et contient le code de notre méthode anonyme. Une méthode anonyme est une méthode d'instance si elle est définie dans le corps d'une méthode d'instance.

On note aussi que cette méthode est référencée par le délégué `<>9__CachedAnonymousMethodDelegate1`, instance de `DelegateType`, qui est un champ statique de la classe encapsulante.

Il est intéressant de remarquer que le nom de cette méthode contient une paire de <>. De ce fait le compilateur ne vous autorise pas à l'invoquer directement à partir de votre code et l'intellisense ne l'affiche pas. En revanche le CLR est tout à fait capable d'utiliser une méthode avec un tel nom.

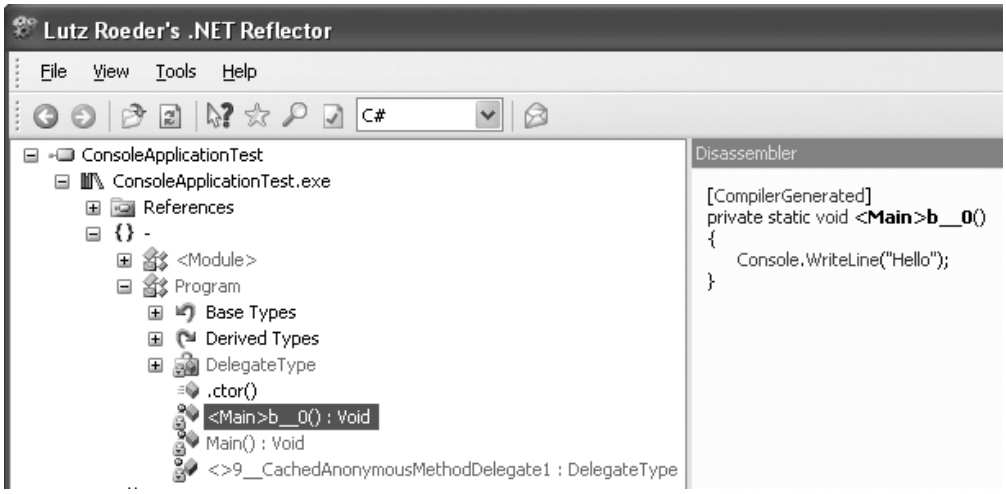


*Figure 14-3 : Cas simple de compilation d'une méthode anonyme*

## *Une méthode anonyme accède à une variable locale de la méthode qui l'encapsule*

Pour ne pas compliquer notre exposé, nous n'avons pas encore mentionné le fait qu'une méthode anonyme a la possibilité d'accéder à une variable locale de la méthode qui l'encapsule. Et c'est bien là que les choses commencent à être intéressantes. Analysons l'exemple suivant :

*Exemple 14-26 :*

```
class Program {
   delegate int DelegateTypeCompteur() ;
   static DelegateTypeCompteur MakeCompteur(){
      int compteur = 0;
      DelegateTypeCompteur delegateInstanceCompteur =
         delegate { return ++compteur; } ;
      return delegateInstanceCompteur ;
   }
   static void Main() {
      DelegateTypeCompteur compteur1 = MakeCompteur() ;
      DelegateTypeCompteur compteur2 = MakeCompteur() ;
      System.Console.WriteLine(compteur1()) ;
      System.Console.WriteLine(compteur1()) ;
      System.Console.WriteLine(compteur2()) ;
      System.Console.WriteLine(compteur2()) ;
   }
}
```

Ce programme affiche :

```
1
2
1
2
```

Cela peut laisser perplexe car la variable `compteur` locale à la méthode `MakeCompteur()` semble « survivre » à l'invocation de celle-ci. Cela contredit complètement la notion de variable locale telle que nous la connaissons. Il semble qu'il existe deux instances de la variable locale `compteur`.

Comme nous l'avons mentionné, en .NET 2.0 il n'y a pas de nouvelles instructions IL pour la gestion des méthodes anonymes. Ce mystère doit donc forcément provenir du compilateur. Une analyse de l'assemblage produit par le compilateur s'impose :

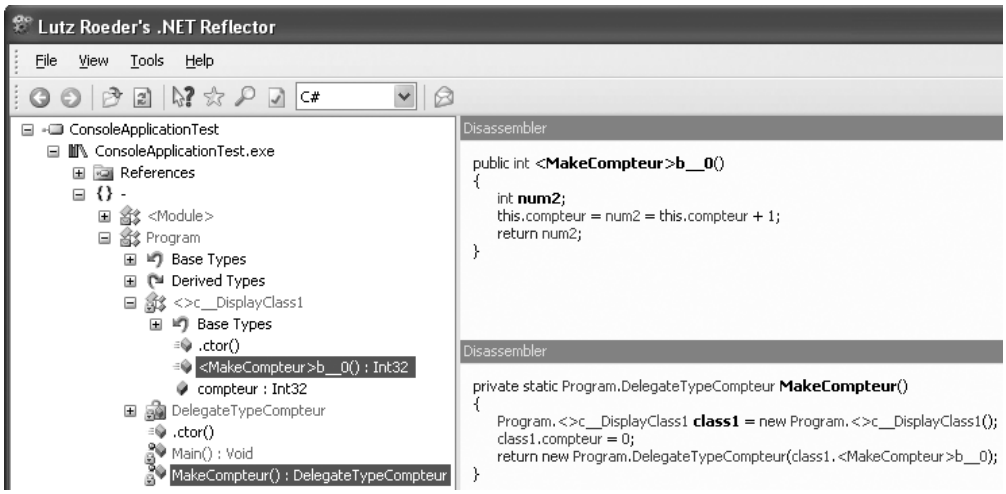


*Figure 14-4 : Cas où une méthode anonyme accède à une variable locale*

Tout devient alors clair :

- Contrairement à la section précédente, le compilateur ne se contente pas de créer une nouvelle méthode. Il crée une nouvelle classe nommée ici `<>c__DisplayClass1`.
- Cette classe contient une méthode d'instance nommée `<MakeCompteur>b__0()` qui a le corps de notre méthode anonyme.
- Cette classe contient aussi un champ d'instance nommé `compteur` qui garde l'état de la variable locale du même nom. On dit que la variable `compteur` est *capturée* par la méthode anonyme.
- La méthode `MakeCompteur()` instancie la classe `<>c__DisplayClass1` et initialise son champ `compteur`.

Il est important de remarquer que la méthode `MakeCompteur()` n'a plus de variable locale `compteur`. Elle utilise le champ `compteur` de la nouvelle instance de `<>c__DisplayClass1`.

Avant d'aller plus loin et d'expliquer pourquoi le compilateur a ce comportement pour le moins inattendu, continuons minutieusement notre analyse.

## *Variable locale capturée et complexité du code*

L'exemple suivant est plus subtil qu'il n'y parait :

*Exemple 14-27 :*

```
using System.Threading ;
class Program {
   static void Main() {
      for (int i = 0 ; i < 5 ; i++)
         ThreadPool.QueueUserWorkItem( delegate {
            System.Console.WriteLine(i) ; }, null) ;
   }
}
```

Ce programme affiche d'une manière non déterministe ceci :

```
0
1
5
5
5
```

On comprend alors que toute les « instances de notre méthode anonyme » partagent la variable locale `i`. Le comportement non déterministe vient du fait que la méthode `Main()` et nos « instances de notre méthode anonyme » sont exécutées par des threads différents. Pour s'en convaincre voici le code décompilé de la méthode `Main()` :

```
private static void Main(){
   bool flag1 ;
   Program.<>c__DisplayClass1 class1 = new Program.<>c__DisplayClass1() ;
   class1.i = 0 ;
   goto Label_0030 ;
Label_000F:
   ThreadPool.QueueUserWorkItem(new WaitCallback(class1.<Main>b__0), null) ;
   class1.i++ ;
Label_0030:
   flag1 = class1.i < 5 ;
   if (flag1){
      goto Label_000F ;
   }
}
```

Notez enfin que la valeur 5 est obtenue lorsque la méthode anonyme est exécutée après que la méthode `Main()` a finit d'exécuter la boucle.

Pour obtenir un comportement déterministe, il suffit de modifier ce programme comme ceci :

*Exemple 14-28 :*

```
using System.Threading ;
class Program {
   static void Main() {
```

```
      for (int i = 0 ; i < 5 ; i++){
         int j = i;
         ThreadPool.QueueUserWorkItem(delegate {
            System.Console.WriteLine(j) ; }, null) ;
      }
   }
}
```

Cette fois ci, le programme affiche bien :

```
0
1
2
3
4
```

Ce comportement vient du fait que la variable locale `j` est capturée à chaque itération comme le montre cette décompilation de la méthode `Main()` :

```
private static void Main(){
   Program.<>c__DisplayClass1 class1 ;
   bool flag1 ;
   int num1 = 0 ;
   goto Label_0029 ;
Label_0004:
   class1 = new Program.<>c__DisplayClass1() ;
   class1.j = num1 ;
   ThreadPool.QueueUserWorkItem(new WaitCallback(class1.<Main>b__0), null) ;
   num1++ ;
Label_0029:
   flag1 = num1 < 5 ;
   if (flag1){
      goto Label_0004 ;
   }
}
```

En matière de capture de variable par une méthode anonyme, rien n'est évident. Cette fonctionnalité est donc à utiliser avec circonspection puisqu'elle peut complexifier la lisibilité de votre code.

Notez enfin qu'une variable locale capturée n'étant plus une variable locale, tout code `unsafe` qui accède à une telle variable doit au préalable l'avoir fixée avec le mot-clé `fixed`.

## *Une méthode anonyme accède à un argument de la méthode qui l'encapsule*

Les arguments d'une méthode peuvent être considérés comme des variables locales. Ainsi, C#2 autorise une méthode anonyme a utiliser des arguments d'une méthode qui l'encapsule. Tout se passe comme si l'argument était une variable locale. Par exemple :

*Exemple 14-29 :*

```
using System ;
class Program {
   delegate void DelegateTypeCompteur() ;
   static DelegateTypeCompteur MakeCompteur(string compteurName) {
      int compteur = 0 ;
      DelegateTypeCompteur delegateInstanceCompteur = delegate{
         Console.WriteLine(compteurName + (++compteur).ToString()) ;
      } ;
      return delegateInstanceCompteur ;
   }
   static void Main() {
      DelegateTypeCompteur compteurA = MakeCompteur("Compteur A:") ;
      DelegateTypeCompteur compteurB = MakeCompteur("Compteur B:") ;
      compteurA() ;
      compteurA() ;
      compteurB() ;
      compteurB() ;
   }
}
```

Ce programme affiche :

```
Compteur A:1
Compteur A:2
Compteur B:1
Compteur B:2
```

Deux restrictions existent cependant quant à la capture d'un argument par une méthode anonyme. L'argument ne doit pas être `out` ou `ref`. Cette conséquence est logique puisqu'un tel argument ne peut plus être considéré comme une variable locale car il « survit » à l'exécution de la méthode.

## *Une méthode anonyme accède à un membre de la classe qui définit la méthode qui l'encapsule*

Une méthode anonyme peut accéder aux membres de la classe qui l'encapsule. Dans le cas de membres statiques il n'y a aucun problème de compréhension. Un champ statique existe en une seule version au sein d'un domaine d'application, et c'est cette version qui est accédée par la méthode anonyme.

Pour bien saisir le cas où une méthode anonyme accède à un membre d'instance de la classe qui l'encapsule, il suffit de considérer la référence `this` comme une variable locale à la méthode d'instance qui définit la méthode anonyme (ce qui est d'ailleurs bien le cas). Par exemple :

*Exemple 14-30 :*

```
delegate void DelegateTypeCompteur() ;
class CompteurBuilder{
   string m_Name ; // Un champ d'instance
```

```
   internal CompteurBuilder(string name) { m_Name = name ; }
   internal DelegateTypeCompteur BuildCompteur(string compteurName) {
      int compteur = 0 ;
      DelegateTypeCompteur delegateInstanceCompteur = delegate {
         System.Console.Write(compteurName +(++compteur).ToString()) ;
         // On aurait pu écrire this.m_Name.
         System.Console.WriteLine(" Compteur fabriqué par:" + m_Name) ;
      } ;
      return delegateInstanceCompteur ;
   }
}
class Program {
   static void Main() {
      CompteurBuilder cBuilder1 = new CompteurBuilder("Fabrique1") ;
      CompteurBuilder cBuilder2 = new CompteurBuilder("Fabrique2") ;
      DelegateTypeCompteur cA = cBuilder1.BuildCompteur("Compteur A:") ;
      DelegateTypeCompteur cB = cBuilder1.BuildCompteur("Compteur B:") ;
      DelegateTypeCompteur cC = cBuilder2.BuildCompteur("Compteur C:") ;
      cA() ;  cA () ;
      cB() ;  cB() ;
      cC() ;  cC() ;
   }
}
```

Ce programme affiche :

```
Compteur A:1 Compteur fabriqué par:Fabrique1
Compteur A:2 Compteur fabriqué par:Fabrique1
Compteur B:1 Compteur fabriqué par:Fabrique1
Compteur B:2 Compteur fabriqué par:Fabrique1
Compteur C:1 Compteur fabriqué par:Fabrique2
Compteur C:2 Compteur fabriqué par:Fabrique2
```

Décompilons la méthode `BuildCompteur()` pour bien montrer que la référence `this` est capturée :

```
internal DelegateTypeCompteur BuildCompteur(string compteurName){
    CompteurBuilder.<>c__DisplayClass1 class1 = new
                                CompteurBuilder.<>c__DisplayClass1() ;
    class1.<>4__this = this ;
    class1.compteurName = compteurName ;
    class1.compteur = 0 ;
    return new DelegateTypeCompteur(class1.<BuildCompteur>b__0) ;
}
```

La référence `this` ne peut pas être utilisée dans une méthode anonyme dont le type encapsulant est une structure (i.e un type valeur). Voici l'erreur générée par le compilateur :

```
Les méthodes anonymes définies dans des structures ne peuvent
accèder au membre 'this'. Néanmoins, vous pouvez copier le membre
'this' dans une variable locale extérieure à la méthode anonyme
et l'utiliser à partir de celle-ci.
```

# Exemples avancés d'utilisation des méthodes anonymes

## Définitions : fermeture (closure en anglais) et environnement lexical

Une *fermeture* (*closure* en anglais) est une fonction qui capture les valeurs des variables de l'*environnement lexical* au moment où elle est créée. L'environnement lexical d'une fonction désigne l'ensemble des variables visibles à l'endroit où la fonction est déclarée.

Notez bien l'utilisation de « au moment » et de « à l'endroit ». Cela nous indique que la fermeture est un concept dynamique (i.e qui n'existe qu'à l'exécution) alors que l'environnement lexical est un concept statique (i.e qui n'existe qu'avant la compilation).

Comprenez bien aussi que la fermeture ne concerne que les variables de l'environnement lexical qui sont effectivement utilisées par la fonction.

La définition d'une fermeture parle de création de fonction. Dans les *langages impératifs* tels que *C, C++, C♯1, Java* ou *VB.NET1* cette notion de création de fonction est absente. Cette notion nous vient des *langages fonctionnels* tels que *Haskell* ou *Lisp* où tout est fonction et où l'état des variables locales des fonctions peut survivre à l'exécution d'une fonction.

Avec ce que l'on a vu concernant le travail du compilateur, la notion de méthode anonyme de C♯2 est bien une implémentation du concept de fermeture. C♯2 est donc plus qu'un langage impératif objet. Notez que ce n'est pas la première fois qu'un langage non fonctionnel intègre la notion de fermeture puisque par exemple les langages *Perl* et *Ruby* supportent cette fonctionnalité.

C♯ → C++ Le concept de fermeture se rapproche aussi du concept de *foncteur* ou de *fonction-objet* du C++ décrit en page 591.

## Un peu plus loin dans la compréhension des fermetures

Lorsqu'une fonction est appelée, elle calcule son résultat en fonction des valeurs de ses arguments mais aussi en fonction du contexte dans lequel elle est invoquée. Le contexte peut être vu comme un ensemble de données d'arrière plan. Ainsi, passer des arguments à une fonction revient à mettre en avant certaines données clés essentielles à l'exécution de la fonction (i.e ses arguments).

Dans un langage objet, pour une méthode d'instance ce contexte est en général l'état de l'objet sur lequel elle est invoquée (état qui est accessible implicitement ou explicitement au travers de la référence `this`). Pour une fonction écrite en C, le contexte est défini par les valeurs des variables globales. Pour une méthode anonyme (i.e une fermeture), ce contexte est défini par les valeurs des variables de son environnement lexical au moment où elle est créée.

Ainsi, de même que la notion de classe, la notion de fermeture est un moyen d'associer un comportement à un état. Dans les langages objets, on associe un ou plusieurs comportements (i.e les méthodes d'instances) à un état (i.e les champs) en passant la référence `this` comme premier argument de la méthode. Ceci nous est caché par le compilateur mais c'est bien ce qui se passe. Pour clarifier tout ceci :

- On peut voir un objet comme un ensemble de données auquel des fonctions sont attachées (par l'astuce du passage de la référence `this` en premier argument).

- On peut voir une fermeture comme une fonction auquel des données sont attachées (par l'astuce de la capture des valeurs des variables de l'environnement lexical).

## *Utilisation des fermetures*

D'après la section précédente, il semblerait que l'on puisse utiliser une méthode anonyme à la place de certaines classes simples à comportement unique. C'est bien ce que l'on faisait dans nos exemples précédents en implémentant un compteur où le comportement est l'incrément du compteur et l'état la valeur du compteur. D'ailleurs nous avons vérifié que la compilation d'une méthode anonyme qui capture un état résulte par la création d'une classe par le compilateur qui n'a qu'une seule méthode.

L'exemple du compteur n'exploite pas la possibilité de passer un argument à une méthode anonyme. En ayant recours à cette possibilité, nous pouvons concevoir une fermeture pour créer un comportement paramétré qui agit sur les instances d'une classe. L'exemple simple est un multiplicateur paramétré d'entiers :

*Exemple 14-31 :*

```
class Program {
   delegate void DelegateMultiplicateur(ref int entierAMultiplier) ;
   static DelegateMultiplicateur BuildMultiplicateur (
                                    int multiplicateurParam){
      return delegate(ref int entierAMultiplier) {
         entierAMultiplier *= multiplicateurParam ;
      } ;
   }
   static void Main() {
      DelegateMultiplicateur multiplicateurPar8=BuildMultiplicateur(8) ;
      DelegateMultiplicateur multiplicateurPar2=BuildMultiplicateur(2) ;
      int entier = 3 ;
      multiplicateurPar8(ref entier) ;
      // Ici entier vaut 24.
      multiplicateurPar2(ref entier) ;
      // Ici entier vaut 48.
   }
}
```

En ayant recours à la possibilité d'utiliser une valeur de retour pour la méthode anonyme, on peut créer des comportements paramétrés qui calculent un résultat en fonction d'un objet. Par exemple :

*Exemple 14-32 :*

```
using System ;
class Article {
   public Article(decimal prix) { m_Prix = prix ; }
   private decimal m_Prix ;
   public decimal Prix { get { return m_Prix ; } }
}
class Program {
```

```
   delegate decimal DelegateTvaComputer(Article article) ;
   static DelegateTvaComputer BuildTvaComputer(decimal tva){
      return delegate(Article article){
         return (article.Prix * (100 + tva)) / 100 ;
      } ;
   }
   static void Main(){
      DelegateTvaComputer tvaComputer19_6 = BuildTvaComputer(19.6m) ;
      DelegateTvaComputer tvaComputer5_5 = BuildTvaComputer(5.5m) ;
      Article article = new Article(97) ;
      Console.WriteLine("Prix TVA 19.6% : "+tvaComputer19_6(article)) ;
      Console.WriteLine("Prix TVA  5.5% : "+tvaComputer5_5(article)) ;
   }
}
```

Comprenez bien que toute la puissance de l'utilisation de fermetures dans les deux exemples précédents vient du fait qu'elles nous évitent de créer des petites classes (qui sont en fait créer implicitement par le compilateur).

## *Délégué et fermeture*

En y regardant de plus prés, on s'aperçoit que la notion de délégué utilisé sur une méthode d'instance en .NET 1.1 est conceptuellement proche de la notion de fermeture. En effet, un tel délégué référence à la fois des données (l'état de l'objet cible) et un comportement. Une contrainte existe cependant : le comportement doit être une méthode d'instance de la classe définissant le type de la référence `this`.

Cette contrainte est affaiblie en .NET 2005. Grâce à certaines surcharges de la méthode `Delegate.CreateDelegate()` vous pouvez maintenant référencer le premier argument d'une méthode statique dans un délégué. Par exemple :

*Exemple 14-33 :*

```
class Program {
   delegate void DelegateType(int writeNTime) ;
   // Cette méthode est déclarée publique pour éviter des
   // problèmes de réflexion sur membres non publics.
   public static void WriteLineNTimes(string s, int nTime) {
      for(int i=0;i<nTime;i++)
         System.Console.WriteLine(s) ;
   }
   static void Main() {
      DelegateType deleg = System.Delegate.CreateDelegate(
         typeof(DelegateType),
         "Bonjour",
         typeof(Program).GetMethod("WriteLineNTimes")) as DelegateType ;
      deleg(4) ;
   }
}
```

Ce programme affiche :

```
Bonjour
Bonjour
Bonjour
Bonjour
```

Notez enfin qu'en interne l'implémentation des délégués a complètement été revue dans la version 2.0 du *framework* et du CLR. La bonne nouvelle est que l'invocation de méthodes au travers de délégués est maintenant beaucoup plus performante.

### *Méthodes anonymes et manipulation de collections*

En page 592 nous montrons comment l'utilisation de méthodes anonymes peut améliorer grandement la syntaxe nécessaire pour manipuler des collections.

## *Les itérateurs avec C♯1*

### *Comprendre les concepts d'énumérable et d'énumérateur*

Avant de nous lancer dans un exemple d'utilisation des itérateurs en C♯1, il est nécessaire de comprendre les concepts d'*énumérable* et d'*énumérateur*.

On dit d'un objet qu'il est énumérable s'il constitue lui-même une collection d'objets et si on peut énumérer cette collection au moyen du mot-clé `foreach`. Concrètement, une condition suffisante (mais pas forcément nécessaire) pour qu'un objet soit énumérable est qu'il implémente l'interface `System.Collections.IEnumerable`. On dit d'un objet que c'est un énumérateur s'il implémente l'interface `System.Collections.IEnumerator`. Voici la définition de ces deux interfaces :

```
public interface System.Collections.IEnumerable{
   System.Collections.IEnumerator GetEnumerator() ;
}
public interface System.Collections.IEnumerator{
   object Current { get ; }
   bool MoveNext() ;
   void Reset() ;
}
```

En analysant ces interfaces, on comprend qu'un client qui veut énumérer la collection d'objets détenue par un objet énumérable, demande à ce dernier un énumérateur au moyen de la méthode `IEnumerator IEnumerable.GetEnumerator()`. Ensuite, le client peut utiliser les méthode `IEnumerator.MoveNext()` et `IEnumerator.Reset()` sur l'énumérateur retourné pour se déplacer dans la collection d'objet. Lorsqu'il veut obtenir un objet, il l'obtient en appelant l'accesseur get de la propriété `Current`. Dans un diagramme UML cela donne :

On vient en fait de décrire exactement le *design pattern itérateur* décrit dans l'ouvrage Gof présenté en page 1031.

### *Un exemple*

Voici un exemple d'implémentation des itérateurs en C♯1. La classe `Personnes` joue le rôle d'énumérable tandis que la classe `PersonnesEnumerator` joue le rôle de l'énumérateur. La classe

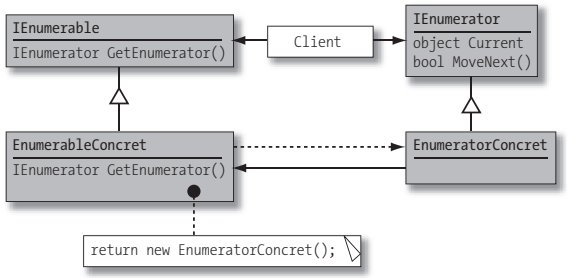


*Figure 14-5 : Le design pattern itérateur*

`Personnes` aurait pu jouer à la fois le rôle de l'énumérable et de l'énumérateur. Il aurait suffit qu'elle implémente en plus l'interface `IEnumerator`. Il est préférable d'isoler l'énumérateur dans une classe tierce dédiée, pour pouvoir implémenter plusieurs énumérateurs pour un même énumérable :

*Exemple 14-34 :*

```
public class Personnes : System.Collections.IEnumerable {
   private class PersonnesEnumerator : System.Collections.IEnumerator {
      private int index = -1 ;
      private Personnes P ;
      public PersonnesEnumerator(Personnes P){ this.P = P ; }
      public bool MoveNext() {
         index++ ;
         return index < P.m_Noms.Length ;
      }
      public void Reset() { index = -1 ; }
      public object Current { get { return P.m_Noms[index] ; } }
   }
   // La méthode GetEnumerator() de IEnumerable.
   public System.Collections.IEnumerator GetEnumerator(){
      return new PersonnesEnumerator(this) ;
   }
   string[] m_Noms ;
   // Le constructeur qui initialise le tableau.
   public Personnes(params string[] Noms){
      m_Noms = new string[Noms.Length] ;
      // Copie le tableau.
      Noms.CopyTo(m_Noms, 0) ;
   }
   // L'indexeur qui retourne le Nom à  partir de l'index.
   private string this[int index]{
      get { return m_Noms[index] ; }
      set { m_Noms[index] = value ; }
   }
}
```

```
class Program {
   static void Main() {
      Personnes arrPersonnes = new Personnes(
                        "Michel", "Christine", "Mathieu", "Julien") ;
      foreach (string s in arrPersonnes)
         System.Console.WriteLine(s) ;
   }
}
```

La syntaxe est un peu lourde au regard de l'envergure de la fonctionnalité. Il est bienvenu que la syntaxe C#2 simplifie ce point. Ce programme affiche :

```
Michel
Christine
Mathieu
Julien
```

Il est clair que le compilateur C# a interprété le mot-clé `foreach` comme ceci :

*Exemple 14-35 :*

```
...
class Program{
   static void Main(){
      Personnes arrPersonnes = new Personnes(
                        "Michel", "Christine", "Mathieu", "Julien") ;
      System.Collections.IEnumerator e = arrPersonnes.GetEnumerator();
      while (e.MoveNext())
         System.Console.WriteLine((string)e.Current) ;
   }
}
```

## *Plusieurs itérateurs sur une même classe*

En fait, l'énumérable n'est pas obligé d'implémenter l'interface `IEnumerable`. On peut délégué cette responsabilité à une classe tierce, `PersonnesEnumerable` par exemple. Voici le programme précédent réécrit :

*Exemple 14-36 :*

```
using System.Collections ;
public class Personnes // N'implémente pas IEnumerable !
   private class PersonnesEnumerator : IEnumerator {
      ...
   }
   private class PersonnesEnumerable : IEnumerable {
      private Personnes m_Personnes ;
      internal PersonnesEnumerable(Personnes personnes) {
         m_Personnes = personnes ;
      }
      IEnumerator IEnumerable.GetEnumerator(){
```

```
            return new PersonnesEnumerator(m_Personnes) ;
        }
    }
    public IEnumerable InOrder{ get {
                            return new PersonnesEnumerable(this) ; } }
    ...
}
class Program{
    static void Main() {
        Personnes arrPersonnes = new Personnes(
                            "Michel", "Christine", "Mathieu", "Julien") ;
        foreach (string s in arrPersonnes.InOrder)
            System.Console.WriteLine(s) ;
    }
}
```

On s'aperçoit que de cette façon il devient possible d'implémenter plusieurs énumérateurs pour notre classe `Personnes`. Par exemple un énumérateur `Reverse` pour parcourir la collection de personnes à l'envers ou bien un énumérateur `Shuffle` pour la traverser de façon aléatoire ou encore un énumérateur `EvenPosOnly` pour la parcourir qu'en ne tenant compte des élément qui ont des positions paires.

## *Problèmes avec les itérateurs de C# v1.x*

En plus d'être assez lourde et malgré le fait que le *design pattern iterateur* est appliqué à la lettre, l'implémentation des itérateurs en C#1 est peu puissante. En effet, il devient vite difficile d'implémenter des itérateurs récursifs pour des collections à peine plus exotiques telles que des arbres.

# *Les itérateurs avec C#2*

## *Un premier exemple avec le mot-clé yield return*

C#2 introduit un nouveau mot-clé `yield return` pour implémenter très simplement le concept d'itérateur. Vous n'avez plus à créer de classe spéciale pour implémenter un énumérateur ou un énumérable. Voici l'Exemple 14-34 réécrit :

*Exemple 14-37 :*

```
public class Personnes : System.Collections.IEnumerable{
    string[] m_Noms ;
    public Personnes(params string[] Noms){
        m_Noms = new string[Noms.Length] ;
        Noms.CopyTo(m_Noms, 0) ;
    }
    // La méthode GetEnumerator() de IEnumerable.
    public System.Collections.IEnumerator GetEnumerator(){
        foreach (string s in m_Noms)
            yield return s;
```

```
   }
}
class Program {
   static void Main() {
      Personnes arrPersonnes = new Personnes(
                     "Michel", "Christine", "Mathieu", "Julien") ;
      foreach (string s in arrPersonnes)
         System.Console.WriteLine(s) ;
   }
}
```

Il est clair que la syntaxe est plus claire qu'en C#1. Cependant l'action du mot-clé `yield return` doit vous paraître étrange. Comment `yield return` peut retourner un énumérateur alors qu'il semble retourner une chaîne de caractères ? Et puis, qu'elle est l'implémentation d'un tel énumérateur puisque clairement, ce programme ne fournit explicitement aucune classe qui implémente `IEnumerator` ? Tout deviendra limpide lorsque l'on procédera à une analyse du travail du compilateur C#2. Présentons d'abord l'étendue des possibilités.

Une méthode peut parfaitement contenir plusieurs fois le mot-clé `yield return`. Ainsi, nous pouvons aussi écrire :

*Exemple 14-38 :*

```
public class Personnes : System.Collections.IEnumerable {
   public System.Collections.IEnumerator GetEnumerator() {
      yield return "Michel" ;
      yield return "Christine" ;
      yield return "Mathieu" ;
      yield return "Julien" ;
   }
}
class Program {
   static void Main() {
      Personnes arrPersonnes = new Personnes() ;
      foreach (string s in arrPersonnes)
         System.Console.WriteLine(s) ;
   }
}
```

D'après les deux exemples précédents, il semble que l'on peut considérer que le programme se branche juste après la dernière instruction `yield return` à chaque itération (et au début pour la première itération). Nous vérifierons que cette intuition est la bonne.

## *Les itérateurs et la généricité*

Il eut été dommage que l'on continue à utiliser la propriété `object Current` de l'interface `IEnumerator` alors que la généricité de C#2 permet d'éviter l'utilisation du type `object` pour signifier, *n'importe quel type*. Ainsi, le *framework* redéfinit les interfaces `IEnumerable` et `IEnumerator` avec la généricité comme ceci :

```
public interface System.Collections.Generic.IEnumerable<T> :
                              System.Collections.IEnumerable {
   System.Collections.Generic.IEnumerator<T> GetEnumerator() ;
}
public interface System.Collections.Generic.IEnumerator<T> :
            System.Collections.IEnumerator, System.IDisposable {
   T Current { get ; }
}
```

On voit que les objets implémentant `IEnumerator<T>` sont disposables. En outre, ces deux interfaces implémentent chacune leurs homologues non générique. Grâce à ces interfaces génériques, nous pouvons maintenant être plus précis et spécifier au compilateur que notre énumérable contient une collection de chaînes de caractères et non d'objets :

*Exemple 14-39 :*

```
using System.Collections.Generic ;
using System.Collections ;
public class Personnes : IEnumerable<string> {
   string[] m_Noms ;
   public Personnes(params string[] Noms) {
      m_Noms = new string[Noms.Length] ;
      Noms.CopyTo(m_Noms, 0) ;
   }
   IEnumerator<string> IEnumerable<string>.GetEnumerator() {
      return PRIVGetEnumerator() ;
   }
   IEnumerator IEnumerable.GetEnumerator() {
      return PRIVGetEnumerator() ;
   }
   private IEnumerator<string> PRIVGetEnumerator() {
      foreach (string s in m_Noms)
         yield return s ;
   }
}
```

Il est dommage de devoir implémenter deux versions de la méthode `GetEnumeror()`. C'est une conséquence immédiate du fait que `IEnumerable<T>` implémente `IEnumerable`. Les ingénieurs de *Microsoft* ont fait ce choix pour s'assurer que tous les types énumérables puissent être utilisés sous une forme non générique. En effet, beaucoup d'API ne prennent en paramètre qu'un `IEnumerable` tandis que beaucoup d'API ne retournent qu'un `IEnumerable<T>`. Sans cette astuce il n'y aurait pas de conversion implicite d'un `IEnumerable<T>` en un `IEnumerable` et les développeurs auraient du souvent préciser explicitement une telle conversion. Ce choix a donc été fait pour privilégier ceux qui utilisent les classes énumérables au détriment de ceux qui les développent. Empiriquement, en tant que développeur, vous vous retrouvez bien plus souvent dans la première situation que dans la deuxième.

## *Plusieurs itérateurs pour une même classe conteneur*

Comme en C#1, il est possible d'implémenter plusieurs énumérateurs pour un même énumérable. Voici un exemple :

*Exemple 14-40 :*

```
public class Personnes{
   string[] m_Noms ;
   public Personnes(params string[] Noms){
      m_Noms = new string[Noms.Length] ;
      Noms.CopyTo(m_Noms, 0) ;
   }
   public System.Collections.Generic.IEnumerable<string> Reverse {
      get {
         for (int i = m_Noms.Length - 1 ; i >= 0 ; i--)
            yield return m_Noms[i] ;
      }
   }
   public System.Collections.Generic.IEnumerable<string> PosPaires {
      get{
         for (int i = 0 ; i < m_Noms.Length ; i++,i++)
            yield return m_Noms[i] ;
      }
   }
   public System.Collections.Generic.IEnumerable<string> Concat {
      get{
         foreach (string s in Reverse)
            yield return s ;
         foreach (string s in PosPaires)
            yield return s ;
      }
   }
}
class Program {
   static void Main() {
      Personnes arrPersonnes = new Personnes(
                  "Michel", "Christine", "Mathieu", "Julien") ;
      System.Console.WriteLine("-->Itérateur Reverse") ;
      foreach (string s in arrPersonnes.Reverse)
         System.Console.WriteLine(s) ;
      System.Console.WriteLine("-->Itérateur PosPaires") ;
      foreach (string s in arrPersonnes.PosPaires)
         System.Console.WriteLine(s) ;
      System.Console.WriteLine("-->Itérateur Concat") ;
      foreach (string s in arrPersonnes.Concat)
         System.Console.WriteLine(s) ;
   }
}
```

Ce programme affiche :

```
-->Itérateur Reverse
Julien
Mathieu
```

```
Christine
Michel
-->Itérateur PosPaires
Michel
Mathieu
-->Itérateur Concat
Julien
Mathieu
Christine
Michel
Michel
Mathieu
```

## *Le mot-clé yield break*

Il est possible que l'on ne souhaite énumérer qu'une partie des objets d'un énumérable. Pour signifier à une boucle `foreach` de s'arrêter il suffit d'employer le mot-clé `yield break`.

*Exemple 14-41 :*

```
...
   public IEnumerator<string> GetEnumerator() {
      for (int i = 0 ; i < 2;i++ )
         yield return m_Noms[i] ;
      yield break;
      // Warning : Unreachable code detected.
      System.Console.WriteLine("hello") ;
   }
...
```

Ce programme affiche :

```
Michel
Christine
```

Le code situé dans le corps d'une méthode après le mot-clé `yield break` n'est pas atteignable (i.e il ne sera jamais exécuté). Si à l'instar de cet exemple vous écrivez du code non atteignable, le compilateur émet un avertissement.

## *Contraintes syntaxique imposées par l'utilisation des mot-clés yield return et yield break*

Les mots-clés `yield break` et `yield return` ne peuvent apparaître que dans le corps d'une méthode, le corps d'un accesseur d'une propriété ou dans le corps d'un opérateur.

Les mots-clés `yield break` et `yield return` ne peuvent apparaître dans le corps d'une méthode anonyme.

Les mots-clés `yield break` et `yield return` ne peuvent apparaître dans une clause `finally`.

Les mots-clés `yield break` et `yield return` ne peuvent apparaître dans une clause `try` qui admet au moins une clause `catch`.

Les mots-clés `yield break` et `yield return` ne peuvent apparaître dans une méthode qui a des arguments `ref` ou `out`. Concrètement, il ne faut pas qu'une telle méthode puisse retourner autre chose qu'un énumérateur.

## *Exemple d'un itérateur récursif*

Voici un exemple qui montre toute la puissance des itérateurs de C♯2 pour énumérer une structure de données non plate, telle qu'un arbre binaire :

*Exemple 14-42 :*

```
using System.Collections.Generic ;
public class Node<T>{
   public Node(T item , Node<T> leftNode , Node<T> rightNode){
      m_Item = item ;
      m_LeftNode = leftNode ;
      m_RightNode = rightNode ;
   }
   public Node<T> m_LeftNode ;
   public Node<T> m_RightNode ;
   public T m_Item ;
}
public class BinaryTree<T> {
   Node<T> m_Root ;
   public BinaryTree(Node<T> Root){
      m_Root = Root ;
   }
   public IEnumerable<T> InOrder {
      get{
         return PrivateScanInOrder(m_Root) ;
      }
   }
   private IEnumerable<T> PrivateScanInOrder(Node<T> root) {
      if (root.m_LeftNode != null) {
         foreach (T item in PrivateScanInOrder(root.m_LeftNode)){
            yield return item ;
         }
      }
      yield return root.m_Item ;
      if (root.m_RightNode != null){
         foreach (T item in PrivateScanInOrder(root.m_RightNode)){
            yield return item ;
         }
      }
   }
}
class Program {
   static void Main() {
      BinaryTree<string> binaryTree = new BinaryTree<string> (
         new Node<string>( "A",
```

```
            new Node<string>( "B" , null , null ),
            new Node<string>( "C" ,
               new Node<string>( "D" , null , null ),
               new Node<string>( "E" , null , null ) ) ) ) ;
      foreach (string s in binaryTree.InOrder)
         System.Console.WriteLine(s) ;
   }
}
```

L'arbre binaire construit dans la méthode `Main()` est celui ci :

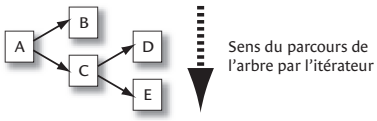


*Figure 14-6 : Arbre binaire*

Le programme affiche ceci :

```
B
A
D
C
E
```

# *Interprétation des itérateurs par le compilateur de C#2*

Voici la section qui va expliquer ce que fait le compilateur C#2 lorsqu'il rencontre les mots-clé `yield break` et `yield return`. Il faut garder à l'esprit qu'aucune instruction IL n'a été rajoutée par rapport à .NET 1.1 pour implémenter cette fonctionnalité (contrairement aux génériques où le *Common Type System* ainsi que le jeu d'instruction IL ont été revus).

## *La classe énumérateur est implémentée automatiquement par le compilateur*

Il est temps de préciser que toute méthode qui contient une instruction `yield` doit retourner un énumérable ou un énumérateur. En conséquence, toute méthode qui contient une instruction `yield` doit avoir pour type de retour une des interfaces `System.Collections.Generic.IEnumerable<T>`, `System.Collections.IEnumerable`, `System.Collections.Generic.IEnumerator<T>` ou `System.Collections.IEnumerator`. Dans tous les cas, cela donne lieu au retour d'un énumérateur puisque la seule raison d'implémenter un énumérable et de pouvoir fournir un énumérateur.

Comme vous pouvez vous en doutez si vous avec lu les sections concernant les méthodes anonymes, pour toute méthode qui contient une instruction `yield` le compilateur crée une classe. Cette classe implémente les quatre interfaces d'énumérations si la méthode retourne une des deux interfaces énumérables. Dans ce cas, une instance de cette classe se retourne elle-même

lorsqu'on la considère comme un énumérable et qu'on lui demande un énumérateur. Cette classe n'implémente que les deux interfaces d'énumérations si la méthode retourne une des deux interfaces d'énumération.

La méthode n'existe plus en tant que telle, mais son corps est alors dans la méthode `MoveNext()` de la nouvelle classe. Vérifions tout ceci sur un exemple :

*Exemple 14-43 :*

```
class Foo{
   public System.Collections.Generic.IEnumerable<string> UnIterateur(){
      yield return "str1" ;
      yield return "str2" ;
      yield return "str3" ;
   }
}
class Program {
   static void Main() {
      Foo collec = new Foo() ;
      foreach (string s in collec.UnIterateur())
         System.Console.WriteLine(s) ;
   }
}
```

Le compilateur C♯2 produit l'assemblage suivant à partir du code précédent (l'assemblage est visualisé avec l'outil *Reflector* décrit en page 24) :

Décompilons la méthode `UnIterateur()` pour vérifier qu'elle crée bien une instance de la classe `Foo.<UnIterateur>d__0` (notez l'utilisation automatique du pattern *using/dispose* pour disposer l'itérateur) :

```
class Foo {
   public System.Collections.Generic.IEnumerable<string> UnIterateur() {
      Foo.<UnIterateur>d__0 d__1 = new Foo.<UnIterateur>d__0(-2);
      d__1.<>4__this = this ;
      return d__1 ;
   }
   ...
}
class Program {
   private static void Main() {
      Foo foo1 = new Foo() ;
      using (IEnumerator<string> enumerator1 =
         ( ( IEnumerator<string>)foo1.UnIterateur().GetEnumerator() ) ) {
         while (enumerator1.MoveNext()) {
            string text1 = enumerator1.get_Current() ;
            Console.WriteLine(text1) ;
         }
      }
   }
}
```

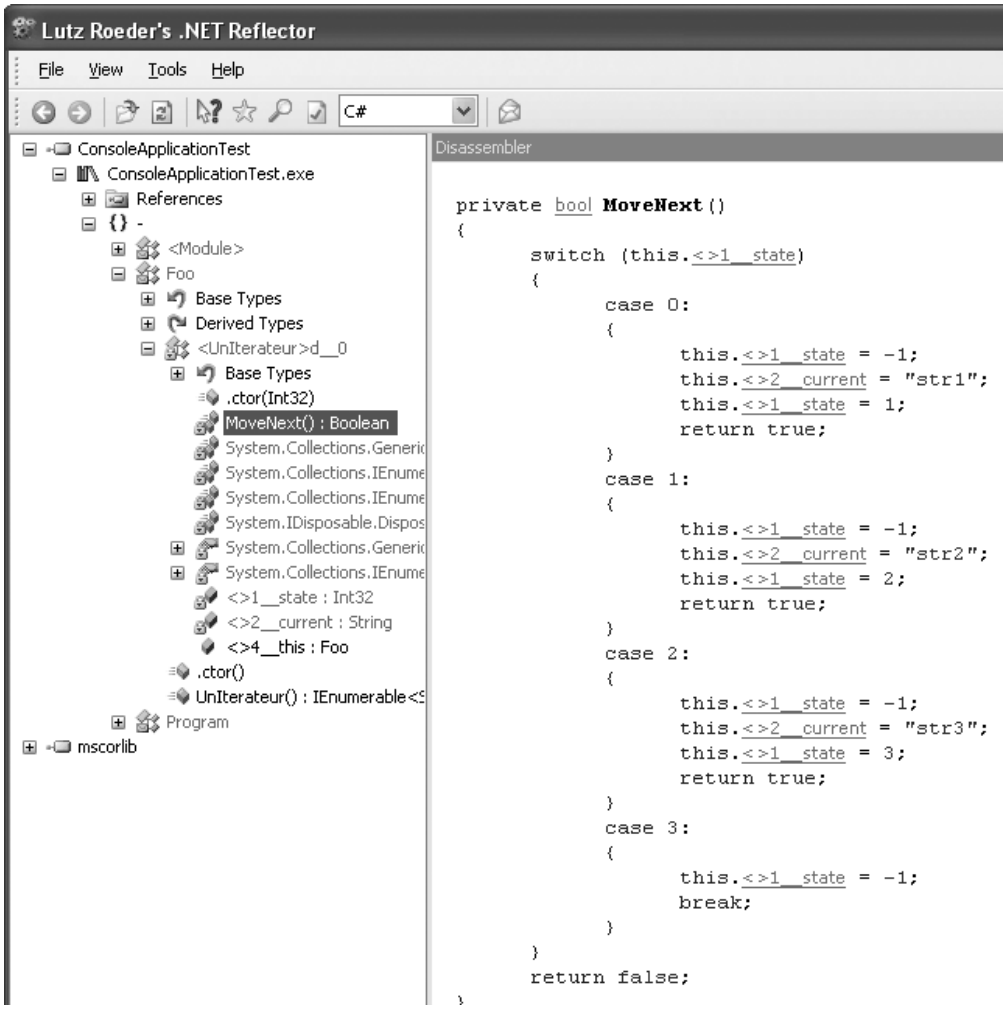


*Figure 14-7 : Interprétation des itérateurs par le compilateur C# 2.0 (1)*

Voici un autre exemple tout aussi instructif :

*Exemple 14-44 :*

```
class Foo {
   public System.Collections.Generic.IEnumerable<int> UnIterateur() {
      for (int i = 0 ; i < 5 ; i++){
         if (i == 3) yield break ;
         yield return i ;
      }
   }
```

```
}
class Program {
   static void Main() {
      Foo collec = new Foo() ;
      foreach (int i in collec.UnIterateur())
         System.Console.WriteLine(i) ;
   }
}
```

Le compilateur C♯2 produit l'assemblage suivant à partir du code précédent :

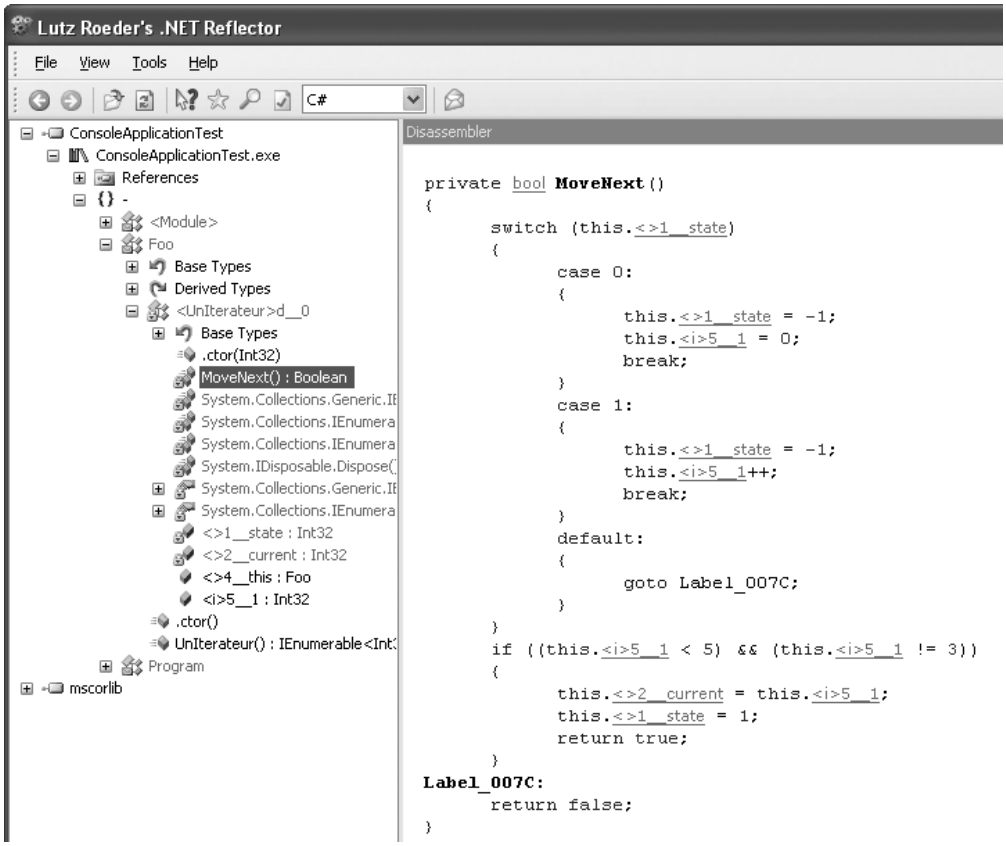


*Figure 14-8 : Interprétation des itérateurs par le compilateur C♯ 2.0 (2)*

## *Une machine à état est fabriquée pour chaque énumérateur*

Il est clair que le compilateur implémente notre méthode contenant un mot-clé `yield`, comme une machine à état. Pour cela, il a besoin de deux champs d'instances `<>1__state` et `<>2__current`. `<>1__state` définit l'état dans lequel se trouve la machine. Rappelez vous, nous avions déduit que le programme se branche juste après la dernière instruction `yield return` à chaque

itération et au début pour la première itération. Cette magie est possible grâce au champ `<>1__state`. Le compilateur insère une instruction `switch` dés le début de la méthode pour que le programme se branche au bon endroit. Le compilateur positionne `<>1__state` pour la prochaine itération avant de quitter une itération.

Le champ `<>2__current` représente la valeur calculée par la dernière itération. Il est donc forcément du type de ce qu'on énumère (`int32` dans nos deux derniers exemples).

Si notre entité énumérable est un objet, i.e si la méthode qui contient une instruction `yield` est une méthode d'instance (comme dans le premier cas), le compilateur produit un champ `<>4__this` qui référence l'objet énumérable. Dans le cas contraire aucun champ n'est créé.

Enfin, remarquez que si la méthode qui contient une instruction `yield` a des variables locales ou des arguments, ceux-ci sont capturés par le compilateur qui en fait alors des champs de la classe qu'il créée. Par exemple le champs `<i>5__1` capture la variable locale `i` dans le second exemple. Ceci est tout à fait comparable à la capture de l'environnement lexical au moyen d'une fermeture que l'on a vu dans le cas des méthodes anonymes. La classe fabriquée par le compilateur est donc bien une fermeture et nul doute que l'on va pouvoir exploiter tout ceci pour aller plus loin que le concept d'itération.

# *Exemples avancés de l'utilisation des itérateurs de C♯2*

D'après la section précédente nous pouvons déduire deux propriétés intéressantes des itérateurs de C♯2 absentes du mécanisme d'itération de C♯1 :

- Les itérateurs de C♯2 supportent le pattern *lazy évaluation*. C'est-à-dire que les éléments de la collection peuvent n'être produits que si besoin est. Autrement dit, comme un itérateur a un état (à l'instar d'une méthode anonyme), les éléments peuvent être calculés un par un, au fur et à mesure des demandes du client. Ils n'ont pas besoin d'être résidents en mémoire.
- Les itérateurs de C♯2 peuvent itérer sur une collection d'éléments à priori infinie (i.e bornée par les limites de la machine).

Nous allons exploiter ces propriétés afin d'utiliser les itérateurs dans d'autres contextes que l'itération sur les éléments d'un énumérable.

## *Définitions : coroutine et de continuation*

Les itérateurs de C♯2 sont en fait une implémentation de la notion de *coroutine*. Une coroutine est une fonction qui a la particularité de reprendre son exécution là où elle s'était terminée la dernière fois qu'on l'a exécuté. La notion de coroutine s'oppose à la notion plus connue de *subroutine*. Une subroutine est une fonction qui recommence au début à chaque exécution.

La notion de coroutine est une spécialisation de la notion de fermeture, puisque pour reprendre là où l'on s'était arrêté, il faut bien pouvoir sauver l'état des variables locales (i.e le contexte d'exécution).

Le terme *continuation* désigne un contexte à partir duquel un programme peut poursuivre son exécution. Une continuation est donc l'ensemble des valeurs des variables locales union l'offset d'une instruction dans le corps d'une coroutine. Une continuation est donc comparable à une pile d'un thread au niveau d'une fonction (*stack frame* en anglais). Cette remarque se relèvera pertinente.

## *Un exemple de continuation avec les itérateurs*

Supposons une implémentation simplifiée d'un jeu d'échec. Les blancs commencent, puis les noirs jouent, puis les blancs jouent, puis les noirs jouent etc. Juste avant que les blancs jouent un coup, il peut être utile de sauver les résultats des calculs afin de pouvoir reprendre là où l'on en était lorsque les noirs auront joués. En C#1 il faudrait prévoir une classe pour stocker ces résultats. En C#2, grâce aux notions de coroutine et de continuation, il suffit juste d'une méthode contenant le mot-clé `yield` :

*Exemple 14-45 :*

```
using System ;
using System.Collections ;
public class Program{
   static IEnumerator White(){
      int resultatCalcul = 0 ;
      while (true){
         Console.WriteLine("white move, résultatCalcul=" +
                           resultatCalcul) ;
         resultatCalcul++ ;
         yield return black;
      }
   }
   static IEnumerator Black(){
      while (true){
         Console.WriteLine("black move") ;
         yield return white ;
      }
   }
   static IEnumerator black;
   static IEnumerator white;
   static void Main() {
      black = Black();
      white = White();
      IEnumerator enumerator = white ; // Honneur aux blancs.
      // On dispatche 5 fois.
      for (int i = 0 ; i < 5;i++){
         enumerator.MoveNext() ;
         enumerator = (IEnumerator)enumerator.Current ;
      }
   }
}
```

Ce programme affiche :

```
white move, résultatCalcul=0
black move
white move, résultatCalcul=1
black move
white move, résultatCalcul=2
```

Il faut bien comprendre le rôle des champs statiques `black` et `white`. Chaque fois que l'on appelle la méthode `White()`, un nouvel itérateur est créé. Il faut donc n'appeler cette méthode qu'une seule fois et référencer l'itérateur retourné avec le champ `white`.

Remarquez aussi que si nous ne limitions pas artificiellement le nombre de coups dans la méthode `Main()` (i.e si l'on remplaçait la boucle `for(int i=0 : i<5 ;i++)` par la boucle `while(true)`) les méthodes `White()` et `Black()` s'appelleraient en alternance sans fin. Si ces méthodes s'appelaient d'une manière plus « traditionnelle », ce comportement ferait rapidement exploser la pile du thread. Ici il n'en ait rien. Cela vient du fait que la notion de continuation peut se comprendre aussi comme une sorte de `goto` inter méthodes (rappelons qu'en C# le l'utilisation du mot-clé `goto` doit être confinée dans le corps d'une méthode).

## *Le pattern Pipeline*

Les itérateurs sont particulièrement adaptés au pattern *pipeline* que vous connaissez tous pour l'avoir utilisez dans vos fenêtres de commande shell. Par exemple :

*Exemple 14-46 :*

```
using System.Collections.Generic ;
class Program{
   static public IEnumerable<int> PipelineIntRange(int begin,int end) {
      System.Diagnostics.Debug.Assert(begin < end) ;
      for(int i=begin;i<=end ;i++)
         yield return i ;
   }
   static public IEnumerable<int> PipelineMultiply(int factor ,
                                                   IEnumerable<int> input){
      foreach (int i in input)
         yield return i * factor ;
   }
   static public IEnumerable<int> PipelineFilterModulo(int modulo ,
                                                   IEnumerable<int> input ){
      foreach (int i in input)
         if( i%modulo == 0 )
            yield return i ;
   }
   static public IEnumerable<int> PipelineJoin(IEnumerable<int> input1,
                                                   IEnumerable<int> input2){
      foreach (int i in input1)
         yield return i ;
      foreach (int i in input2)
         yield return i ;
   }
   static void Main(){
      foreach (int i in PipelineJoin(
         PipelineIntRange(-4, -2), PipelineFilterModulo( 3,
                                      PipelineMultiply( 2,
                                      PipelineIntRange(1, 10) ) ) ) )
         System.Console.WriteLine(i) ;
```

```
   }
}
```

Le programme affiche ceci :

```
-4
-3
-2
6
12
18
```

On comprend bien -4,-3 et -2 mais il est un peu plus compliqué de saisir le pourquoi du 6,12 et 18. Voici une explication schématisée :

```
PipelineIntRange(1,10) produit      1  2  3  4  5  6  7  8  9  10
PipelineMultiply(2) produit         2  4  6  8  10 12 14 16 18 20
PipelineFilterModulo(3) produit     6        12    18
```

En faisant l'expérience de modifier le `PipelineIntRange` comme ceci...

*Exemple 14-47 :*

```
...
   static public IEnumerable<int> PipelineIntRange(int begin, int end){
      System.Diagnostics.Debug.Assert(begin < end) ;
      for (int i = begin ; i <= end ; i++){
         System.Console.WriteLine("Production de:" + i) ;
         yield return i ;
      }
   }
...
using System.Collections.Generic ;
class Program{
   static public IEnumerable<int> PipelineIntRange(int begin, int end){
      System.Diagnostics.Debug.Assert(begin < end) ;
      for (int i = begin ; i <= end ; i++){
         System.Console.WriteLine("Production de:" + i);
         yield return i ;
      }
   }
   static public IEnumerable<int> PipelineMultiply(int factor ,
                                                   IEnumerable<int> input){
      foreach (int i in input)
         yield return i * factor ;
   }
   static public IEnumerable<int> PipelineFilterModulo(int modulo ,
                                                   IEnumerable<int> input ){
      foreach (int i in input)
         if( i%modulo == 0 )
            yield return i ;
```

```
    }
    static public IEnumerable<int> PipelineJoin(IEnumerable<int> input1,
                                                IEnumerable<int> input2){
       foreach (int i in input1)
          yield return i ;
       foreach (int i in input2)
          yield return i ;
    }
    static void Main(){
       foreach (int i in PipelineJoin(
          PipelineIntRange(-4, -2), PipelineFilterModulo( 3,
                                    PipelineMultiply( 2,
                                    PipelineIntRange(1, 10) ) ) ) )
          System.Console.WriteLine(i) ;
    }
}
```

...on obtient cette sortie :

```
Production de:-4
-4
Production de:-3
-3
Production de:-2
-2
Production de:1
Production de:2
Production de:3
6
Production de:4
Production de:5
Production de:6
12
Production de:7
Production de:8
Production de:9
18
Production de:10
```

On voit bien qu'à aucun moment les entiers produits par un maillon du pipeline ne sont stockés d'une quelconque manière. Dés que le producteur initial (i.e `PipelineIntRange`) crée un entier, ce dernier est consommé immédiatement. Chaque maillon est un producteur et un consommateur d'entier (mis à part `PipelineIntRange` qui n'est que producteur d'entier).

## *Continuation vs. Threading*

Lorsque l'on parle de modèle producteur/consommateur, on est en général dans un contexte multithread. Cela fait deux fois que l'on parle de thread (rappelez vous, nous avons vu qu'une continuation est comparable à une pile d'un thread en train d'exécuter une fonction).

En fait, il est possible d'implémenter certains patterns de concurrence avec les itérateurs de C♯2. Voici un autre exemple simple où un producteur calcule les nombres de la suite de *Fibonacci* tandis qu'un consommateur affiche ces valeurs sur la console :

*Exemple 14-48 :*

```
using System.Collections ;
using System.Threading ;
public class Program{
   static AutoResetEvent eventProducterDone=new AutoResetEvent(false) ;
   static AutoResetEvent eventConsumerDone =new AutoResetEvent(false) ;
   static int currentFibo ;
   static void Fibo(){
      int i1 = 1 ;
      int i2 = 1 ;
      currentFibo = 0 ;
      // Le producteur enclanche le mécanisme.
      eventProducterDone.Set() ;
      while(true) {
         // On attend que le consommateur ait fini.
         eventConsumerDone.WaitOne() ;
         // On produit.
         currentFibo = i1 + i2 ;
         i1 = i2 ;
         i2 = currentFibo ;
         // On signale que l'on a produit.
         eventProducterDone.Set() ;
      }
   }
   static void Main() {
      Thread threadProducteur = new Thread(Fibo);
      threadProducteur.Start() ;
      for (int i = 1 ; i < 10 ; i++) {
         // On attend que le producteur ait fini.
         eventProducterDone.WaitOne();
         // On consomme.
         System.Console.WriteLine(currentFibo);
         // On signale que l'on a consommé.
         eventConsumerDone.Set();
      }
   }
}
```

Remarquez que l'état du thread producteur (i.e les valeurs de `i1` et `i2`) est stocké à tous moment sur sa pile. Voici la même problématique implémentée à l'aide d'un itérateur :

*Exemple 14-49 :*

```
using System.Collections.Generic ;
public class Program {
   static IEnumerator<int> Fibo(){
```

```
      int i1 = 1 ;
      int i2 = 1 ;
      int currentFibo = 0 ;
      while (true){
         currentFibo = i1 + i2 ;
         i1 = i2 ;
         i2 = currentFibo ;
         // On signale que l'on a produit.
         yield return currentFibo ;
      }
   }
   static void Main() {
      IEnumerator<int> e = Fibo() ;
      for (int i = 1 ; i < 10 ; i++){
         // On donne la main au producteur pour qu'il produise.
         e.MoveNext() ;
         // On consomme.
         System.Console.WriteLine(e.Current) ;
      }
   }
}
```

Cette fois ci, les valeurs de `i1` et `i2` sont à tous moment stockées dans l'énumérateur créé par l'appel à la méthode `Fibo()`.

## *Une limitation des itérateurs C♯2*

La limitation présentée dans la présente section ne sera pas gênante dans l'immense majorité des itérateurs créés. Pour la mettre en évidence, essayons d'utiliser un itérateurs récursifs pour appliquer le *crible d'Eratosthène* qui permet de calculer les nombres premiers. Le principe du crible est simple. Voici un schéma explicatif :

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 |
  3   5   7   9    11    13    15    17 | 3 est premier,
                                        | 4,6,8,10,12,14,16,18 et
                                        | 20 sont multiples de 2.
      5   7        11    13          17 | 5 est premier, 9 et 21
                                        | sont multiples de 3.
          7        11    13          17 | 7 est premier, il n'y
                                        | a pas de multiple de 5.
                   11    13          17 | 11 est premier, il n'y
                                        | a pas de multiple de 7.
                         13          17 | 13 est premier, il n'y
                                        | a pas de multiple de 11.
                                     17 | 17 est premier, il n'y
                                        | a pas de multiple de 13.
```

Le premier nombre de chaque étape est un nombre premier. À chaque étape on enlève ce nombre ainsi que tous ses multiples.

L'idée est d'avoir autant de *pipeline* que de nombres premiers et de faire la cascade sur un premier itérateur qui produit les entiers de 2 à *n*. Pour évaluer le nombre de pipeline nécessaire (i.e le nombre de nombres premiers entre 2 et *n*) on se sert d'un théorème conjecturé par *Gauss* et démontré par *de la Vallée Poussin* qui affirme que si $P(n)$ désigne le nombre de nombres premiers inférieurs à *n* alors :

$$P(n) \approx \frac{n}{\ln n}$$

Voici le programme :

*Exemple 14-50 :*

```
using System ;
using System.Collections.Generic ;
class Program {
   static public IEnumerable<int> PipelineIntRange(int begin, int end){
      System.Diagnostics.Debug.Assert(begin < end) ;
      for (int i = begin ; i <= end ; i++)
         yield return i ;
   }
   static public IEnumerable<int> PipelinePrime(IEnumerable<int> input){
      using (IEnumerator<int> e = input.GetEnumerator()){
         e.MoveNext() ;
         int premier = e.Current ;
         // Le premier nombre obtenu est forcément un premier.
         Console.WriteLine(premier) ;
         if (premier != 0){
            while (e.MoveNext()){
               // Elimine tous les multiples de premier.
               if (e.Current % premier != 0)
                  yield return e.Current ;
            }
         }
      }
   }
   const int N = 100 ;
   static void Main() {
      // Applique la formule de Gauss/de la Vallée Poussin
      // pour obtenir le nombre d'itérateur.
      int N_PREMIER = (int)Math.Floor( ((double)N)/Math.Log(N) ) ;
      // Produit un pipeline de N_PREMIER PipelinePrime
      // chainé avec un PipelineIntegerRange.
      // Chaque appel à PipelinePrime produit un itérateur.
      List<IEnumerable<int>> list = new List<IEnumerable<int>>();
      list.Add(PipelinePrime( PipelineIntRange(2, N)) ) ;
      for( int i=1 ; i<N_PREMIER ; i++ )
         list.Add( PipelinePrime(list[i-1]) ) ;
      // Traverse toutes les valeurs du dernier itérateur dans la chaine
      // afin de provoquer la cascade des calculs.
```

```
        foreach ( int i in list[N_PREMIER-1]) ;
    }
}
```

La limitation de C♯2 mise en évidence ici est l'impossibilité pour un itérateur de se référencer lui-même. En effet, on ne peut pas utiliser le mot clé `this` dans le corps d'une méthode contenant une instruction `yield`. Dans le cas d'une méthode d'instance, le mot clé `this` référence l'instance courante. Dans le cas d'une méthode statique le compilateur génère une erreur. Tous se passe comme si l'on n'était pas sensé savoir qu'une classe est produite par le compilateur ! On pourrait utiliser la réflexion pour récupérer une référence mais cette solution n'est pas satisfaisante.

# Le framework .NET

# 15

# *Collections*

L'utilisation de collections est fondamentale dans le développement. On a très souvent à manipuler des données structurées en listes, en tableaux, en dictionnaires etc. Ce chapitre présente les différentes classes du *framework* qui répondent aux différents besoins des développeurs.

Nous présentons d'abord les similitudes dans la manipulation des différents types de collections. Ensuite, nous présentons les tableaux, leur utilisation en C# ainsi que les concepts sous-jacents de .NET. Puis nous présentons les collections qui stockent leurs éléments dans une séquence tel qu'une liste, une file d'attente ou une pile. Enfin nous présentons des collections particulières appelées dictionnaires.

Durant notre exposé, nous nous efforçons de souligner les différences entre chaque implémentation afin de vous aider à choisir quel type de collection est le mieux adapté à chacun de vos besoins. Vous pouvez aussi consulter l'article **Selecting a Collection Class** des **MSDN** lorsque vous hésitez entre plusieurs implémentations.

L'ensemble des types de collections proposé par le *framework* peut s'avérer assez limité dans le cadre de certaines applications. Par exemple, il n'y a pas d'implémentation des classiques arbres binaires ou ensemble. En cherchant sur internet, vous trouverez sûrement des implémentations directement réutilisables. Notamment, vous pouvez avoir recours au *framework open-source* **Power Collections** de l'entreprise **Wintellect** qui a pour but de combler certaines lacunes du *framework* (`http://www.wintellect.com/powercollections/`).

## *Parcours des éléments d'une collection avec « foreach » et « in »*

Grâce aux mots-clés `foreach` (« pour chaque » en français) et `in`, C# dispose d'une syntaxe particulièrement conviviale pour parcourir les éléments d'une collection. L'utilisation de `foreach` et de `in` est possible sur n'importe quel type qui implémente l'interface `System.Collections.IEnumerable`.

Il est de votre responsabilité de veiller à ce que la taille d'une collection ne change pas durant le parcours des éléments de celle-ci. En d'autres termes, vous devez synchroniser les accès à vos collections entre les différents threads d'un programme.

## *Exemple d'utilisation de foreach et in sur un tableau*

Par exemple, pour calculer la somme des éléments d'un tableau d'entiers à une dimension il suffit d'écrire :

*Exemple 15-1 :*

```
using System ;
class Program {
   static void Main() {
      int[] tab = { 1, 3, 4, 8, 2 } ;
      // Calcul de la somme des éléments du tableau.
      int somme = 0 ;
      foreach ( int i in tab )
         somme += i ;
      // somme vaut : 1+3+4+8+2 = 18
   }
}
```

`foreach` permet aussi le parcours des éléments de tableaux à plusieurs dimensions. L'ordre dans lequel les éléments sont parcourus est illustré par l'exemple suivant :

*Exemple 15-2 :*

```
class Program {
   static void Main() {
      string[,] tab = { { "do", "ré", "mi" }, { "fa", "sol", "la" } } ;
      foreach ( string s in tab )
         System.Console.Write(s + ",") ;
   }
}
```

L'affichage de ce programme est :

```
do,ré,mi,fa,sol,la,
```

## *foreach et les tableaux irréguliers*

La notion de tableau irrégulier est définie un peu plus loin dans ce chapitre. `foreach` permet aussi le parcours d'un tableau irrégulier. Néanmoins, il faut prendre en compte qu'un tableau irrégulier est un tableau de tableaux. En général on utilise des boucles `foreach` imbriquées pour parcourir les éléments d'un tableau irrégulier. Par exemple :

*Exemple 15-3 :*

```
class Program {
   static void Main() {
      int[][] t1 = new int[3][] ;
```

```
      t1[0] = new int[12] ;
      t1[1] = new int[5] ;
      t1[2] = new int[9] ;
      int somme = 0 ;
      foreach ( int[] tab in t1 )
         foreach ( int i in tab )
            somme += i ;
   }
}
```

Le bénéfice de l'utilisation de `foreach` est que l'on n'a pas besoin de connaître la taille des dimensions d'un tableau pour pouvoir le parcourir. Il n'y a aucun risque de débordement. Ce bénéfice est encore plus significatif lors du parcours de tableaux irréguliers.

La variable définie dans une boucle `foreach` n'est accessible qu'en lecture. Cela veut dire que dans notre exemple, nous ne pouvons réaliser aucune affectation ni sur la référence tab, ni sur l'entier i. En revanche, nous pourrions très bien appeler une méthode ou une propriété sur ces objets qui modifie leur état interne.

### *Implémenter l'utilisation de la syntaxe foreach et in sur vos propres classes*

C# 2.0 présente le concept d'itérateur qui permet d'implémenter facilement l'utilisation de la syntaxe foreach et in sur vos propres classes. Ce concept d'itérateur fait l'objet de la section page 539.

## *Les tableaux*

### *Références vers un tableau et création de tableaux*

C# permet la création et l'utilisation de *tableaux* à une ou plusieurs dimensions. Voici la syntaxe qui permet de déclarer une référence vers un type de tableau :

```
int [] r1 ;      // r1 est une référence vers un tableau d'entiers de

                 // dimension 1.
int [,] r2 ;     // r1 est une référence vers un tableau d'entiers de

                 // dimension 2.
int [,,] r3 ;    // r1 est une référence vers un tableau d'entiers de

                 // dimension 3.
double [,,,] r4;// r4 est une référence vers un tableau de doubles de

                 // dimension 4.
```

Il est essentiel de comprendre que les types `int[]`, `int[,]`, `int[,,]` et `double[,,,]` représentent des classes (donc des types référence) qui sont fabriquées et gérées à l'exécution par le CLR. De plus, chacune de ces classes dérive de la classe abstraite `System.Array` que nous décrivons un peu plus loin. Cela entraîne les remarques suivantes :

- L'allocation d'un tableau ne se fait que lorsque vous utilisez l'opérateur `new`. Les lignes de code **ci-dessus** n'allouent pas un seul tableau. Elles définissent quatre références, chacune vers un type de tableau particulier. Lorsque vous créez un tableau, la taille de chacune des dimensions doit être précisée soit statiquement par une constante entière, soit dynamiquement par une variable d'un type entier. Par exemple :

  *Exemple 15-4 :*

```
public class Program {
   public static void Main() {
      byte i = 2 ;
      long j = 3 ;

      // t1 est un tableau à une dimension d'éléments de type 'int'.
      // Il contient 6 éléments.
      int [] t1 = new int [6] ;

      // t2 est un tableau à deux dimensions d'éléments
      // de type 'double'. Il contient 12 éléments (j=3 et 3x4=12).
      double [,] t2 = new double [j,4] ;

      // t3 est un tableau à 3 dimensions d'éléments des références de
      // type 'object'.Il contient 30 éléments(j=3 et i=2 et 3x5x2=30).
      object [,,] t3 = new object [j,5,i] ;
   }
}
```

- Les tableaux sont alloués sur le tas et non sur la pile. L'allocation de certains tableaux sur la pile est possible dans des situations très particulières (voir page 508).
- La responsabilité de la désallocation d'un tableau incombe au ramasse-miettes.
- Chaque type de tableau est dérivé de la classe `object` puisque la classe `System.Array` dérive de la classe `object`.
- On ne copie pas physiquement un tableau avec l'opérateur d'affectation « = ». Avec cet opérateur on ne fait qu'obtenir une autre référence vers le même tableau.
- Le passage d'un tableau à une méthode se fait toujours par référence, même si le mot-clé `ref` n'est pas utilisé.

C# oblige tous les éléments d'un tableau à avoir le même type. Cependant cette contrainte peut être facilement contournée. Il suffit de spécifier que les éléments sont des références vers une classe de base (resp. vers une interface), pour qu'en fait, chaque élément puisse être une référence vers un objet de n'importe quelle classe dérivée (resp. de n'importe quelle classe qui implémente l'interface).

## *Accès aux éléments d'un tableau et détection des débordements*

Comme dans la plupart des langages, la syntaxe d'accès aux éléments d'un tableau est la même que ce soit en écriture ou en lecture. Cette syntaxe est assez similaire à celle de l'utilisation des indexeurs d'une classe. Par exemple :

```
int [,] tab = new int[2,3] ;
tab[2,1] = 5 ;       // Accés en écriture.
int i = tab[2,1] ;   // Accés en lecture.
```

Notez surtout qu'à l'instar de la plupart des langages, en C#, les tableaux sont « zéro indexés ». Cela veut dire que si la taille d'une dimension est un nombre entier N, un index sur cette dimension prend les N valeurs entières 0,1,2...,N-1. De plus le type d'un index est forcément un type entier. Enfin sachez que l'accès aux éléments d'un tableau à partir d'index se fait en un temps constant, (i.e indépendamment du nombre d'éléments du tableau).

Un avantage d'être exécuté par une machine virtuelle telle que le CLR est qu'elle vérifie systématiquement lors d'un accès à un élément qu'il n'y a pas de débordement du tableau dans aucune des dimensions. Dans le cas d'un débordement, une exception de type `System.IndexOutOfRangeException` est automatiquement lancée par le CLR. Par exemple :

```
int [,] tab = new int[2,3] ;
// Débordement sur la première dimension, une exception est lancée.
tab[13,1] = 5 ; short i = 14 ;
// Débordement sur la deuxième dimension, une exception est lancée.
int e = tab[0,i] ;
```

## *Les tableaux irréguliers (jagged arrays)*

On définit par le terme *tableau irrégulier* (*jagged array* en anglais) un tableau unidimensionnel dont les éléments sont des tableaux. Certains articles utilisent aussi le terme de *tableau déchiqueté* pour désigner un tableau irrégulier. La Figure 15-1 montre un tableau de trois éléments dont le premier élément est un tableau à 12 éléments, le deuxième un tableau à 5 éléments et le troisième un tableau à 9 éléments.

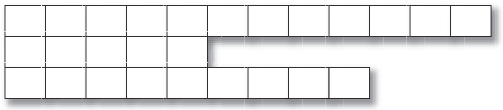

*Figure 15-1 : Représentation d'un tableau irrégulier*

C# a une syntaxe intuitive pour définir les tableaux irréguliers. Par exemple on pourrait référencer le tableau de la Figure 15-1 avec la référence `t1` suivante (en considérant que les éléments sont des entiers) :

```
int [][] t1 ;
```

La construction de ce tableau est effectuée par le code suivant :

```
int [][] t1 = new int [3][] ;
t1[0] = new int[12];
t1[1] = new int[5] ;
t1[2] = new int[9] ;
```

Voici d'autres types de tableaux irréguliers :

```
int [][,,] t1 ;
string [,,][][] t2 ;
double [][][,,][,,,] t3 ;
```

Comme vous pouvez le constater, la représentation mentale des tableaux irréguliers est plus compliquée que l'idée de tableau multidimensionnel. En outre, contrairement aux tableaux multidimensionnels, les tableaux irréguliers ne sont pas « *CLS compliant* ». Les autres langages ne les implémentent pas nécessairement. Il faut donc restreindre leur utilisation à l'intérieur d'un assemblage C# et ne jamais les mettre en arguments de méthodes exposées par un assemblage. Cependant, l'utilisation de tableaux irréguliers est plus performante car le langage IL contient des instructions spécialement optimisées pour la manipulation de tableau unidimensionnel. L'exemple suivant exhibe un facteur d'optimisation supérieur à 2 :

*Exemple 15-5 :*

```
using System;
using System.Diagnostics;
class Program {
   const int N_ITERATION = 10000;
   const int N_ELEM = 100;
   static void Main() {
      int tmp = 0;

      int[][] arrayJagged = new int[N_ELEM][];
      for (int i = 0; i < N_ELEM; i++)
         arrayJagged[i] = new int[N_ELEM];
      Stopwatch sw = Stopwatch.StartNew();
      for (int k = 0; k < N_ITERATION; k++)
         for (int i = 0; i < N_ELEM; i++)
            for (int j = 0; j < N_ELEM; j++){
               tmp = arrayJagged[i][j];
               arrayJagged[i][j] = i * j;
            }
      Console.WriteLine("Tableau irrégulier: " + sw.Elapsed );

      int[,] arrayMultiDim = new int[N_ELEM, N_ELEM];
      sw = Stopwatch.StartNew();
      for (int k = 0; k < N_ITERATION; k++)
         for (int i = 0; i < N_ELEM; i++)
            for (int j = 0; j < N_ELEM; j++){
               tmp = arrayMultiDim[i, j];
               arrayMultiDim[i, j] = i * j;
            }
      Console.WriteLine("Tableau multidimensionnel:" + sw.Elapsed );
   }
}
```

Enfin notez que le CLR détecte les débordements d'accès sur les tableaux irréguliers. Par exemple :

*Exemple 15-6 :*

```
class Program{
    static void Main() {
        int[][] t1 = new int[3][] ;
        t1[0] = new int[12] ;
        t1[1] = new int[5] ;
        t1[2] = new int[9] ;
        int i = t1[0][7] ;  // OK, ici il n'y a pas de débordement.
        int j = t1[1][7] ;  // Débordement sur la deuxième dimension,
    }                       // une exception est lancée.
}
```

## *Initialisation des éléments d'un tableau*

Lorsqu'un tableau a des éléments d'un type valeur, ces éléments sont automatiquement créés lors de la création du tableau. Ils sont par défaut initialisés à zéro.

```
int [] tab = new int [3] ;
int i = tab[0] ;
// Ici i vaut zéro.
```

Dans le cas d'un tableau avec des éléments d'un type référence, les références sont nulles par défaut. Il faut donc initialiser les références une à une avec les objets adéquats.

*Exemple 15-7 :*

```
class Article {
   private decimal m_Prix ;
   public Article(decimal prix){m_Prix = prix;}
}
public class Program {
   public static void Main() {
      Article [] tab = new Article[3] ;
      // Ici aucun article n'a été créé.
      tab[1] = new Article(98.5M) ;
      tab[2] = new Article(190M) ;
      tab[3] = new Article(299.0M) ;
      // Ici le tableau référence les trois articles.
   }
}
```

Ces remarques sont valables que le tableau soit irrégulier ou pas. En revanche ce qui suit n'est valable que pour les tableaux non irréguliers. C# autorise l'écriture de tableaux constants. Par exemple :

```
// tabval est de dimension deux.
int [,] tabval = { {3,4,5} , {7,8,9} } ;
// tabref est de dimension un.
Article [] tabref =
          { new Article(98.5M), new Article(190M), new Article(299.0M) } ;
```

On ne précise pas la taille de chacune des dimensions puisqu'elles sont implicitement contenues dans les tableaux constants. En outre, le compilateur détecte et sanctionne d'une erreur, l'écriture d'un tableau constant irrégulier :

```
// Erreur de compilation.
int [][] tabval = { {3,4} , {7,8,9} } ;
```

## Covariance sur les tableaux

Lorsqu'il existe une relation d'héritage entre deux types référence `B` et `D` (`D` dérive de `B` si `B` est une classe ou `D` implémente `B` si `B` est une interface) alors, il est possible de convertir implicitement un tableau de type `D[]` en un tableau de type `B[]`. Il est aussi possible de convertir explicitement un tableau de type `B[]` en un tableau de type `D[]`. Dans ce cas, le CLR vérifie la validité de la conversion pour chaque élément et produit une exception de type `InvalidCastException` si une telle conversion échoue. Cette possibilité de conversion est appelée *covariance* sur les tableaux.

*Exemple 15-8 :*

```
class B { }
class D1 : B { }
class D2 : B { }
public class Program {
   public static void Main() {
      D2[] td2 = { new D2(), null , new D2() } ;
      B[]  tb  = td2 ;       // Conversion implicite de D2[] vers B[].
      D1[] td1 = (D1[])tb ;  // Conversion explicite de B[] vers D1[].
      // Compile mais une exception de type InvalidCastException
   }  // est lancée à l'exécutition pour la seconde conversion.
}
```

## La classe System.Array

Comme nous l'avons vu, chaque type de tableau est une classe fabriquée à l'exécution par le CLR qui dérive de la classe abstraite `System.Array`. Ceci permet d'utiliser les nombreux membres de cette classe.

### Membres de System.Array

Les propriétés de `System.Array` les plus couramment utilisées sont :

- `int Length{ get ; }`
  Retourne le nombre d'éléments du tableau. Dans les tableaux irréguliers `Length` retourne le nombre de sous tableaux.
- `int Rank{ get ; }`
  Retourne le nombre de dimensions du tableau. Attention, dans les tableaux irréguliers `Rank` retourne le nombre de dimensions du tableau de sous tableaux.

Parmi les méthodes de `System.Array` les plus utilisées on remarque :

- `int GetLength(int dimension)`
  Retourne la taille de la dimension précisée, `dimension` étant le numéro « zéro indexé » de la dimension du tableau (i.e spécifier 0 pour avoir la taille de la première dimension). Si la dimension est négative ou supérieure au rang du tableau, alors l'exception `IndexOutOfRange` est lancée.
- `static void Copy(Array src, Array dest, ...)`
  `void CopyTo(Array dest, ...)`
  Les différentes surcharges de ces méthodes permettent de copier tous les éléments ou une sous séquence d'éléments d'un tableau unidimensionnel dans un autre tableau unidimensionnel.
- `static int IndexOf<T>(Array array, T value, ...)`
  `static int LastIndexOf<T>(Array array, T value, ...)`
  Les différentes surcharges de ces méthodes renvoient l'index de la première (ou de la dernière) occurrence de `value` dans un tableau unidimensionnel (ou dans une sous séquence d'éléments). Attention, soyez conscient que la notion d'égalité est différente selon que `T` est un type valeur ou référence. Si la sous séquence d'éléments précisée n'est pas valide, l'exception `IndexOutOfRange` est lancée. Si `array` a plus d'une dimension, l'exception `RankException` est lancée.
- `static void Reverse(Array array, ...)`
  Les différentes surcharges de cette méthode inversent la séquence (ou une sous séquence) des éléments de `array`. Si la sous séquence d'éléments précisée n'est pas valide, l'exception `IndexOutOfRange` est lancée. Si `array` a plus d'une dimension, l'exception `RankException` est lancée.
- `static int BinarySearch<T>(T[] array,T value, ... )`
  Effectue une recherche dichotomique d'un élément dans un tableau déjà trié (ou dans une sous séquence selon les surcharges). Si `value` est trouvée, son index est retourné. Sinon, la valeur retournée représente l'opposé de « l'index+1 » du plus petit élément plus grand que `value`. Si `value` est plus grand que tous les éléments, la valeur retournée représente l'opposé de la « taille du tableau+1 ». Par exemple:

  *Exemple 15-9 :*

```
public class Program {
   public static void Main() {
      int[] t = new int[2] ;
      t[0] = 10 ;
      t[1] = 20 ;
      int a = System.Array.BinarySearch(t, 5) ;
      int b = System.Array.BinarySearch(t, 10) ;
      int c = System.Array.BinarySearch(t, 15) ;
      int d = System.Array.BinarySearch(t, 25) ;
      // Ici,  a vaut -1    b vaut 0    c vaut -2    d vaut -3.
   }
}
```

- Certaines surcharges de `BinarySearch<T>()` acceptent un objet implémentant `IComparer<T>` pour spécifier l'opération de comparaison.

## *Les membres de System.Array ajoutés avec la version 2.0*

- `static void Resize<T>(ref T[] array, int newSize)`

  Redimensionne un tableau unidimensionnel. Si la nouvelle taille est plus grande que la taille initiale, les nouveaux éléments prennent la valeur par défaut du type `T` (0 pour si type valeur, `null` si type référence). Sinon les éléments en bout de tableau sont supprimés. Si la référence `array` est nulle, alors cette méthode crée un nouveau tableau unidimensionnel de la taille spécifiée.

- `static void ConstrainedCopy(Array src, int srcIndex, Array dest, int destIndex, int length)`

  Cette nouvelle méthode réalise une copie d'un tableau unidimensionnel ou multidimensionnel avec la garantie que le tableau destination n'est pas modifié si l'opération échouait. Pour cela elle utilise le mécanisme de région d'exécution contrainte décrit en page 125. Cette méthode a un coût et ne doit être utilisée qu'à bon escient (dans un contexte *multithreads*). Dans le cas d'une copie d'une sous séquence d'un tableau multidimensionnel, il faut considérer que vous avez à faire à des tableaux unidimensionnels avec les éléments des différentes dimensions mis bout à bout.

- `static IList<T> AsReadOnly(T[] array)`

  Retourne une référence de type `IList<T>` qui permet de n'accéder au tableau sous-jacent qu'en lecture seule. Attention, il s'agit bien du même tableau. Comme l'illustre l'exemple suivant, tous changements sur le tableau initial est visible de la référence `IList<T>` :

  *Exemple 15-10 :*

```
using System.Collections.Generic ;
public class Program {
   public static void Main() {
      int[] t = new int[2] ;
      t[0] = 10 ;
      t[1] = 20 ;
      IList<int> tReadOnly = System.Array.AsReadOnly(t) ;
      t[1] = 25 ;
      // Ici t[1] vaut 25.
      tReadOnly[1] = 30 ; // Ici une exception de type
   }                      // NotSupportedException est lancée.
}
```

Un peu plus loin, nous décrivons de nombreuses nouvelles méthodes de la classe `System.Array` permettant de manipuler simplement et efficacement les éléments d'un tableau.

## *Tableaux non zéro-indexés*

La classe présente plusieurs surcharges de la méthode `CreateInstance()`. Ces surcharges sont utilisées par le compilateur C# pour instancier des tableaux multidimensionnels. Une de ces surcharges n'est jamais utilisée par le compilateur C#. Néanmoins, vous pouvez utiliser explicitement cette surcharge afin de créer des tableaux bidimensionnels non zéros indexés. L'exemple suivant illustre cette possibilité :

*Exemple 15-11 :*

```
public class Program {
   static void Main() {
      int[] lengths = { 4, 5 } ;
      int[] lowerBounds = { -2, 3 } ;
      double[,] arrBiDim = System.Array.CreateInstance(
         typeof(double), lengths, lowerBounds) as double[,] ;
      double d1 = arrBiDim[-2, 5] ;  // OK, les indexes sont valides.
      double d2 = arrBiDim[0, 0] ;   // Lancement de l'exception
   }                                 // IndexOutOfRangeException.
}
```

Le tableau `arrayBiDim` contient 20 éléments (4x5) dont les coordonnées vont de [-2,3] à [1,7] inclus. L'exemple illustre aussi le fait que le CLR détecte les débordements éventuels.

# *Tableaux de bits*

## *La classe System.Collection.BitArray*

Vous pouvez utiliser la classe `System.Array` pour stocker un tableau unidimensionnel de booléens, mais le *framework* .NET met à votre disposition la classe `System.Collections.BitArray` spécialement prévue à cet effet. Les avantages d'utiliser cette classe à la place de `System.Array` sont multiples :

- Pour stocker un tableau de N booléens, l'implémentation *Microsoft* de la classe `BitArray` a besoin d'au plus (N/32)+1 entiers (= 4 octets). En effet, les booléens y sont stockés sous une forme compacte, à raison de 32 booléens par entier. En revanche, chaque booléen est stocké sur un octet lorsque vous utilisez la classe `Array` pour les stocker.
- La classe `BitArray` a des indexeurs entiers. La syntaxe conviviale d'accès aux éléments (en lecture et écriture) avec l'opérateur `[]` reste possible sur les tableaux de type `BitArray`.
- La classe `BitArray` admet plusieurs constructeurs très pratiques pour initialiser des tableaux de booléens.

| | |
|---|---|
| `BitArray(int)` | L'argument spécifie le nombre de bits dans le tableau de bits. Ils sont tous initialisés à `false`. |
| `BitArray(int,bool)` | Le premier argument spécifie le nombre de bits dans le tableau de bits. Le second spécifie la valeur initiale des arguments. |
| `BitArray(bool[])` | Le tableau de type `BitArray` est initialisé à partir d'un tableau de booléens de type `Array`. |
| `BitArray(byte[])` | Le tableau de type `BitArray` est initialisé à partir d'un tableau d'octets de type `Array`. |

- La classe `BitArray` admet plusieurs méthodes très pratiques pour travailler avec des booléens. En voici quelques-unes :

| | |
|---|---|
| `BitArray Not()` | Inverse chaque booléen du tableau. Ceux positionnés à `false` sont positionnés à `true` et vice-versa. |
| `Void SetAll(bool)` | Positionne tous les booléens du tableau à la valeur passée en argument. |
| `BitArray And(BitArray)` | Effectue un « et logique » avec les booléens du tableau passé en argument. Si les deux tableaux ne sont pas de même taille l'exception `System.ArgumentException` est lancée. Les booléens modifiés sont ceux du tableau sur lequel est appelé cette méthode et la référence renvoyée référence ce même tableau. |
| `BitArray Or(BitArray)` | Effectue un « ou logique » avec les booléens du tableau passé en argument. Même remarques que pour la méthode `And()` |
| `BitArray Xor(BitArray)` | Effectue un « ou exclusif » avec les booléens du tableau passé en argument. Même remarques que pour la méthode `And()` |

Voici un exemple pour exposer tout ceci :

*Exemple 15-12 :*

```
using System.Collections ;
public class Program {
   public static void Main() {
      BitArray bArr1 = new BitArray (10,true) ;
      BitArray bArr2 = new BitArray (10,false) ;
      BitArray bArr3 = bArr1.And(bArr2) ;
      // Ici, bArr3[0] vaut false et bArr1[0] vaut false.
      bArr3[0] = true ;
      // Ici, bArr3[0] vaut true et bArr1[0] vaut true.
   }
}
```

## *La structure System.Collections.Specialized.BitVector32*

La structure `BitVector32` permet de contenir un tableau d'exactement 32 bits. Si elle est adaptée à vos besoins, préférez utiliser cette structure plus performante que la classe `BitArray`. Vous pouvez construire une instance de `BitVector32` à partir d'une autre instance de `BitVector32` ou à partir d'un entier. Cette structure présente un indexeur qui vous permet d'accéder en lecture ou en écriture à un bit en spécifiant sa position dans l'intervalle fermé [0,31]. Elle présente aussi des opérations spécifiques à la manipulation de bits avec des masques ou avec des sections.

# *Les séquences*

Nous présentons ici des interfaces puis des classes spécifiques pour la manipulation de tableaux dont la taille peut varier au fur et à mesure de l'ajout ou de la suppression d'éléments. On appelle de tels tableaux des *séquences*.

## *L'interface System.Collections.Generic.ICollection<T>*

```
public interface ICollection<T> : IEnumerable<T>
```

L'interface `System.Collections.Generic.ICollection<T>` est implémentée par toutes les classes représentant des collections. Voici les membres de l'interface `ICollection<T>` :

- `int Count{ get ; }`
  Cette propriété retourne le nombre d'éléments dans la collection.
- `bool IsReadOnly{ get ; }`
  Cette propriété retourne `true` si la collection n'est accessible qu'en lecture seule.
- `void Add( T item )`
  Ajoute un élément à la collection.
- `void Clear()`
  Supprime tous les éléments de la collection.
- `bool Contain( T item )`
  Retourne `true` si la collection contient l'élément spécifié.
- `bool Remove( T item )`
  Supprime la première occurrence de l'élément spécifié. Retourne `true` si une occurrence de l'élément spécifié a été effectivement supprimée.
- `public virtual void CopyTo( T[] array, int index )`
  Copie les éléments de la collection dans le tableau `array`. Il faut que le tableau `array` ait une seule dimension. Si le tableau `array` ne respecte pas la contrainte de la dimension unique, l'exception `System.ArgumentException` est lancée.
- `index` spécifie l'indice du tableau `array` à partir duquel commence la copie. Si le tableau destination n'est pas assez grand (i.e si la taille de `array` — `index` est inférieure au nombre d'éléments de la collection) l'exception `System.ArgumentException` est lancée.
- Si les éléments sont de type référence, ce sont simplement les références qui sont copiées et pas les objets. C'est ce qu'on appelle une *copie superficielle*. Voici un exemple où les éléments de type référence d'une liste, sont copiés dans un tableau :

*Exemple 15-13 :*

```
using System.Collections.Generic ;
class Article{
   public decimal Prix ;
   public Article(decimal Prix) { this.Prix = Prix ; }
}
public class Program {
   public static void Main() {
```

```
        Article a1 = new Article(98.5M) ;
        Article a2 = new Article(190M) ;
        ICollection<Article> collection = new List<Article>() ;
        collection.Add(a1) ;
        collection.Add(a2) ;
        Article[] tableau = new Article[2] ;
        // Copie les éléments de 'collection' dans 'tableau'.
        collection.CopyTo(tableau, 0) ;
        tableau[0].Prix = 80M ;
        decimal d = a1.Prix ;
        // Ici d vaut 80.
    }
}
```

## *L'interface System.Collections.Generic.IList<T>*

```
public interface IList<T> : ICollection<T>, IEnumerable<T>
```

L'interface `System.Collections.Generic.IList<T>` permet de considérer qu'une collection est une liste. L'interface `IList<T>` étend l'interface `ICollection<T>`. L'implémentation de prédilection de cette interface est la classe `System.Collections.Generic.List<T>`, présentée ci-après. Les membres de l'interface `IList<T>` sont :

- `int IndexOf( T item )`
  Retourne l'index de l'élément spécifié (-1 si l'élément n'est pas trouvé).
- `void Insert( int index , T item )`
- Cette méthode insère un élément dans la liste à la position représentée par l'index, augmentant ainsi la taille de la liste d'une unité. Si l'index n'est pas valide, l'exception `ArgumentOutOfRangeException` est lancée. Si l'insertion échoue, en général parce que la liste n'est accessible qu'en lecture seule, l'exception `NotSupportedException` est lancée.
- `void RemoveAt( int index )`
  Supprime l'élément situé à la position spécifiée par l'index. Les exceptions `ArgumentOutOfRangeException` et `NotSupportedException` peuvent être lancées pour les mêmes raisons énoncées dans la méthode précédente.
- `T this[int index] { get; set; }`
- Indexeur qui permet d'accéder en lecture ou en écriture à l'élément situé à la position spécifiée par l'index. Les exceptions `ArgumentOutOfRangeException` et `NotSupportedException` peuvent être lancées pour les mêmes raisons énoncées dans les méthodes précédentes.

## *La classe System.Collections.Generic.List<T>*

La classe `System.Collections.ArrayList` très utilisée dans les applications développées avec .NET 1.x doit maintenant être abandonnée au profit de la classe `System.Collections.Generic.List<T>` définie comme ceci :

```
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>,
                       IList,    ICollection,    IEnumerable
```

Cette classe générique est très optimisée. Notamment, l'utilisation de `List<object>` est bien plus performante que l'utilisation de `ArrayList`.

Contrairement à une instance de la classe `Array`, une instance de `List<T>` a un nombre d'éléments variable. Comme dans tout compromis, cet avantage des listes sur les tableaux se paye. En l'occurrence, cet avantage se paye par une demande d'allocation mémoire lorsque le nombre d'éléments de la liste grandit. En revanche, à l'instar des tableaux, l'accès aux éléments d'une liste à partir d'un index se fait en un temps constant (i.e indépendant du nombre d'éléments). Nous verrons dans la prochaine section un autre modèle de liste implémenté par la classe `LinkedList<T>` où l'insertion se fait un temps constant et où l'accès aux éléments se fait en un temps proportionnel au nombre d'éléments. Vous connaissez maintenant les points clés pour décider si vous devez utiliser la classe `Array`, la classe `List<T>` ou la classe `LinkedList<T>`, selon vos besoins spécifiques.

La classe `List<T>` présente les constructeurs suivants (la notion de capacité est expliquée un peu plus loin dans cette section) :

- `List<T>()`
  Dans l'implémentation *Microsoft*, ce constructeur sans argument construit une instance de capacité 0.
- `List<T>(int capacity)`
  Ce constructeur construit une instance de `List<T>` de capacité `capacity`.
- `ArrayList(IEnumerable<T> collection)`
  Ce constructeur construit une instance de `ArrayList` avec les éléments de la collection énumérable `collection`.

En plus des membres des interfaces `IList<T>` et `ICollection<T>`, la classe `List<T>` présente les membres suivants :

- `T[] ToArray()`
  Equivalent à `ICollection<T>.CopyTo(T[],0)`.
- `int IndexOf(T value, ...)`
  `int LastIndexOf(T value, ...)`
  Même comportement que les différentes surcharges des méthodes `IndexOf()` et `LastIndexOf()` de la classe `Array`., mis à part qu'ici ce sont des méthodes d'instance et non des méthodes statiques.
- `int BinarySearch(T value, ...)`
  Même comportement que les différentes surcharges de la méthode `BinarySearch()` de la classe `Array`.
- `IList<T> AsReadOnly()`
  Mêmes remarques que pour la méthode `AsReadOnly()` de la classe `Array`.
- `void AddRange(IEnumerable collection)`
  `void InsertRange(int index, IEnumerable collection)`
  `List<T> GetRange(int index, int count)`
  `void RemoveRange(int index, int count)`
  Ces méthodes permettent d'insérer, d'ajouter, d'obtenir ou de supprimer un ensemble d'éléments contigus de la liste sur laquelle elles sont appelées. La différence entre l'insertion et l'ajout est que l'ajout insère les éléments à la fin de la liste.

## *Capacité d'une instance de List<T>*

La capacité d'une instance de `List<T>` peut être gérée au moyen de la propriété `Capacity`. Le problème avec les collections de taille variable est que l'allocation/désallocation de mémoire lors de la modification du nombre des éléments nuit aux performances. Pour atténuer les effets de ce problème, il faut diminuer le nombre de demandes d'allocation/désallocation. Ceci est possible en allouant plus de mémoire que nécessaire, pour anticiper sur les futures insertions d'éléments. De même il faut désallouer la mémoire que lorsqu'un certain nombre d'éléments ont été désalloués. La même problématique est exposée pour la classe `StringBuilder` en page 376.

Une conséquence de l'utilisation d'un mécanisme de capacité, est qu'à un instant donné, le nombre d'éléments effectivement contenu dans la liste est différent du nombre d'éléments que la liste peut contenir sans avoir besoin de nouvelles allocations. Le nombre d'éléments effectivement contenus dans la liste, est représenté par la propriété `Count` de l'interface `ICollection`. Le nombre d'éléments que la liste peut contenir sans avoir besoin de nouvelle allocation est représenté par la propriété `Capacity`.

La valeur de `Capacity` est par définition constamment supérieure ou égale à la valeur de `Count`. Aussi, si vous essayez d'affecter une valeur à `Capacity` inférieure à la valeur de `Count` l'exception `ArgumentOutOfRangeException` est lancée. Notez que vous pouvez à tout moment supprimer les éléments non utilisés en appelant la méthode `void TrimExcess()`. La gestion de l'expansion de la capacité est, en général, laissée aux algorithmes internes du *framework* .NET comme le montre l'exemple suivant :

*Exemple 15-14 :*

```
using System.Collections.Generic ;
public class Program {
   public static void Main() {
      List<int> list = new List<int>(3) ;
      for (int i = 0 ; i < 8 ; i++){
         list.Add(i) ;
         System.Console.WriteLine("Count:{0}  Capacité:{1}",
                                  list.Count, list.Capacity ) ;
      }
      list.TrimExcess();
      System.Console.WriteLine("Count:{0}  Capacité:{1}",
                               list.Count, list.Capacity ) ;
   }
}
```

Cet exemple affiche :

```
Count:1  Capacité:3
Count:2  Capacité:3
Count:3  Capacité:3
Count:4  Capacité:6
Count:5  Capacité:6
Count:6  Capacité:6
Count:7  Capacité:12
Count:8  Capacité:12
Count:8  Capacité:8
```

Le constructeur de `List<T>` que nous utilisons dans l'exemple précédent accepte une valeur entière qui représente la capacité initiale. En outre, il est clair que l'implémentation *Microsoft* double le nombre d'éléments à chaque fois qu'une allocation est nécessaire. On dit de la classe `List<T>` qu'elle a un *facteur d'expansion (grow factor* en anglais*)* égal à 2.

## *Les interfaces System.ComponentModel.IBindingList et System.ComponentModel.IListSource*

Les *frameworks Windows Forms 2.0* ainsi qu'*ASP.NET 2.0* présentent des facilités pour lier une source de données à des contrôles d'affichage de données. Une source de données est souvent une liste de données. Pour lier une telle liste de données à un contrôle d'affichage ces technologies requièrent une classe qui implémente IBindingList telle que la classe System.ComponentModel.BindingList<T> (voir page 693) ou une classe qui implémente IListSource tel que la classe DataSet (voir page 929).

## *La classe System.Collections.Generic.LinkedList<T>*

La classe `System.Collections.Generic.LinkedList<T>` représente le concept de *liste doublement liée*. Une liste doublement liée est une collection de nœuds liés les uns aux autres à la façon des maillons d'une chaîne. Chaque nœud est lié au nœud qui le précède et au nœud qui le suit. Chaque nœud connaît la liste liée qui le contient. Chaque nœud contient un élément de la liste. La liste doublement liée ne connaît que le premier et le dernier nœud. Tout ceci est illustré par la figure suivante :

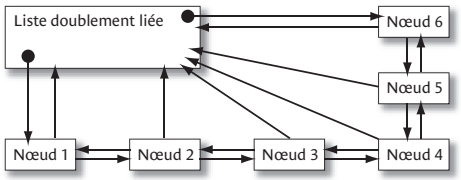


*Figure 15-2 : Une liste doublement liée*

En conséquence, l'accès à un élément se fait en un temps proportionnel au nombre d'éléments tandis que l'insertion ou la suppression d'un élément se fait en un temps constant (i.e indépendant du nombre d'éléments).

La classe `System.Collections.Generic.LinkedList<T>` est définie comme ceci :

```
public class LinkedList<T> : ICollection<T>, IEnumerable<T>, ICollection,
  IEnumerable, ISerializable, IDeserializationCallback
```

La classe `System.Collections.Generic.LinkedListNode<T>` qui représente le concept de noeud est définie comme ceci :

```
public sealed class LinkedListNode<T> {
   public LinkedListNode(T value) ;
   public System.Collections.Generic.LinkedList<T> List { get ; }
   public System.Collections.Generic.LinkedListNode<T> Next { get ; }
```

```
    public System.Collections.Generic.LinkedListNode<T> Previous { get ; }
    public T Value { get ; set ; }
}
```

Remarquez que les propriétés `List`, `Next` et `Previous` ne sont pas accessibles en écriture, ce qui implique que seules les opérations d'une liste doublement liée permettent de lier un nœud à une liste.

En plus des méthodes de l'interface `ICollection<T>` la classe `LinkedList<T>` présente les membres suivants :

- `void AddHead(T value)`
  `void AddHead(LinkedListNode<T> node)`
  `void AddTail(T value)`
  `void AddTail(LinkedListNode<T> node)`
  Ajoute un élément en tête de liste avec `AddHead()` ou en bout de liste `AddTail()`.
- `void AddAfter(LinkedListNode<T> node, T value)`
  `void AddAfter(LinkedListNode<T> node,LinkedListNode<T> _node)`
  `void AddBefore(LinkedListNode<T> node, T value)`
  `void AddBefore(LinkedListNode<T> node,LinkedListNode<T> _node)`
  Ajoute un élément avant ou après un élément donné. L'exception `InvalidOperationException` est lancée si `node` n'appartient pas à la liste concernée.
- `void RemoveHead()`
  `void RemoveTail()`
  Supprime l'élément en tête de liste avec `RemoveHead()` ou en bout de liste avec `RemoveTail()`. L'exception `InvalidOperationException` est lancée si la liste est vide.

## *Liste de chaînes de caractères*

Vous pouvez hésiter entre deux implémentations de listes de chaînes de caractères proposées par le *framework* : La classe `System.Collections.Specialized.StringCollection` (disponible depuis la version 1.0) et la classe `System.Collections.Generic.List<string>` (disponible seulement depuis la version 2.0).

La réponse est simple. Il faut préférer la version générique List<string>. Pas tant à cause de ses performances (qui sont à peine meilleures que celles de StringCollection) mais plutôt pour des raisons de cohérence de design puisque peu à peu, toutes vos collections vont être de type générique. En outre, List<string> implémente des interfaces génériques telles que IEnumerable<string> non supportées par `StringCollection`.

L'utilisation de la classe `StringCollection` n'a maintenant de sens que si vous souhaitez rester cohérent avec du code utilisant déjà cette classe.

## *La classe System.Collections.Generic.Queue<T>*

La notion de *file d'attente* est souvent utilisée en programmation. Une file d'attente sert en général à traiter des messages dans leur ordre d'arrivé, de la même manière qu'un commerçant sert ses clients dans l'ordre dans lequel ils sont arrivés. Le terme classique pour désigner ce comportement est *FIFO* (*First In First Out* que l'on peut traduire par premier arrivé premier servi). À

l'instar d'une file d'attente chez un commerçant, nous parlerons de début de file d'attente et de fin de file d'attente.

Pour reproduire un tel comportement, il est conseillé d'utiliser la classe `System.Collections.Generic.Queue<T>` prévue à cet effet et définie comme ceci :

```
public class Queue<T> : ICollection<T>, IEnumerable<T>,
                        ICollection,    IEnumerable
```

Cette classe présente principalement les trois méthodes suivantes :

- `void Enqueue(T item)`
  Cette méthode ajoute l'élément `item` à la fin de la file d'attente.
- `T Dequeue()`
  Cette méthode retire l'élément qui est au début de la file d'attente et le renvoie. Si la file d'attente est vide, l'exception `InvalidOperationException` est lancée.
- `T Peek()`
  Cette méthode renvoie l'élément qui est au début de la file d'attente sans le retirer. Si la file d'attente est vide, l'exception `InvalidOperationException` est lancée.

À l'instar de la classe `List<T>`, la classe `Queue<T>` a une capacité et implémente la méthode `TrimToSize()` pour minimiser les baisses de performances provoquées par les allocations/désallocations induites par les changements de taille de la file d'attente. La classe `Queue<T>` présente aussi un constructeur qui permet de fixer la capacité initiale.

## *La classe System.Collections.Generic.Stack<T>*

La notion de *pile* (*stack* en anglais) est souvent utilisée en programmation. Par exemple chaque thread utilise une pile pour stocker ses données de traitement. Une pile sert en général à traiter des messages dans l'ordre de dépôt sur la pile, de la même manière qu'une pile de dossiers est traitée. En général on commence par le dossier du dessus de la pile. Le terme classique pour désigner ce comportement est *LIFO* (*Last In First Out* que l'on peut traduire par dernier arrivé premier servi). À l'instar d'une pile de dossier, nous parlerons du sommet de la pile.

Pour reproduire un tel comportement il est conseillé d'utiliser la classe `System.Collections.Generic.Stack<T>` prévue à cet effet et définie comme ceci :

```
public class Queue<T> : ICollection<T>, IEnumerable<T>,
                        ICollection,    IEnumerable
```

Cette classe présente principalement les trois méthodes suivantes :

- `void Push(T item)`
  Cette méthode ajoute l'élément `item` sur le sommet de la pile.
- `T Pop()`
  Cette méthode retire l'élément qui est situé au sommet de la pile et le renvoie. Si la pile est vide, l'exception `InvalidOperationException` est lancée.
- `T Peek()`
  Cette méthode renvoie l'élément qui est situé au sommet de la pile sans le retirer. Si la pile est vide, l'exception `InvalidOperationException` est lancée.

À l'instar de la classe `List<T>` et de la classe `Queue<T>`, la classe `Stack<T>` dispose d'une capacité et implémente la méthode `TrimExcess()` pour minimiser les baisses de performances provoquées par les allocations/désallocations induites par les changements de taille de la pile. La classe `Stack<T>` présente aussi un constructeur qui permet de fixer la capacité initiale.

# *Les dictionnaires*

Les dictionnaires sont des collections dont les éléments sont des couples clé valeur. Dans un couple, la clé sert à indexer la valeur. Par exemple dans les dictionnaires d'une langue, les mots sont les clés et les définitions les valeurs. Dans les classes que nous présentons dans cette section, les clés et les valeurs peuvent être de types quelconques. Précisons qu'un dictionnaire ne peut contenir plusieurs couples clé/valeur ayant la même clé.

Les deux principales caractéristiques d'un dictionnaire sont :

- l'insertion rapide d'un couple clé/valeur.
- la recherche rapide d'une valeur à partir de sa clé.

La rapidité est cruciale, car on utilise des dictionnaires avant tout pour des raisons de performance. Pour optimiser l'insertion et la recherche, deux familles d'algorithmes existent. Dans le *framework* .NET ils donnent lieu aux deux implémentations présentées dans cette section.

## *L'interface Sytem.Collections.Generic.IDictionary<K,V>*

Les deux implémentations de dictionnaires, à savoir les classes `System.Collections.Generic.SortedDictionary<K,V>` et `System.Collections.Generic.Dictionary<K,V>`, implémentent l'interface `System.Collections.Generic.IDictionary<K,V>` définie comme suit :

```
public interface IDictionary<K, V> : ICollection<KeyValuePair<K, V>>,
                                     IEnumerable<KeyValuePair<K, V>>
```

Ainsi, un dictionnaire peut être vu comme une collection d'instances de la structure `System.Collections.Generic.KeyValuePair<K,V>` définie comme suit :

```
public struct KeyValuePair<K, V>{
   public K Key ;
   public V Value ;
}
```

Les méthodes de l'interface `IDictionary<K,V>` sont les suivantes :

- `void Add(K key, V value)`
  Ajoute le couple `key/value` au dictionnaire. Si un couple clé/valeur existe déjà avec cette clé, l'exception `ArgumentException` est lancée. Si le dictionnaire sous-jacent est en lecture seule, l'exception `NotSupportedException` est lancée.
- `bool Remove(K key)`
  Supprime le couple ayant pour clé l'argument `key`. Si un tel couple n'est pas présent, cette méthode retourne `false`. Si le dictionnaire sous-jacent est en lecture seule, l'exception `NotSupportedException` est lancée.

- `bool ContainsKey(K key)`
  Retourne `true` si le dictionnaire contient un couple ayant pour clé l'argument `key`.

L'interface `IDictionary<K,V>` présente aussi les propriétés suivantes :

- `V this[ K key ] {get;set;}`
  Indexeur pour accéder en lecture et en écriture aux couples clé/valeur à partir des clés.
- `ICollection<K> Keys {get;}`
  Retourne une collection contenant toutes les clés du dictionnaire.
- `ICollection<V> Values {get;}`
  Retourne une collection contenant toutes les valeurs du dictionnaire.

## *La classe System.Collections.Generic.SortedDictionary<K,V>*

L'implémentation `System.Collections.Generic.SortedDictionary<K,V>` du concept de dictionnaire, repose sur le fait qu'il doit exister une relation d'ordre total sur les clés. Les couples clé/valeur sont constamment ordonnés selon l'ordre total des clés. L'algorithme dichotomique peut ainsi être appliqué lors de l'insertion d'un couple clé/valeur et lors de la recherche d'une clé. L'avantage principal de cet algorithme et qu'il est performant puisqu'il aboutit en un temps logarithmique. C'est cet algorithme que l'on applique intuitivement lorsque l'on recherche un mot dans un dictionnaire de la langue Française par exemple. Dans ce cas, l'ordre total sur les clés est l'ordre alphabétique des mots du dictionnaire. Il nous permet en quelques secondes de trouver parmi des dizaines de milliers de définitions, la définition correspondante au mot cherché.

Clairement, l'implémentation de `SortedDictionary<K,V>` doit être à même de comparer deux instances du type `K`. Si les clés ne supportent pas l'interface `IComparable<K>` ou si le comparateur par défaut `Comparer<K>.Default` ne convient pas, vous pouvez utiliser certains constructeurs de `SortedDictionary<K,V>` qui acceptent un paramètre de type `IComparer<K>`. Cette interface est décrite un peu plus loin.

La classe `SortedDictionary<K,V>` est définie comme suit :

```
public class SortedDictionary<K, V> : IDictionary<K, V>,
     ICollection<KeyValuePair<K, V>>, IEnumerable<KeyValuePair<K, V>>,
     IDictionary, ICollection, IEnumerable
```

En plus des membres de l'interface `IDictionary<K,V>`, la classe `SortedDictionary<K,V>` présente la méthode `bool TryGetValue(K key, out V value)` qui retourne une valeur en fonction de sa clé. Si la clé est trouvée, la valeur de retour est `true`. Sinon, elle vaut `false`.

## *La classe System.Collections.Generic.Dictionary<K,V>*

L'implémentation de la classe `System.Collections.Generic.Dictionary<K,V>` est basée sur le concept de *table de hachage* que nous allons décrire. Ainsi, elle corrige certains problèmes potentiels inhérents à l'implémentation de la classe `SortedDictionary<K,V>` :

- Nous n'avons pas toujours une relation d'ordre total sur les clés du dictionnaire.
- Si l'opération de comparaison de deux clés est une opération coûteuse, les performances d'utilisation de `SortedList<K,V>` sont dégradées.

- L'insertion et la recherche dans une table de hachage se font en un temps en général constant (i.e indépendamment du nombre d'éléments), ce qui est mieux que le temps logarithmique de l'algorithme dichotomique de l'implémentation de `SortedDictionary<K,V>`.

Voici la définition de la classe `Dictionary<K,V>` :

```
public class Dictionary<K, V> : IDictionary<K, V>,
   ICollection<KeyValuePair<K, V>>, IEnumerable<KeyValuePair<K, V>>,
   IDictionary, ICollection, IEnumerable,
   ISerializable, IDeserializationCallback
```

## *Les tables de hachage*

L'idée d'une table de hachage est de calculer une valeur de hachage sur la clé du dictionnaire, à l'insertion ou à la recherche. La valeur de hachage correspondant à la clé est un entier. Une fois que la clé est hachée en un entier, on utilise un algorithme de recherche en temps constant, basé sur certaines propriétés des groupes finis d'ordre premier.

Cependant il peut toujours arriver que deux clés différentes aient la même valeur de hachage entière. On nomme ce phénomène *collision* entre clés. Pour pallier ce problème, le concept de table de hachage introduit la notion de panier (*bucket* en anglais). La table de hachage contient plusieurs paniers indexés par les valeurs de hachage. Un panier indexé par la valeur de hachage H contient tous les couples clé/valeur dont la clé a pour valeur de hachage H. Ces relations d'inclusion sont exposées par la Figure 15-3.

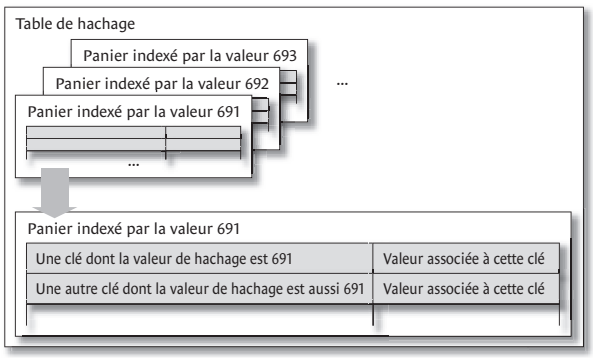


*Figure 15-3 : Table de hachage, paniers et couples clé/valeur*

Cette notion de panier est interne à l'implémentation de la table de hachage et le développeur n'aura jamais à manipuler de paniers. Pour les curieux sachez que la magie de la recherche en temps constant découle d'un théorème d'arithmétique qui dit que l'ordre de tout élément d'un groupe fini d'ordre premier est égal à l'ordre du groupe. Ainsi, l'implémentation de `Dictionary<K,V>` maintient en interne un nombre premier de paniers qui varie dans le temps selon le nombre de paniers courant. En appliquant le théorème précédent, le panier dans lequel doit être contenu une valeur dont la clé admet une valeur de hachage H a pour index H modulo le nombre courant de paniers.

Avec cet algorithme, l'opération d'insertion d'un couple clé/valeur est une opération en temps constant s'il n'y a pas de collision. L'opération de recherche d'une clé se fait en trois étapes :

- D'abord l'obtention de la valeur de hachage de la clé ;
- ensuite la recherche du panier (en temps constant) ;
- enfin, une fois le panier trouvé, la recherche séquentielle dans le panier s'il y a collision.

## *Notion de facteur de chargement*

Nous souhaitons vous sensibiliser ici sur un détail important des algorithmes de tables de hachage. D'après ce que nous avons présenté, dans une table de hachage, plus le rapport (nombre de couple clé/valeur) / (nombre de panier) est petit, plus les recherches seront rapides, car moins il y aura de collisions. Malheureusement plus ce rapport est petit, plus la table de hachage consomme de la place mémoire et donc, en définitive, nuit au performance. Ce compromis est géré en interne dans la classe `Dictionary<K,V>` par ce que l'on nomme le *facteur de chargement* (*load factor* en anglais).

À chaque insertion d'un couple clé/valeur dans une table de hachage, le rapport #couples/#paniers augmente. Lorsque ce rapport dépasse le facteur de chargement, le nombre de panier est automatiquement augmenté pour être égal au plus petit nombre premier plus grand que deux fois le nombre de paniers courant.

La classe `Dictionary<K,V>` est optimisée et vous n'avez pas la possibilité d'agir sur le facteur de chargement. En revanche, certains constructeurs protégés de la classe `Dictionary<K,V>` acceptent la valeur de la capacité initiale. Si la valeur initiale de la capacité n'est pas un nombre premier, elle sera automatiquement positionnée à un nombre premier supérieur à cette valeur.

## *Méthodes GetHashCode() de la classe object*

La classe `object` présente les deux méthodes Equals() et GetHashCode(), utilisées sur le type K par les algorithmes de la classe `Dictionary<K,V>`. Si vous décidez de redéfinir la méthode Equals() sur une classe dont les instances sont susceptibles de représenter des clés dans une table de hachage, il est impératif que vous redéfinissiez la méthode GetHashCode() de façon à satisfaire la contrainte suivante : pour toutes instances x et y de votre type si x.GetHashCode() et y.GetHashCode() sont deux entiers différents, alors les expressions x.Equals(y) et y.Equals(x) doivent valoir toutes les deux false. Si vous ne suivez pas cette contrainte, vous vous exposez à des comportements erronés des tables de hachage. Pour vous aider à satisfaire cette contrainte, le compilateur C#2 émet un avertissement lorsque vous redéfinissez la méthode Equals() sans redéfinir la méthode GetHashCode().

## *Valeurs de hachages obtenues sur les chaînes de caractères*

L'implémentation par défaut de la méthode String.GetHashCode() tient compte de la casse mais ne tient pas compte de la culture. Si vous souhaitez paramétrer la façon dont la culture est prise en compte ou si vous souhaitez ne pas tenir compte de la casse lors de l'obtention d'une valeur de hachage à partir d'une chaîne de caractères, il faut que vous utilisiez une instance d'une classe dérivée de la classe abstraite `System.StringComparer`. Une telle instance est obtenue à partir d'une des propriétés de cette classe telle que StringComparer CurrentCultureIgnoreCase{get;} ou StringComparer Ordinal{get;}. On remarque que cette astuce permet que les classes du *framework* dérivées de la classe StringComparer sont en fait déclarées comme interne.

### *L'interface System.Collections.Generic.IEqualityComparer<T>*

Dans le cas où vous ne pouvez pas (ou ne souhaitez pas) modifier la classe représentant les clés pour introduire votre algorithme de hachage, vous pouvez implémenter l'interface `IEqualityComparer<K>` dans une classe propriétaire spécialement prévue à cet effet. Cette interface est définie comme suit :

```
public interface IEqualityComparer<T> {
   bool  Equals(T x, T y) ;
   int   GetHashCode(T obj) ;
}
```

Certains constructeurs de la classe `Dictionary<K,V>` acceptent cette interface en argument.

### *Conseils pour les algorithmes de hachage*

Il est conseillé que votre algorithme de hachage ait les propriétés suivantes :

- L'algorithme doit avoir une bonne distribution aléatoire afin d'éviter les collisions.
- L'algorithme doit être rapide à exécuter.
- Deux objets ayant le même état doivent avoir la même valeur de hachage.
- Le calcul de la valeur de hachage doit utiliser des champs immuables. Des champs immuables sont des champs initialisés à la construction de l'objet, dont la valeur ne change pas durant la vie de l'objet. Si ce conseil n'est pas respecté, la valeur de hachage d'un même objet risque de ne pas être constante dans le temps.

### *Un exemple*

Voici un exemple ou nous implémentons notre propre algorithme de hachage pour les instances de la classe `Personne`. Notre algorithme multiplie la valeur du code de hachage de la chaîne de caractères du nom de la personne par son année de naissance. Notez que l'on ne craint pas les débordements lors de la multiplication car par défaut, les dépassements de capacité de multiplication d'entiers ne provoquent pas d'exception.

*Exemple 15-15 :*

```
using System.Collections.Generic ;
class Personne {
   public Personne(string nom, int anneeNaissance) {
      m_Nom = nom ;
      m_AnneeNaissance = anneeNaissance ;
   }
   public override int GetHashCode() {
      return m_AnneeNaissance * m_Nom.GetHashCode();
   }
   public override bool Equals(object o) {
      Personne personne = o as Personne;
      if (personne != null)
         return (personne.GetHashCode() == GetHashCode());
      return false;
   }
```

```
   private string m_Nom ;
   private int m_AnneeNaissance ;
}
class Program {
   public static void Main() {
      Dictionary<Personne,string> dico = new
                           Dictionary<Personne,string>() ;
      Personne julien = new Personne( "Julien" , 2002) ;
      Personne mathieu = new Personne( "Mathieu" , 2001) ;
      dico.Add( julien, "20 rue Arson" ) ;
      dico.Add( mathieu, "90 Rue Barberis" ) ;
      bool b = dico.ContainsKey(julien) ;
      // Ici, b vaut true.
   }
}
```

## *Parcours des éléments d'un dictionnaire*

Lorsque nous avons présenté l'interface `IDictionary<K,V>` implémentée par tous les dictionnaires, nous avons vu que cette interface implémente l'interface `IEnumerator<KeyValuePair<K,V>>`. Ainsi, lorsque l'on parcourt les éléments d'un dictionnaire, nous énumérons en fait les couples clés/valeurs et non seulement les clés ou seulement les valeurs. Voici un exemple illustrant le parcours des éléments d'un dictionnaire :

*Exemple 15-16 :*

```
using System.Collections.Generic ;
class Program {
   public static void Main() {
      Dictionary<string,string> dico = new Dictionary<string,string>() ;
      dico.Add("France", "20 Rue Arson") ;
      dico.Add("Francis", "90 Rue Barberis") ;
      foreach (KeyValuePair<string,string> elem in dico)
         System.Console.WriteLine(elem.Key + " : " + elem.Value);
   }
}
```

Cet exemple affiche :

```
France : 20 Rue Arson
Francis : 90 Rue Barberis
```

Grâce à l'utilisation de types génériques, le parcours des éléments des dictionnaires en .NET v2 avec une boucle de type `foreach` n'utilise pas une opération de *unboxing* superflue et pénalisante pour les performances comme c'était le cas en .NET v1.x.

# *Trier les éléments d'une collection*

Vous avez la possibilité de trier les éléments d'une collection de type `System.Array` ou `System.Collections.Generic.List<T>`. Les autres types de collections n'ont pas besoin d'être triés pour les raisons suivantes :

- `System.Collections.BitArray` et `System.Collections.Specialized.BitVector32`
  Il n'y a aucun sens à trier des bits !
- `System.Collections.Generic.Queue<T>` et `System.Collections.Generic.Stack<T>`
  Ces types de collections sont utilisés car l'accès à leurs éléments est très particulier (accès *FIFO* ou accès *LIFO*). Le tri d'une de ces collections modifierait l'accès à ses éléments, et donc sa raison d'être.
- `System.Collections.Generic.LinkedList<T>`
  Si vous avez à trier les éléments d'une liste liée, mieux vaut utiliser la classe `List<T>` pour représenter votre liste.
- `System.Collections.Generic.SortedDictionary<T>`
  Les éléments d'une telle collection sont par définition constamment triés par rapport à leurs clés. Il n'est donc pas nécessaire de prévoir une opération de tri.
- `System.Collections.Generic.Dictionary<K,T>`
  Cette implémentation utilise un algorithme basé sur les valeurs de hachages des clés pour avoir un accès rapide aux éléments. Le tri des éléments perturberait irrémédiablement l'organisation interne nécessaire à la rapidité de ces accès.

Il existe de nombreux algorithmes de tri. L'algorithme retenu par l'implémentation *Microsoft* de .NET est l'algorithme `QuickSort`. Cet algorithme s'exécute en un temps $\mathrm{O}(n\log_2(n))$, si $n$ est le nombre d'éléments de la collection à trier. Cet algorithme est un algorithme de tri instable. Un tri instable ne conserve pas forcément l'ordre de départ des éléments de même valeur.

## *Les interfaces IComparer<T> et IComparable<T>*

Pour pouvoir trier des objets, il faut pouvoir les comparer. Il y a plusieurs manières de comparer les instances d'une classe :

- Soit la classe implémente une des interfaces `System.IComparable` ou `System.IComparable<T>` qui, grâce à la méthode `int CompareTo(T other)` permet d'ordonnancer l'instance courante avec l'instance passée en argument.
- Soit une autre classe implémente l'interface `System.Collections.IComparer` ou `System.Collections.Generic.IComparer<T>` pour ordonnancer les instances de la classe concernée (grâce à la méthode `int Compare(object/T,object/T)`). Dans ce cas, cette autre classe doit être encapsulée dans la classe concernée si la comparaison doit se faire entièrement ou partiellement sur des membres privés. Cette autre classe est en général dérivée de la classe abstraite `System.Collections.Generic.Comparer<T>` qui, bien évidemment, implémente l'interface `IComparer<T>`.
- Signalons enfin que certaines méthodes de comparaison utilisent un délégué de type `System.Comparison<T>` pour comparer les éléments. Cette délégation est définie comme ceci :

```
public delegate int System.Comparison<T>(T x, T y) ;
```

Lorsque vous le pouvez, il est préférable d'utiliser la première manière. Vous êtes cependant obligé d'utiliser une des deux autres manières si vous ne pouvez ou ne souhaitez pas modifier la classe dont les instances doivent être comparées.

Tous les types primitifs du CTS représentant des nombres supportent les interfaces `IComparable` et `IComparable<T>`.

## *Trier les éléments d'un tableau*

La classe `System.Array` présente de nombreuses surcharges de la méthode `Sort()` pour trier des éléments. Les méthodes non présentées ici permettent de ne trier qu'une partie du tableau. Notez que les méthodes de tri d'un tableau sont statiques et qu'il est préférable d'utiliser les surcharges génériques pour bénéficier du typage fort.

- `public static void Sort<K,V>(K[] arrayKeys, V[] arrayValues)`
  Trie `arrayValues` en se servant des éléments de `arrayKeys` comme clés de tri. La classe des éléments de `arrayKeys` doit implémenter l'interface `IComparable<T>` :

  *Exemple 15-17 :*

```
public class Program {
   public static void Main() {
      string[] tabNoms = { "Jean", "Seb", "Eva", "Paul" } ;
      int[] tabKeys = { 3, 1, 6, 2 } ;
      System.Array.Sort<int,string>(tabKeys, tabNoms) ;
      // Ici tabNoms vaut {"Seb","Paul","Jean","Eva"}.
      // Ici tabKeys vaut {1,2,3,6}.
   }
}
```

- `public static void Sort<T>(T[] array)`
  Trie `array`, en supposant que la classe des éléments de `array` implémente l'interface `IComparable<T>`. Voici un exemple de tri sur un tableau d'entiers :

  *Exemple 15-18 :*

```
public class Program {
   public static void Main() {
      int[] tab = { 3, 1, 6, 2 } ;
      System.Array.Sort(tab) ;
      // Ici tab vaut {1,2,3,6}.
   }
}
```

- Voici un exemple de tri sur un tableau avec des éléments d'une classe propriétaire qui implémente l'interface `IComparable` :

  *Exemple 15-19 :*

```
using System ;
class Article : IComparable<Article>{
   public decimal prix ;
   public Article(decimal prix) { this.prix = prix ; }
   int IComparable<Article>.CompareTo(Article other){
      return prix.CompareTo(other.prix) ;
   }
}
public class Program {
   public static void Main() {
      Article[] tab = { new Article(98M) , new Article(19M) ,
```

```
                        new Article(9.5M) } ;
      Array.Sort<Article>(tab) ;
      //  Ici, tab[0].prix vaut 9.5 ; tab[1].prix vaut 19 ;
      //  tab[2].prix vaut 98
   }
}
```

- `public static void Sort<T>(T[] array, IComparer<T> comparer)`
  Trie `array`, en utilisant l'objet `comparer` pour comparer les éléments. Voici l'exemple précédent réécrit avec cette syntaxe de `Sort()` :

  *Exemple 15-20 :*

```
using System.Collections.Generic ;
class Article {
   public class CmpArticle : IComparer<Article>{
      int IComparer<Article>.Compare(Article a1, Article a2){
         return a1.prix.CompareTo(a2.prix);
      }
   }
   public decimal prix ;
   public Article(decimal prix) { this.prix = prix ; }
}
public class Prog{
   public static void Main(){
      Article[] tab = { new Article(98M) , new Article(19M) ,
                        new Article(9.5M) } ;
      System.Array.Sort<Article>( tab , new Article.CmpArticle() ) ;
      //  Ici, tab[0].prix vaut 9.5 ; tab[1].prix vaut 19 ;
      //  tab[2].prix vaut 98
   }
}
```

- `public static void Sort<T>(T[] array, Comparison<T> comparison)`
  Trie `array`, en utilisant la méthode référencée par le délégué `comparison` pour comparer les éléments. Voici l'exemple précédent réécrit avec cette syntaxe de `Sort()` :

  *Exemple 15-21 :*

```
class Article {
   public static int MethodCmp(Article a1, Article a2){
      return a1.prix.CompareTo(a2.prix);
   }
   public decimal prix ;
   public Article(decimal prix) { this.prix = prix ; }
}
public class Program {
   public static void Main(){
      Article[] tab = { new Article(98M) , new Article(19M) ,
                        new Article(9.5M) } ;
      System.Array.Sort<Article>( tab , Article.MethodCmp ) ;
```

```
      //  Ici, tab[0].prix vaut 9.5 ; tab[1].prix vaut 19 ;
      //  tab[2].prix vaut 98
   }
}
```

## *Trier les éléments d'une instance de List<T>*

La classe `System.Collections.Generic.List<T>` présente les méthodes de tri suivantes. Notez que les méthodes de tri d'une liste sont non statiques.

- `void Sort()`
  `void Sort(Comparison<T> comparison)`
  `void Sort(IComparer<T> comparer)`
  `void Sort(int index, int count, IComparer<T> comparer)`
  Trie une partie des éléments d'une liste. Cette partie est contenue entre les éléments `index` et `index+count`.

# *Foncteurs et manipulation des collections*

Il est préférable d'avoir assimilé les notions de méthodes anonymes et d'itérateurs introduites par C#2 avant d'aborder la présente section.

## *Nouvelles délégations spécialisées et foncteurs*

L'espace de noms `System` contient quatre nouvelles délégations particulièrement utiles pour manipuler et obtenir des informations à partir des collections :

```
namespace System {
   public delegate void Action<T> ( T obj ) ;
   public delegate bool Predicate<T> ( T obj ) ;
   public delegate U    Converter<T,U> ( T from ) ;
   public delegate int  Comparison<T> ( T x, T y ) ;
}
```

Dans l'Exemple 15-21 nous avons eu l'occasion de nous servir d'un délégué instance de `Comparison<T>` pour trier les éléments d'un tableau. L'exemple suivant expose quatre différents traitements sur des listes (une requête, un calcul, un tri et une conversion), effectués grâce à des instances de ces délégations :

*Exemple 15-22 :*

```
using System.Collections.Generic ;
class Program {
   class Article {
      public Article(decimal prix,string name){Prix=prix ; Name=name;}
      public readonly decimal Prix ;
      public readonly string  Name ;
   }
   static bool IsEven(int i) { return i % 2 == 0 ; }
```

```
    static int sum = 0 ;
    static void AddToSum(int i) { sum += i ; }
    static int CompareArticle(Article x, Article y){
       return Comparer<decimal>.Default.Compare(x.Prix, y.Prix) ;
    }
    static decimal ConvertArticle(Article article){
       return (decimal)article.Prix ;
    }

    static void Main(){
       // Recherche de tous les entiers pairs.
       // Utilisation implicite d'un délégué de type Predicate<T>.
       List<int> integers = new List<int>() ;
       for(int i=1 ; i<=10 ; i++)
          integers.Add(i) ;
       List<int> even = integers.FindAll( IsEven );

       // Somme les éléments de la liste dans le champ statique sum.
       // Utilisation implicite d'un délégué de type Action<T>.
       integers.ForEach( AddToSum );

       // Tri d'une liste d'éléments d'un type complexe.
       // Utilisation implicite d'un délégué de type Comparison<T>.
       List<Article> articles = new List<Article>() ;
       articles.Add( new Article(5,"Tongues") ) ;
       articles.Add( new Article(3,"Ballon") ) ;
       articles.Sort( CompareArticle );

       // Cast des éléments d'une liste d'éléments d'un type complexe.
       // Utilisation implicite d'un délégué de type Converter<T,U>.
       List<decimal> artPrix =
                 articles.ConvertAll<decimal>( ConvertArticle );
    }
}
```

Les lecteurs qui ont eu l'occasion d'utiliser la *Standard Template Library* (*STL*) de C++ reconnaissent la notion de *foncteur* (*fonctor* en anglais) aussi nommée *fonction-objet*. Un foncteur est un traitement paramétré. Les foncteurs sont particulièrement utiles pour effectuer un même traitement sur chacun des éléments d'une collection. En C++ on surchargeait l'opérateur parenthèse pour implémenter la notion de foncteurs. En .NET, un foncteur prend la forme d'un délégué. En effet, dans le programme précédent, les quatre délégués créés implicitement sont autant d'exemples de foncteurs.

## *Utilisation des méthodes anonymes*

Comme le montre l'exemple suivant, les méthodes anonymes de C# se révèlent être particulièrement adaptées pour implémenter la notion de foncteur. Notez qu'à l'instar du deuxième foncteur qui stocke la somme des éléments d'une liste d'entiers, un foncteur peut encapsuler un état.

*Exemple 15-23 :*

```
using System.Collections.Generic ;
class Program {
   class Article {
      public Article(decimal prix,string name){Prix=prix ; Name=name;}
      public readonly decimal Prix ;
      public readonly string  Name ;
   }
   static void Main(){
      // Recherche de tous les entiers pairs.
      // Utilisation implicite d'un délégué de type Predicate<T>.
      List<int> integers = new List<int>() ;
      for(int i=1 ; i<=10 ; i++)
         integers.Add(i) ;
      List<int> even =integers.FindAll(delegate(int i){return i%2==0;});

      // Somme les éléments de la liste.
      // Utilisation implicite d'un délégué de type Action<T>.
      int sum = 0 ;
      integers.ForEach(delegate(int i) { sum += i ; });

      // Tri d'une liste d'éléments d'un type complexe.
      // Utilisation implicite d'un délégué de type Comparison<T>.
      List<Article> articles = new List<Article>() ;
      articles.Add(new Article(5,"Tongues")) ;
      articles.Add(new Article(3,"Ballon")) ;
      articles.Sort(delegate(Article x, Article y){
         return Comparer<decimal>.Default.Compare(x.Prix,y.Prix) ; });

      // Cast des éléments d'une liste d'éléments d'un type complexe.
      // Utilisation implicite d'un délégué de type Converter<T,U>.
      List<decimal> artPrix = articles.ConvertAll<decimal>(
         delegate(Article article) { return (decimal)article.Prix ; });
   }
}
```

## *Support des classes List<T> et Array*

L'utilisation des foncteurs n'est possible que sur les collections de type `List<T>` et `Array`. En effet, seules ces collections présentent des méthodes qui acceptent des foncteurs pour traiter leurs éléments. Ces méthodes aux noms suffisamment éloquents pour se passer de commentaires, sont listées ci-dessous :

```
public class List<T> : ... {
   public int FindIndex(Predicate<T> match) ;
   public int FindIndex(int index, Predicate<T> match) ;
   public int FindIndex(int index, int count, Predicate<T> match) ;
   public int FindLastIndex(Predicate<T> match) ;
   public int FindLastIndex(int index, Predicate<T> match) ;
```

```
   public int FindLastIndex(int index, int count, Predicate<T> match) ;
   public List<T> FindAll(Predicate<T> match) ;
   public T Find(Predicate<T> match) ;
   public T FindLast(Predicate match) ;
   public bool Exists(Predicate<T> match) ;
   public bool TrueForAll(Predicate<T> match) ;

   public int RemoveAll(Predicate<T> match) ;
   public void ForEach(Action<T> action) ;
   public void Sort(Comparison<T> comparison) ;
   public List<U> ConvertAll<U>(Converter<T,U> converter) ;
   ...
}
public class Array {
   public static int FindIndex<T>(T[] array, int startIndex,
                                  int count, Predicate<T> match) ;
   public static int FindIndex<T>(T[] array, int startIndex,
                                  Predicate<T> match) ;
   public static int FindIndex<T>(T[] array, Predicate<T> match) ;
   public static int FindLastIndex<T>(T[] array, int startIndex,
                                     int count, Predicate<T> match) ;
   public static int FindLastIndex<T>(T[] array, int startIndex,
                                     Predicate<T> match) ;
   public static int FindLastIndex<T>(T[] array, Predicate<T> match) ;
   public static T[] FindAll<T>(T[] array, Predicate<T> match) ;
   public static T Find<T>(T[] array, Predicate<T> match) ;
   public static T FindLast<T>(T[] array, Predicate<T> match) ;
   public static bool Exists<T>(T[] array, Predicate<T> match) ;
   public static bool TrueForAll<T>(T[] array, Predicate<T> match) ;
   public static void ForEach<T>(T[] array, Action<T> action) ;
   public static void Sort<T>(T[] array, System.Comparison<T> comparison) ;
   public static U[] ConvertAll<T, U>( T[] array,
                                      Converter<T, U> converter) ;
   ...
}
```

## *Itérateurs et collections*

Il est aisé d'étendre ce genre de fonctionnalités à tous types de collection grâce aux itérateurs comme le montre le programme suivant (basé sur le *pattern pipeline* vue en page 554) :

*Exemple 15-24 :*

```
using System ;
using System.Collections.Generic ;
class Program{
   static public IEnumerable<T> Filter<T> (
         Predicate<T> predicate, IEnumerable<T> collection) {
      foreach (T item in collection)
```

```
         if (predicate(item))
            yield return item ;
   }
   static public IEnumerable<T> Transform<T> (
         Converter<T,T> transformer, IEnumerable<T> collection) {
      foreach (T item in collection)
         yield return transformer(item) ;
   }
   static public IEnumerable<U> Converter<T, U> (
         Converter<T, U> converter, IEnumerable<T> collection) {
      foreach (T item in collection)
         yield return converter(item) ;
   }
   static public IEnumerable<int> PipelineIntRange(int begin, int end) {
      System.Diagnostics.Debug.Assert( begin < end ) ;
      for ( int i = begin ; i <= end ; i++ )
         yield return i ;
   }
   static void Main(){
      int modulo = 3 ;
      int factor = 2 ;
      foreach (string s in
         Converter<int,string>( delegate(int item) {
            return "Hello:" + item.ToString() ; },
               Filter( delegate(int item) {
                  return (item % modulo == 0) ; },
                     Transform( delegate(int item) {
                        return item * factor ; },
                           PipelineIntRange(1, 10) ) ) ) )
         Console.WriteLine(s) ;
   }
}
```

Ce programme affiche :

```
Hello:6
Hello:12
Hello:18
```

# *Correspondance entre System.Collections.Generic et System.Collections*

Voici un tableau de correspondance entre ces deux espaces de noms. Rappelons que l'espace de noms System.Collections n'est supporté que pour des raisons de compatibilité ascendante avec le *framework* .NET 1.x et qu'il n'y a pas lieu de l'exploiter dans des développements spécifiques avec .NET 2.0 :

| System.Collections.Generics | System.Collections |
|---|---|
| Comparer<T> | Comparer |
| Dictionary<K,T> | HashTable |
| List<T>LinkedList<T> | ArrayList |
| Queue<T> | Queue |
| SortedDictionary<K,T> SortedList<K,T> | SortedList |
| Stack<T> | Stack |
| ICollection<T> | ICollection |
| IComparable<T> | System.IComparable |
| IComparer<T> | IComparer |
| IDictionary<K,T> | IDictionary |
| IEnumerable<T> | IEnumerable |
| IEnumerator<T> | IEnumerator |
| IList<T> | IList |

# 16

# *Bibliothèques de classes*

## *Fonctions mathématiques*

### *La classe System.Math*

La classe `System.Math` ne contient que des champs et des méthodes statiques.

Il y a deux champs statiques : les deux constantes mathématiques e (champ statique « E ») et p (champ statique « PI »). Ces champs sont de type `double`.

Les méthodes statiques représentent l'ensemble des fonctions mathématiques classiques, à savoir les fonctions trigonométriques, les fonctions logarithmiques et puissances, les arrondis etc. Si une valeur d'entrée est hors de l'ensemble, la définition de la fonction mathématique correspondante, la valeur `double.NaN` est retournée mais aucune exception n'est lancée (à part pour la méthode `Round()`). Les comportements de ces fonctions aux limites infinies découlent logiquement des comportements des fonctions mathématiques correspondantes (grâce aux valeurs `double.NegativeInfinity` et `double.PositiveInfinity` qui peuvent être en entrée ou en sortie selon les fonctions).

Voici la liste exhaustive de ces fonctions :

- Les méthodes définies pour plusieurs types numériques :

| | |
|---|---|
| `Type Abs(Type a)` | Cette méthode renvoie la valeur absolue de `a`. `Type` peut être n'importe quel type numérique signé, réel ou entier. |
| `Type Min(Type a, Type b)`<br>`Type Max(Type a, Type b)` | Ces méthodes renvoient le minimum (respectivement le maximum) de `a` et de `b`. `Type` peut être n'importe quel type numérique signé ou non, réel ou entier. |

| | |
|---|---|
| `int Sign(Type a)` | Cette méthode renvoie le signe de a, sous la forme d'un entier : −1 si a est négatif ; 0 si a est égal à 0 ; 1 si a est positifType peut être n'importe quel type numérique signé, réel ou entier. |

- Les méthodes trigonométriques. Tous les angles sont précisés en radians. Si un argument est hors de l'ensemble de définition d'une telle fonction, la valeur `double.NaN` est retournée.

| | |
|---|---|
| `double Cos(double d)` | Retourne le cosinus de d. L'ensemble de définition est tous les nombres réels et l'ensemble image est [ -1 ; 1 ]. |
| `double Sin(double d)` | Retourne le sinus de d. L'ensemble de définition est tous les nombres réels et l'ensemble image est [ -1 ; 1 ]. |
| `double Tan(double d)` | Retourne la tangente de d. L'ensemble de définition est tous les nombres réels (contrairement à la tangente mathématique qui n'est pas définie pour un ensemble de points discrets) et l'ensemble image est tous les nombres réels. |
| `double Acos(double d)` | Retourne l'arc cosinus de d. L'ensemble de définition est [ -1 ; 1 ] et l'ensemble image est [ 0 ; p ]. |
| `double Asin(double d)` | Retourne l'arc sinus de d. L'ensemble de définition est [ -1 ; 1 ] et l'ensemble image est [ -p/2 ; p/2 ]. |
| `double Atan(double d)` | Retourne l'arc tangente de d. L'ensemble de définition est tous les nombres réels et l'ensemble image est [ -p/2 ; p/2 ]. p/2 est atteint pour la valeur `double.PositiveInfinity` alors que -p/2 est atteint pour la valeur `double.NegativeInfinity`. |
| `double Atan2(double y,double x)` | Retourne l'angle entre la demi-droite positive des abscisses et le vecteur qui part de l'origine jusqu'au point (x,y). L'ensemble image est donc [ -p ; p ]. |
| `double Cosh(double d)` | Retourne le cosinus hyperbolique de d. L'ensemble de définition est tous les nombres réels et l'ensemble image est [ 1 ; `double.PositiveInfinity`]. `double.PositiveInfinity` est atteint pour les valeurs `double.NegativeInfinity` et `double.PositiveInfinity`. |
| `double Sinh(double d)` | Retourne le sinus hyperbolique de d. L'ensemble de définition est tous les nombres réels et l'ensemble image est [`double.NegativeInfinity` ; `double.PositiveInfinity`]. `double.PositiveInfinity` est atteint pour la valeur `double.PositiveInfinity` et `double.NegativeInfinity` est atteint pour la valeur `double.NegativeInfinity`. |

| | |
|---|---|
| `double Tanh(double d)` | Retourne la tangente hyperbolique de `d`. L'ensemble de définition est tous les nombres réels et l'ensemble image est [ -1 ; 1 ]. 1 est atteint pour la valeur `double.PositiveInfinity` et `-1` est atteint pour la valeur `double.NegativeInfinity`. |

- Les méthodes puissances, exponentielles et logarithmiques :

| | |
|---|---|
| `double Sqrt(double d)` | Retourne la racine carrée de `d`. L'ensemble de définition est tous les nombres réels positifs ou nul. Si une valeur négative est passée, la valeur `double.Nan` est retournée. |
| `double Pow(double x,double y)` | Retourne `x` puissance `y`. |
| `double Exp(double d)` | Retourne l'exponentielle du nombre `d` (i.e le nombre e puissance `d`). |
| `double Log(double d)` | Retourne le logarithme népérien du nombre `d`. Si une valeur négative est passée, la valeur `double.Nan` est retournée. |
| `double Log(double d,double b)` | Retourne le logarithme en base `b` du nombre `d`. Si une valeur négative est passée pour `d` ou `b`, la valeur `double.Nan` est retournée. |
| `double Log10(double d)` | Retourne le logarithme en base décimale du nombre `d`. Si une valeur négative est passée, la valeur `double.Nan` est retournée. |

- `Type Round(Type d)`
  `Type Round(Type d, int n)`
  `Type` est un type réel. La méthode `Round()` permet d'arrondir un nombre réel à la précision `n`. Par exemple :

```
Math.Round(3.1415)   // retourne 3.0
Math.Round(3.1415,2) // retourne 3.14
```

Si n est négatif l'exception `System.ArgumentOutOfRangeException` est lancée.

- `double Ceiling(double d)`
  `double Floor(double d)`
  *Ceiling* veut dire plafond en anglais, et *floor* veut dire sol. La méthode `Floor()` renvoie donc la partie entière de `d`, soit le plus grand entier inférieur à `d`. De même, la méthode `Ceiling()` renvoie la partie entière de `d` +1, soit le plus petit entier supérieur à `d`.

- `double` **`IEEERemainder`**`(double x,double y)`
  Retourne le nombre `x-(yQ)` ou `Q` est égal au quotient de `x` par `y` arrondi à l'entier le plus proche. Si ce quotient est pile entre deux nombres entiers, la valeur entière paire est choisie. Si `y` est nul, `double.NaN` est retournée.

De grands domaines mathématiques comme le calcul matriciel ou les nombres complexes ne sont pas nativement couverts par le *framework* .NET. Gageons que des librairies vont être créées. Notons que Fortran.NET sera sûrement le langage de choix pour les applications calculatoires.

## *La classe System.Random*

L'utilisation de la classe `System.Random` permet de générer des nombres aléatoires. Les méthodes de cette classe ne sont pas statiques. Autrement dit, cette classe doit être instanciée pour être utilisée. Les méthodes de cette classe sont :

| | |
|---|---|
| `Random()` | Constructeur d'un objet `Random`. La valeur de base de l'algorithme de génération aléatoire (aussi appelée graine ou *seed* en anglais) est calculée à partir de l'heure. |
| `Random(int seed)` | Constructeur d'un objet `Random`. La valeur de base de l'algorithme de génération aléatoire est seed. |
| `int Next()` | Renvoie un nombre entier aléatoirement choisi dans l'ensemble 0 et `int.MaxValue`. |
| `int Next(int a)` | Renvoie un nombre entier aléatoirement choisi dans l'ensemble 0 et `a-1`. |
| `int Next(int a,int b)` | Renvoie un nombre entier aléatoirement choisi dans l'ensemble `a` et `b-1`. |
| `double NextDouble()` | Renvoie un nombre réel aléatoirement choisi dans l'ensemble [ 0 ; 1.0 [. |
| `void NextBytes(Byte[] Tab)` | Rempli le tableau d'octets avec pour chaque octet un nombre entier aléatoirement choisi dans l'ensemble 0 et 255. |

# *Données temporelles (dates, durées...)*

Avec le *framework* .NET, les développeurs *Microsoft* disposent enfin d'une gestion des données temporelles cohérente. Jusqu'ici, dans le monde *Microsoft*, il fallait se contenter de plusieurs classes qui représentaient les dates sur quatre octets (pour les classes `CTime` ou `time_t`) ou huit octets (pour la classe `COleDateTime`). Les fonctionnalités de ces classes étaient redondantes avec des méthodes avec des noms et des arguments différents.

En .NET les données temporelles sont principalement manipulées à l'aide de deux structures :

- La structure `System.DateTime` dont les instances représentent une date.

- La structure `System.TimeSpan` dont les instances représentent un intervalle de temps. Par exemple il existe une opération qui permet de récupérer une instance de `System.TimeSpan` à partir de deux instances de `System.DateTime`.

## *La structure System.DateTime*

Les dates représentées par les instances de `System.DateTime` sont dans l'intervalle de temps compris entre les deux dates suivantes :

- minuit (début de la journée) 1er janvier 0001 et
- minuit (fin de la journée) du 31 décembre 9999.

Dans une instance de `System.DateTime`, en interne, la date est représentée par un entier de type `long` (un entier signé sur huit octets). La correspondance est d'une unité pour un intervalle de temps de 100 nanosecondes. Cet entier est accessible en lecture/écriture avec la propriété non statique `Ticks`. Cet entier est compris entre les deux champs statiques constants `MinValue` et `MaxValue` de type `DateTime`. Les deux entiers correspondants aux dates extrémales citées plus haut sont 0 et 3.155.378.975.999.999.999.

Dans tout l'ouvrage nous ne considérons que le calendrier grégorien, utilisé dans le monde occidental. Sachez que l'espace de noms `System.Globalization` contient en plus de la classe `GregorianCalendar`, d'autres classes qui représentent d'autres calendriers comme `HebrewCalendar`, `JapaneseCalendar`, `JulianCalendar` etc.

La classe `DateTime` comprend de nombreux membres, statiques ou non. La liste exhaustive de ces membres se trouve à l'article **DateTime Members** des **MSDN**. Néanmoins, nous vous présentons quelques-uns de ces membres les plus couramment utilisés :

- Les constructeurs :

  ```
  DateTime(int année, int mois, int jour)
  DateTime(int année, int mois, int jour, int heure, int minute, int seconde)
  DateTime(int année, int mois, int jours, int heure, int minute, int seconde,
  int millisec)
  DateTime(long Ticks)
  ```

  `année` est entre 1 et 9999. `mois` est entre 1 et 12. `jour` est entre 1 et 28, 29, 30 ou 31 (en fonction du mois). `heure` est entre 0 et 23. `minute` est entre 0 et 59. `seconde` est entre 0 et 59. `millisec` est entre 0 et 999.
- Les représentations du temps courant (très utiles, notamment pour les tests de performances) :

| | |
|---|---|
| `static DateTime Now` | Cette propriété statique accessible en lecture seulement retourne une instance de `DateTime` qui représente la date et l'heure courante. La précision est de l'ordre du centième de seconde. |
| `static DateTime Today` | Cette propriété statique, accessible en lecture seulement, retourne une instance de `DateTime` qui représente la date courante avec l'heure initialisée à minuit au début de la journée. |

| | |
|---|---|
| `static DateTime UtcNow` | Cette propriété statique, accessible en lecture seulement, retourne une instance de `DateTime` qui représente la date et l'heure courante représentée en temps universel. |

- Les extractions d'information en unités temporelles familières d'une instance de `DateTime` (membres non statiques). Sauf remarque contraire, ces propriétés sont accessibles en lecture/écriture :

| | |
|---|---|
| `DateTime Date` | Renvoie la date avec la partie temps initialisée à minuit au début de la journée. Cette propriété est accessible en lecture seulement. |
| `int Day` | Compris entre 1 et 31. |
| `DayOfWeek DayOfWeek` | Renvoie une des valeurs de l'énumération `System.DayOfWeek` qui sont : `Sunday Monday Tuesday Wednesday Thursday Friday Saturday` |
| `int DayOfYear` | Compris entre 1 et 366. |
| `int Hour` | Compris entre 0 et 23. |
| `int Millisecond` | Compris entre 0 et 999. |
| `int Minute` | Compris entre 0 et 59. |
| `int Month` | Compris entre 1 et 12. |
| `int Second` | Compris entre 0 et 59. |
| `TimeSpan TimeOfDay` | Durée depuis minuit. |
| `int Year` | Compris entre 1 et 9999. |

- Les opérations sur les dates :

| | |
|---|---|
| `DateTime Add(TimeSpan dT)operator+` | Ajoute la durée `dT` à la date. L'opérateur + est en général préféré. |
| `TimeSpan Subtract(DateTime date)operator-` | Soustrait `date` à la date représentée par l'instance. L'opérateur — est en général préféré. Remarquez que la durée retournée peut être négative. |
| `static int Compare(DateTime d1,DateTime d2)operator< <= > >= ==` | Renvoie 0 si `d1` égal à `d2`.Renvoie 1 si `d1` antérieur à `d2`.Renvoie -1 si `d1` postérieur à `d2`.L'utilisation des opérateurs de comparaison est en général préférée. |
| `static bool IsLeapYear(int année)` | Renvoie `true` si l'année spécifiée est bissextile. |

| | |
|---|---|
| `static int DayInMonths(int année,int mois)` | Renvoie le nombre de jour pour le mois spécifié. Par exemple `DayInMonths(2000,2)` renvoie 29 car l'année 2000 est bissextile. |

- `string ToString(string Format)`
  Renvoie une date dans une chaîne de caractères. La chaîne `Format` spécifie la façon dont la présentation de la date est formatée. Dans la suite nous supposons l'affichage du programme suivant (avec la chaîne XXX variable) :

```
// le 15 octobre 2002 à 14 heures 30 minutes 0 seconde
DateTime d = new DateTime(2002,10,15,14,30,0) ;
Console.WriteLine( d.ToString(XXX) ) ;
```

| **Indicateurs pour la représentation générale d'une date** | |
|---|---|
| XXX | Affichage |
| "d" | 15/10/2002 |
| "D" | mardi 15 octobre 2002 |
| "f" | mardi 15 octobre 2002 14:30 |
| "F" | mardi 15 octobre 2002 14:30:00 |
| "g" | 15/10/2002 14:30 |
| "G" | 15/10/2002 14:30:00 |
| "M" | 15 octobre |
| "R" | Tue, 15 Oct 2002 14:30:00 GMT |
| "s" | Dates formatées pour être triées simplement. Par exemple :2002-10-15T14:30:00 |
| "t" | 14:30 |
| "T" | 14:30:00 |
| "u" | 2002-10-15 14:30:00Z |
| "U" | mardi 15 octobre 2002 12:30:00 Temps universel : à cette date la France a deux heures d'avance sur le temps universel. |
| "y" | octobre 2002 |

| **Indicateurs pour la représentation personnalisée d'une date** | |
|---|---|
| XXX | Affichage |

| | |
|---|---|
| : | Séparateur des heures:minutes:secondes. Ce séparateur dépend de la culture choisie dans Windows. |
| / | Séparateur des années/mois/jours. Ce séparateur dépend de la culture choisie dans Windows. |
| y | Année, représentée par un numéro de 0 à 99. |
| yy | Année, représentée par un numéro de 00 à 99. |
| yyy | Année, représentée par un numéro de 0000 à 9999. |
| M | Mois, représenté par un numéro de 1 à 12. |
| MM | Mois, représenté par un numéro de 01 à 12. |
| MMM | Mois, représenté par une abréviation de trois lettres dans la langue d'installation de Windows (Par exemple `oct.`). |
| MMMM | Mois, dans la langue d'installation de Windows |
| d | Jour, représenté par un numéro de 1 à 31. |
| dd | Jour, représenté par un numéro de 01 à 31. |
| ddd | Jour, représenté par une abréviation de trois lettres dans la langue d'installation de Windows (Par exemple `dim.`). |
| dddd | Jour, dans la langue d'installation de Windows |
| h | Heure, représentée par un numéro de 0 à 11. |
| hh | Heure, représentée par un numéro de 00 à 11. |
| H | Heure, représentée par un numéro de 0 à 23. |
| HH | Heure, représentée par un numéro de 00 à 23. |
| m | Minute, représentée par un numéro de 0 à 59. |
| mm | Minute, représentée par un numéro de 00 à 59. |
| s | Seconde, représentée par un numéro de 0 à 59. |
| ss | Seconde, représentée par un numéro de 00 à 59. |

Certains indicateurs comme y ou d ont donc un double sens. Si c'est le seul caractère de la chaîne, le *framework* choisira la représentation générale de la date.

| **Exemples de représentations personnalisées de date utilisant les indicateurs présentés dans le tableau ci-dessus** | |
|---|---|
| XXX | Affichage |
| "y/M/d h:m:s" | 2/10/15 2:30:0 |
| "yy/MM/dd HH:mm:ss" | 02/10/15 14:30:00 |
| "ddd d MMM yyy" | mar. 15 oct. 2002 |
| "le dddd d MMMM yyyy" | le mardi 15 octobre 2002 |
| "m minutes s secondes" | 30 30inue0 0 0econ15e0 |
| "m 'minutes' s 'secondes'" | 30 minutes 0 secondes |

- `static DateTime Parse(string s)`
  Fabrique une date à partir d'une chaîne de caractères. Si la date est non valide l'exception `FormatException` est lancée. Si la chaîne est vide l'exception `ArgumentException` est lancée. Par défaut la représentation de la chaîne de caractères doit être celle de la culture d'installation de *Windows*. Par exemple si votre version de *Windows* est française, vous pouvez écrire :

  *Exemple 16-1 :*

```
...
      string s = "15/10/2002 14:30:0" ;
      DateTime d = DateTime.Parse(s) ;
      Console.WriteLine( d.ToString("G")) ;
...
```

  Qui affiche : `15/10/2002 14:30:00`
  Si la date est représentée dans une autre culture vous pouvez écrire :

  *Exemple 16-2 :*

```
...
      string s = "10/15/2002 14:30:0";
      DateTime d=DateTime.Parse(s,
                     new System.Globalization.CultureInfo("en-US") ) ;
      Console.WriteLine( d.ToString("G")) ;
...
```

  Qui affiche : `15/10/2002 14:30:00`
  Vous pouvez aussi vous servir du nom des mois (en abrégé ou non) par exemple :

  *Exemple 16-3 :*

```
...
      string s = "15 octobre 2002 14:30:0" ;
      DateTime d = DateTime.Parse(s) ;
      Console.WriteLine( d.ToString("G")) ;
...
```

Affiche : 15/10/2002 14:30:00

*Exemple 16-4 :*

```
...
      string s = "15 oct. 2002 14:30:0" ;
      DateTime d = DateTime.Parse(s) ;
      Console.WriteLine( d.ToString("G")) ;
...
```

Affiche : 15/10/2002 14:30:00

*Exemple 16-5 :*

```
...
      string s = "15 october 2002 14:30:0" ;
      DateTime d=DateTime.Parse(s,
                     new System.Globalization.CultureInfo("en-US")) ;
      Console.WriteLine( d.ToString("G")) ;
...
```

Affiche : 15/10/2002 14:30:00

## *La structure System.TimeSpan*

Les instances de la structure `System.TimeSpan` représentent une durée. On montre dans la section précédente que de telles instances peuvent être obtenues à partir d'opérations sur des dates (par exemple une soustraction d'une date à une autre pour savoir quelle est la durée entre deux dates).

La représentation interne de la durée dans une instance de `TimeSpan` est la même que pour une instance de `DateTime`. La seule différence est qu'une durée peut être négative.

La classe `TimeSpan` comprend de nombreux membres, statiques ou non. La liste exhaustive de ces membres se trouve à l'article **TimeSpan Members** des **MSDN**. Néanmoins nous vous présentons quelques-uns de ces membres les plus couramment utilisés :

- Les constructeurs :
  `TimeSpan(int jours, int heures, int minutes, int secondes)`
  `TimeSpan(int heures, int minutes, int secondes)`
  `TimeSpan (long Ticks)`
  Tous les arguments peuvent prendre une valeur positive ou négative quelconque.
- Les extractions d'information en unités temporelles familières d'une instance de `TimeSpan` (membres non statiques). Ces propriétés sont accessibles en lecture écriture :

| | |
|---|---|
| `int Days` | Nombre de jours pleins contenus dans la durée. Si la durée est négative, c'est le nombre de « jours négatifs » plein contenus dans la durée. |
| `double TotalDays` | Nombre de jours contenus dans la durée. La partie fractionnaire correspond à la fraction de jour non plein. |

| | |
|---|---|
| `int Hours` | Nombre d'heures pleines contenues dans la durée purgée de tous les jours pleins. La valeur retournée est donc entre 0 et 23 si la durée est positive et entre 0 et -23 si la durée est négative. |
| `double TotalHours` | Nombre d'heures contenues dans la durée. La partie fractionnaire correspond à la fraction de l'heure non pleine. |
| `int Minutes` | Nombre de minutes pleines contenues dans la durée purgée de toutes les heures pleines. La valeur retournée est donc entre 0 et 59 si la durée est positive et entre 0 et -59 si la durée est négative. |
| `double TotalMinutes` | Nombre de minutes contenues dans la durée. La partie fractionnaire correspond à la fraction de la minute non pleine. |
| `int Seconds` | Nombre de secondes pleines contenues dans la durée purgée de toutes les minutes pleines. La valeur retournée est donc entre 0 et 59 si la durée est positive et entre 0 et -59 si la durée est négative. |
| `double TotalSeconds` | Nombre de secondes contenues dans la durée. La partie fractionnaire correspond à la fraction de la seconde non pleine. |
| `int Milliseconds` | Nombre de millisecondes pleines contenues dans la durée purgée de toutes les secondes pleines. La valeur retournée est donc entre 0 et 999 si la durée est positive et entre 0 et -999 si la durée est négative. |
| `double TotalMilliseconds` | Nombre de millisecondes contenues dans la durée. La partie fractionnaire correspond à la fraction de la milliseconde non pleine. |
| `long Ticks` | Nombre d'intervalles de 100 nanosecondes contenus dans la durée. C'est donc la représentation interne de la durée. |

- `TimeSpan Duration()`
  Renvoie la durée en valeur absolue d'un intervalle de temps.
- Les opérateurs + - != == < <= > >= sont redéfinis et permettent de manipuler simplement et logiquement les instances de `TimeSpan`.

# *Volumes, répertoires et fichiers*

Les volumes, les répertoires et les fichiers font partie intégrante du système d'exploitation. Le *framework* .NET offre un ensemble de classes permettant de les manipuler d'une manière indépendante du système d'exploitation sous-jacent. Ces classes sont toutes dans l'espace de noms `System.IO`,

La gestion des droits d'accès aux répertoires et aux fichiers fait partie du cadre plus général de la gestion des droits d'accès, décrit dans le chapitre sur la sécurité.

## *Manipulation de volumes*

Le *framework* .NET présente la classe `System.IO.DriveInfo` qui permet de représenter un *volume* (*drive* en anglais). Cette classe présente la méthode statique `DriveInfo[] GetDrives()` qui

permet de récupérer tous les volumes disponibles de la machine. Lorsque vous disposez d'une instance de `DriveInfo`, les propriétés suivantes permettent d'obtenir des informations relatives au volume sous-jacent :

| Propriétés de `DriveInfo` | Fonctionalité |
|---|---|
| `long AvailableFreeSpace{get;}` | Retourne le nombre d'octets libres sur le volume. Prend en compte les quotas. |
| `long TotalFreeSpace{get;}` | Retourne le nombre d'octets libres sur le volume. Ne prend pas en compte les quotas. |
| `string DriveFormat{get;}` | Retourne le nom du système de fichier utilisé (`NTFS`, `FAT32` etc). |
| `DriveType DriveType{get;}` | Retourne une valeur de l'énumération `DriveType` décrivant le type de volume (`CDRom`, `Fixed`, `Removable` etc). |
| `bool IsReady{get;}` | Détermine si le volume est prêt. |
| `string Name{get;}` | Retourne le nom du volume. |
| `directoryInfo RootDirectory{get;}` | Retourne le répertoire racine du volume. |
| `long TotalSize{get;}` | Retourne la taille en octets du volume. |
| `string VolumeLabel{get;set;}` | Retourne ou positionne le label (i.e le nom) du volume. |

Cette classe admet aussi un constructeur acceptant une chaîne de caractères précisant le label du volume que l'on souhaite représenter.

## *Manipulation de répertoires*

Le *framework* .NET présente les deux classes `System.IO.Directory` et `System.IO.DirectoryInfo` qui permettent de manipuler les *répertoires* (*directory* en anglais) d'un système d'exploitation. La plupart des fonctionnalités de ces deux classes sont identiques. En fait, elles présentent chacune une façon de travailler avec les répertoires :

- `Directory` ne contient que des membres statiques. Elle permet de travailler avec les répertoires sans avoir à instancier d'objet.
- `DirectoryInfo` ne contient que des membres non statiques. Pour travailler avec la classe `DirectoryInfo`, il faut l'instancier. Chaque instance de `DirectoryInfo` représente un répertoire. La classe `DirectoryInfo` est dérivée de la classe `System.IO.FileSystemInfo` qui représente un fichier au sens large, c'est-à-dire un fichier ou un répertoire.

Ces classes sont très complètes. Voici quelques-unes unes des fonctionnalités qu'elles présentent. Pour la liste exhaustive des membres de ces classes, référez-vous aux **MSDN** :

| **Membres de `Directory` ou `DirectoryInfo`** | **Fonctionalité** |
|---|---|
| `NameAttributesLastAccessTimeLastWriteTime-CreationTime` | Obtention et modification de toutes les informations relatives à un répertoire comme son nom (`Name`), ses attributs (`Attributes`), la date du dernier accès (`LastAccessTime`), la date de la dernière modification (`LastWriteTime`) ou la date de création (`CreationTime`). |
| `string Directory.GetCurrentDirectory()void Directory.SetCurrentDirectory(string)` | Obtention et modification du répertoire d'exécution de l'application. |
| `DirectoryInfo Direc-tory.CreateDirectory(string)DirectoryInfo Directory.CreateSubdirectory(string)` | Création de nouveaux répertoires. |
| `void Directory.Delete(string[,bool bRe-cursive])void DirectoryInfo.Delete([bool bRecursive])` | Destruction d'un répertoire et de son contenu. |
| `string[] Directory.GetDirectories(string) DirectoryInfo[] Directo-ryInfo.GetDirectories()` | Obtention des sous répertoires d'un répertoire. |
| `string[] Directory.GetFiles(string) FileInfo[] DirectoryInfo.GetFiles()` | Obtention des fichiers contenus dans un répertoire. |
| `DirectoryInfo Direc-tory.GetParent(string)DirectoryInfo Di-rectoryInfo. Parent` | Obtention du répertoire parent d'un répertoire. |
| `bool Directory.Exists(string)bool Directo-ryInfo.Exists` | Test de l'existence d'un répertoire. |
| `void Directory.Move(string src, string dest)void DirectoryInfo.MoveTo(string src)` | Déplacement d'un répertoire et de son contenu. |
| `DirectoryInfo.Refresh()` | Rafraîchissement des informations d'une instance de `DirectoryInfo`. |

Voici un petit exemple permettant de lister l'arborescence des sous répertoires du répertoire courant de l'application. Notez l'appel récursif de la méthode `DisplayDirectory()`, et la manière utilisée pour construire la chaîne de caractères d'indentation :

*Exemple 16-6 :*

```
using System ;
using System.IO ;
class Program {
   static void DisplayDirectory(DirectoryInfo dir, string sIndent) {
      Console.WriteLine(sIndent + dir.Name) ;
      foreach (DirectoryInfo sousdir in dir.GetDirectories())
         DisplayDirectory(sousdir, sIndent + " ") ;
   }
   static void Main() {
      DirectoryInfo dir = new
            DirectoryInfo( Directory.GetCurrentDirectory() );
      DisplayDirectory(dir, string.Empty) ;
   }
}
```

## *Manipulation de fichiers*

Le *framework* .NET présente les deux classes `System.IO.File` et `System.IO.FileInfo` qui permettent de manipuler les *fichiers* (*files* en anglais) d'un système d'exploitation. La plupart des fonctionnalités de ces deux classes sont identiques. En fait, elles présentent chacune une façon de travailler avec les fichiers :

- `File` ne contient que des membres statiques. Elle permet de travailler avec les fichiers sans avoir à instancier d'objet.
- `FileInfo` ne contient que des membres non statiques. Pour travailler avec `FileInfo`, il faut l'instancier. Chaque instance de `FileInfo` représente un fichier. `FileInfo` est dérivée de la classe `System.IO.FileSystemInfo` qui représente un fichier au sens large, c'est-à-dire un fichier ou un répertoire.

Ces classes sont très complètes. Voici quelques-unes des fonctionnalités qu'elles présentent. Pour la liste exhaustive des membres des classes `File` et `FileInfo`, référez-vous aux **MSDN** :

Les opérations d'accès et de modification du contenu d'un fichier sont présentées page 636. En effet, l'accès et la modification d'un fichier font partie du cadre plus général de la gestion d'un flot de données à partir d'une source ou vers une destination.

| **Membres de `File` ou `FileInfo`** | **Fonctionnalité** |
|---|---|
| `NameAttributesLastWriteTimeCreationTime` | Obtention et modification de toutes les informations relatives à un fichier, comme son nom (`Name`), ses attributs (`Attributes`), la date du dernier accès (`LastAccessTime`), la date de la dernière modification (`LastWriteTime`) ou la date de création (`CreationTime`). |

| | |
|---|---|
| `DirectoryInfo FileInfo. Directory` | Obtention du répertoire d'un fichier avec la propriété accessible en lecture seule `DirectoryInfo Directory` de la classe `FileInfo`. |
| `bool File.Exists(string)bool FileInfo.Exists` | Test de l'existence d'un fichier. |
| `string FileInfo.Extension` | Obtention de l'extension d'un fichier. |
| `long FileInfo.Length` | Obtention de la taille d'un fichier (en octets). |
| `FileStream File.Create(string) FileStream File.Create()` | Création d'un fichier. |
| `void File.Move(string src, string dest)void FileInfo.MoveTo(string)` | Déplacement d'un fichier. |
| `void File.Copy(string src, string dest,[bool bOverwrire])void FileInfo.CopyTo(string,[bool bOverwrire])` | Copie d'un fichier. |
| `void File.Delete(string)void FileInfo.Delete()` | Destruction d'un fichier. |

## *Manipulation de chemins (Path)*

Le *framework* .NET présente la classe `System.IO.Path` qui permet de manipuler les *chemins* vers les fichiers et répertoires, d'une manière indépendante du système d'exploitation sous-jacent. En effet, le formatage d'une chaîne de caractères représentant un chemin dépend du système d'exploitation sous-jacent. Par exemple certains systèmes d'exploitation ne représentent pas un volume par une lettre tandis que d'autres n'acceptent pas plus de trois caractères pour l'extension d'un fichier.

Les chemins sont représentés par des chaînes de caractères. Tous les membres de la classe `Path` sont statiques. La classe permet de manipuler des *chemins relatifs* et des *chemins absolus*.

Faites attention, en C# le caractère « \ » de séparation dans un chemin doit être écrit « \\ » dans une chaîne de caractères non *verbatim*. Notez que la propriété accessible en lecture seule `char PathSeparator` de la classe `Path`, représente le caractère de séparation utilisé par le système d'exploitation sous-jacent.

La classe `Path` présente d'autres propriétés intéressantes comme la propriété `char[] InvalidPathChars` qui contient tous les caractères non utilisables dans une chaîne de caractères décrivant un chemin dans le système d'exploitation sous-jacent. Pour la liste exhaustive des membres de `Path`, référez-vous aux MSDN.

## *Gérer les événements survenants sur un fichier ou un répertoire*

La classe `System.IO.FileSystemWatcher` permet d'intercepter les événements qui surviennent sur un fichier, sur un répertoire ou sur un répertoire et l'arborescence des sous répertoires. Les fichiers et répertoires à surveiller peuvent être locaux ou distants. Cependant les fichiers et répertoires en lecture seule comme ceux des CDs et des DVDs ne peuvent être surveillés.

Les événements de la classe `FileSystemWatcher` sont les suivants. Notez que pour que les événements soient effectivement interceptés, il faut positionner la propriété `EnableRaisingEvents` à `true` :

- `Changed` : cet événement est déclenché lorsqu'un changement quelconque intervient sur un des objets (fichiers ou répertoires) surveillés.
- `Created` : cet événement est déclenché lorsqu'un objet (fichier ou répertoire) est ajouté dans le répertoire (ou un des sous répertoires) surveillés.
- `Deleted` : cet événement est déclenché lorsqu'un objet (fichier ou répertoire) surveillé est détruit.
- `Error` : en interne, la classe `FileSystemWatcher` contient un tampon pour stocker les événements interceptés mais dont la procédure de rappel n'a pas encore été appelée. Si les limites de ce tampon sont atteintes, l'événement `Error` est déclenché. Vous pouvez accéder et modifier la taille du tampon avec la propriété `InternalBufferSize`.
- `Renamed` : cet événement est déclenché lorsqu'un objet (fichier ou répertoire) surveillé est renommé.

Vous avez la possibilité d'affiner le type des événements que vous souhaitez intercepter avec les techniques suivantes :

- En positionnant la propriété `IncludeSubdirectories` pour indiquer si vous souhaitez surveiller aussi les sous répertoires.
- En modifiant la propriété `string Filter` de la classe `FileSystemWatcher` vous pouvez imposer une contrainte sur les fichiers à surveiller. Par défaut cette propriété vaut `"*"`. Par exemple vous pouvez décider de ne surveiller que les fichiers textes en affectant `"*.txt"` à cette propriété.
- Lorsque vous utilisez l'événement `Changed`, vous interceptez l'événement « le contenu du fichier est changé ». La propriété `NotifyFilter` de la classe `FileSystemWatcher` permet de choisir d'intercepter un ou plusieurs autres types de changements. Cette propriété prend ses valeurs dans l'énumération indicateur binaire `System.IO.NotifyFilters` :

| **Valeur de l'énumération `System.IO.NotifyFilters`** | **Description** |
|---|---|
| `Attributs` | Les attributs d'un objet (fichier ou répertoire) à surveiller ont changé. |
| `CreationTime` | La date de création d'un objet (fichier ou répertoire) à surveiller a changé. |
| `DirectoryName` | Le nom d'un répertoire à surveiller a changé. |

| | |
|---|---|
| `FileName` | Le nom d'un fichier à surveiller a changé. |
| `LastAccess` | La date de dernier accès d'un objet (fichier ou répertoire) à surveiller a changé. |
| `LastWrite` | La date de dernier accès en écriture d'un objet (fichier ou répertoire) à surveiller a changé. |
| `Security` | Les attributs de sécurité d'un objet (fichier ou répertoire) à surveiller ont changé. |
| `Size` | La taille d'un objet (fichier ou répertoire) à surveiller a changé. |

Pour la liste exhaustive des membres de `FileSystemWatcher` veuillez vous référer aux **MSDN**.

Le programme suivant surveille les changements de contenu, les changements de nom de sous répertoire et les accès à tous les fichiers texte situés dans le répertoire courant de l'application ou dans un de ses sous répertoires :

*Exemple 16-7 :*

```
using System ;
using System.IO ;
public class Program {
   public static void Main() {
      FileSystemWatcher watcher = new FileSystemWatcher() ;
      watcher.Path = Directory.GetCurrentDirectory() ;
      watcher.NotifyFilter = NotifyFilters.LastAccess |
               NotifyFilters.DirectoryName ;
      watcher.Filter = "*.txt" ;
      watcher.IncludeSubdirectories = true ;
      watcher.Changed += new FileSystemEventHandler(OnChange) ;
      watcher.EnableRaisingEvents = true ;
      Console.WriteLine("Pressez \'q\' pour stopper l'application.") ;
      while (Console.Read() != 'q') ;
   }
   public static void OnChange(object source, FileSystemEventArgs e) {
      Console.WriteLine("Fichier : " + e.FullPath
             + " Changement:" + e.ChangeType) ;
   }
}
```

# *Base des registres*

## *Introduction*

La *base des registres* (*registry* en anglais) appelée aussi *registre,* est une base de données qui renferme la plupart des informations consommées par le système d'exploitation. Ces informations

sont très hétéroclites. Cela va du type de clavier utilisé jusqu'aux préférences utilisées pour les logiciels installés sur la machine. Le registre a été introduit sous Windows NT 3.x. Avant, on utilisait des fichiers textes de type `.ini`, ce qui avait pour principal inconvénient de perturber l'organisation du système. La base des registres vise à éliminer ces inconvénients. Vous pouvez visualiser et modifier les informations contenues dans la base des registres avec un des utilitaires `regedit.exe` ou `regedt32.exe`. Ces éditeurs donnent une représentation hiérarchique des informations de la base des registres. Celles-ci sont regroupées comme pour un système de fichiers, sauf qu'au lieu de répertoires on parle de clés et de sous-clés. Sous NT on préfère utiliser `regedt32.exe`.

Il est déconseillé d'accéder à la base des registres à partir de vos applications .NET pour deux raisons :

- Il ne faut jamais y sauver la configuration de vos applications. En effet, les fichiers de configuration XML décrits tout au long de cet ouvrage ont été conçus à cette fin. Ils ont l'avantage d'être physiquement stockés dans le même répertoire que l'application. On peut ainsi réaliser un déploiement type XCopy.
- La base des registres n'existe principalement que sous les systèmes d'exploitation *Microsoft*. Bien que cette technique de base des registres ait été portée sous d'autres systèmes d'exploitation, son utilisation hors du monde *Microsoft* reste marginale. Si vous accédez à la base des registres, votre application ne pourra pas être portée facilement vers un autre système d'exploitation supportant .NET.

## *Structure du registre*

La hiérarchie des informations du registre est la suivante (en partant du plus général au plus particulier) :

- Les clés racines (préfixée par HKEY_) ;
- les clés ;
- les sous-clés ;
- les entrées de valeur.

Les clés racines contiennent des clés qui, elles-mêmes, contiennent des sous-clés. On parle aussi de *ruches* qui sont des regroupements de clés. Une entrée est constituée de trois parties :

- le nom de la valeur ;
- le type de donnée ;
- la valeur elle-même.

Chaque valeur a l'un des types suivants. Nous précisons le type utilisé dans la base des registres. Nous précisons aussi le type .NET associé à une valeur récupérée de la base des registres dans du code .NET.

| Type utilisé dans la base des registres | Type .NET | Description |
|---|---|---|
| REG_DWORD | System.Int32 | Nombre codé sur quatre octets. Les paramètres de services et des pilotes de périphériques, les adresses mémoire et les interruptions sont en général de ce type. |
| REG_SZ | System.String | Chaîne de caractères au format Unicode. |
| REG_MULTI_SZ | System.String[] | Tableau de chaînes de caractères au format Unicode. |
| REG_EXPAND_SZ | System.String | Chaîne de caractères extrapolables au format Unicode : ce format est utilisé pour les chaînes de caractères formées de variables d'environnement comme %SystemRoot%. |
| REG_BINARY | System.Byte[] | Données au format binaire. Par exemple des informations concernant les composants matériels. |

## *Hiérarchie du registre*

Le registre est structuré en cinq clés racines :

- HKEY_CLASSES_ROOT : Contient les sous-clés suivantes :
  - Les extensions de nom de fichier : on y trouve par exemple la clé `.zip` permettant d'associer les fichiers d'extension `.zip` à l'application *WinZip* (ou à une autre application) par l'intermédiaire des sous-clés de définition de classes.
  - Les PROGID des classes COM enregistrées sur la machine : pour chaque PROGID on définit son CLSID, et éventuellement des attributs comme le numéro de version de la classe.
  - CLSID : contient tous les CLSID de toutes les classes COM enregistrées sur la machine. Chaque CLSID contient le PROGID, le chemin et le nom du fichier qui contient l'implémentation de la classe, et éventuellement des attributs, comme le mode d'exécution des objets de la classe.
- HKEY_CURRENT_USER : comporte le profil du dernier utilisateur qui a ouvert une session sur la machine. Cette clé référence la clé HKEY_USERS qui contient les dernières données en cours, les paramètres du bureau et de l'imprimante, les connexions réseau et les préférences d'application. Une application peut par exemple y stocker une grande quantité de préférences comme la taille de la fenêtre principale, la position des sous fenêtres ou les outils disponibles par icone.
- HKEY_LOCAL_MACHINE : renferme les informations matérielles et logicielles spécifiques de l'ordinateur local. Ces données sont indépendantes de l'utilisateur courant. Cette clé comporte les sous-clés suivantes :

  - HARDWARE : cette base de données est reconstituée à chaque lancement du système. Il s'agit d'une description du matériel : les données du BIOS, de la couche d'abstraction du matériel, les paramètres des adaptateurs SCSI et des cartes vidéo. Le programme de diagnostics de *Windows NT* prend ici ses informations.
  - SAM : base de données de sécurité des domaines (dans NT Server) et des comptes d'utilisateurs de groupes (dans NT Workstation). Les informations présentes dans cette clé sont reproduites dans la clé HKEY_LOCAL_MACHINE\SECURITY\SAM.
  - SECURITY : le sous-système de sécurité de NT tire de cette sous-clé les stratégies de sécurité et les droits des utilisateurs.
  - SOFTWARE : contient les données relatives aux logiciels installés sur la machine.
  - SYSTEM : renseignements sur le démarrage (services, pilotes etc) et le comportement du système.
- HKEY_USERS : contient toutes les données de profil de l'utilisateur courant, ainsi que les données de profil de tous les utilisateurs qui se sont connectés à la machine. On y trouve aussi le profil par défaut (sous-clé .DEFAULT). La clé HKEY_USERS\.DEFAULT constitue une ruche dont les fichiers correspondants sont DEFAULT et DEFAULT.LOG. L'éditeur de stratégie système permet de modifier et de créer des profils d'utilisateurs.
- HKEY_CURRENT_CONFIG : comporte les données de configuration du profil matériel actif, parmi lesquelles les paramètres d'affichage et des pilotes périphériques.

## *Lecture/écriture dans la base des registres*

La classe `Microsoft.Win32.RegistryKey` a été spécialement conçue pour avoir accès en lecture et en écriture aux données de la base des registres. Voici un exemple qui montre comment lire la valeur d'une clé :

*Exemple 16-8 :*

```
// Accès en lecture à la clé : // b vaut false
using System ;
using Microsoft.Win32 ;
class Program {
   static void Main() {
      string[] sTab = new String[4] ;
      sTab[0] = "HKEY_LOCAL_MACHINE" ;
      sTab[1] = "SOFTWARE" ;
      sTab[2] = "Microsoft" ;
      sTab[3] = ".NETFramework" ;
      string sSubKey = "DbgManagedDebugger" ;
      string sKey = sTab[0] ;
      RegistryKey rKey = Registry.LocalMachine ;
      for (int i = 1 ; i < sTab.Length ; i++) {
         sKey += "/" + sTab[i] ;
         rKey = rKey.OpenSubKey(sTab[i]) ;
      }
      Console.WriteLine("Valeur de la clé  {0}/{1}", sKey, sSubKey) ;
      Console.WriteLine(rKey.GetValue(sSubKey)) ;
```

```
    }
}
```

Cet exemple affiche :

```
Valeur de la clé  HKEY_LOCAL_MACHINE/SOFTWARE/Microsoft/.NETFramework/
DbgManagedDebugger
C:\Program Files\Fichiers communs\Microsoft Shared\VS7Debug\vs7jit.exe
PID %d AP PDOM %d EXTEXT "%s" EVTHDL %d
```

# *Le débogage*

## *Attributs pour personnaliser la visualisation des états de vos objets*

L'espace de noms `System.Diagnostics` présente des attributs qui vous permettent de personnaliser complètement l'affichage des états de vos objets durant le débogage :

- `DebuggerDisplayAttribute`
  S'applique aux classes, structures, délégations, énumérations, champs, propriétés et assemblages. Cet attribut permet de personnaliser la ligne de description d'un objet fourni par le débogueur. Cette ligne peut contenir des expressions entre accolades qui peuvent être le nom d'un champ, d'une propriété ou d'une méthode. Comprenez bien qu'une telle ligne doit aller à l'essentiel en restant concise :

  *Exemple 16-9 :*

```
using System.Diagnostics ;
[DebuggerDisplay("{Description} prix:{m_Prix} euros")]
class Article {
   private decimal m_Prix ;
   private string m_Description ;
   public string Description{
      get{return m_Description;}
   }
   public Article(string description,decimal prix) {
      m_Description = description ;
      m_Prix = prix ;
   }
}
class Program  {
   static void Main() {
      Article article = new Article("Chaussure", 120) ;
   }
}
```

- `DebuggerBrowsable`
  S'applique aux champs et propriétés. Permet de signifier au débogueur comment il doit afficher ce champ ou cette propriété à l'aide d'une des trois valeurs de l'énumération `DebuggerBrowsableState` à savoir : `Collapsed` : le débogueur affiche la valeur du champ ou de la propriété lorsque l'état de l'objet est déplié (c'est le comportement par défaut) ;

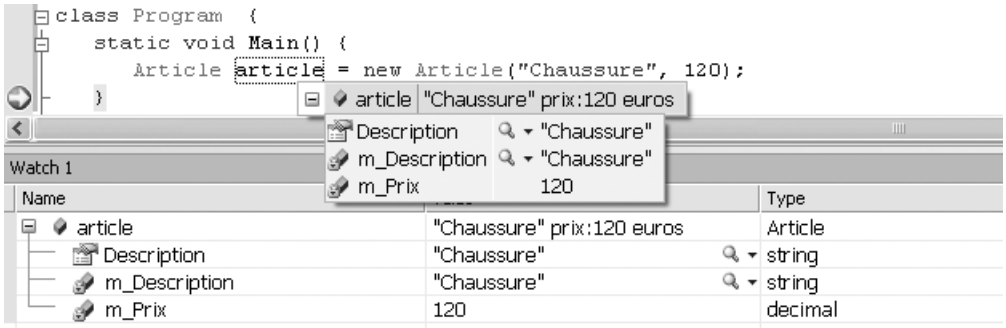


*Figure 16-1 : Utilisation de l'attribut DebuggerDisplay*

`RootHidden` : le débogueur n'affiche pas la valeur lorsque l'état de l'objet est déplié mais affiche les valeurs des sous membres du champ ou de la propriété concernée ; `Never` : le débogueur n'affiche pas la valeur lorsque l'état de l'objet est déplié.

- `DebuggerTypeProxyAttribute`

  S'applique aux structures, classes et assemblages. Cet attribut appliqué à un type `T` permet de spécifier un type proxy spécialisé dans l'affichage de l'état d'une instance de `T`. En général, ce type proxy est une classe privée encapsulée dans le type `T` du fait qu'une telle classe a accès à tous les membres privés de la classe encapsulante :

  *Exemple 16-10 :*

```
[System.Diagnostics.DebuggerTypeProxy(typeof(ArticleProxy))]
class Article {
   private class ArticleProxy {
      private Article m_Article;
      public ArticleProxy(Article article) {m_Article = article;}
      public string Prix { get{ return m_Article.m_Prix + " euros";}}
   }
   private decimal m_Prix ;
   private string m_Description ;
   public Article(string description,decimal prix) {
      m_Description = description ;
      m_Prix = prix ;
   }
}
class Program  {
   static void Main() {
      Article article = new Article("Chaussure", 120) ;
   }
}
```

  Notez la présence de la ligne `Raw View` qui permet d'obtenir la vue par défaut de l'état de l'objet :

- `DebuggerVisualizerAttribute`

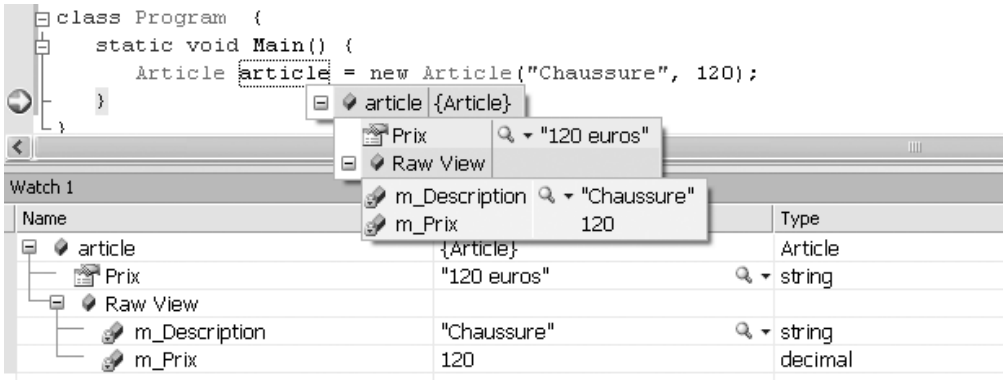


*Figure 16-2 : Utilisation de l'attribut DebuggerTypeProxy*

S'applique aux structures, classes et assemblages. Cet attribut constitue la pierre angulaire d'un système très puissant de visualisation d'état des objets durant le débogage. L'idée est de dessiner l'état d'un objet directement dans une fenêtre avec GDI+. On peut ainsi visualiser au débogage une image, un objet en 3D, un graphe etc. Plus d'information concernant l'utilisation de cet attribut sont disponibles dans les **MSDN**.

## *Attributs permettant de ne se focaliser que sur son code lors du débogage*

L'espace de noms `System.Diagnostics` présente des attributs vous permettant de spécifier au débogueur des assemblages, des modules ou des zones de code que vous ne souhaitez pas déboguer. En général on exploite ces attributs pour ne pas être gêné lors du débogage par le code des bibliothèques que l'on utilise :

- `DebuggableAttribute`
  S'applique aux assemblages et aux modules. Cet attribut permet de spécifier ou non l'ensemble des modes de débogages que le compilateur JIT doit appliquer sur le code IL de l'assemblage ou du module ciblé. Ces modes sont décrits par l'indicateur binaire `DebuggableAttribute.DebuggingModes`. On peut citer le mode *Edit and Continue*, le mode *optimisations JIT désactivées*, le mode *ignorer les informations du fichier* `.pdb` ou le mode qui permet d'activer la production par le compilateur JIT des informations qui établissent le lien entre le code IL et le code machine généré (*JIT tracking*).
- `DebuggerHiddenAttribute`
  S'applique aux constructeurs, aux méthodes et aux propriétés. Cet attribut permet de spécifier que le débogueur ne doit pas entrer dans cette méthode. En conséquence, vous ne pouvez mettre de point d'arrêt dans une méthode marquée avec cet attribut. En revanche, si une méthode marquée avec cet attribut appelle une méthode non marquée avec cet attribut, il est tout à fait possible de déboguer le code de la seconde méthode. Dans ce cas, une fenêtre de pile contenant `External Code` apparaîtra dans la pile d'appels du débogueur.
- `DebuggerStepThroughAttribute`

S'applique aux classes, structures constructeurs et méthodes. Cet attribut est comparable à `DebuggerHiddenAttribute` mis à part que dans la situation décrite, à la place d'une fenêtre de pile `External Code` il n'y aura pas de fenêtre de pile du tout.

- `DebuggerNonUserCodeAttribute`

  S'applique aux classes, structures, constructeurs, méthodes et propriétés. Cet attribut est comparable à `DebuggerHiddenAttribute`.

## *Résoudre les problèmes de débogueur*

Il arrive parfois que vous ne puissiez pas déboguer votre application correctement. *MinKwan Park* a écrit un excellent article qui décrit de nombreuses situations problématiques ainsi que leurs résolutions. Une version traduite en français par *Frédéric De Lène Mirouze* est disponible sur le site *dotnetguru.org* à l'adresse `http://www.dotnetguru.org/articles/dossiers/debug/debug.htm`.

# *Les traces*

Le *framework* .NET présente les deux classes `System.Diagnostics.Trace` et `System.Diagnostics.Debug` qui vous permettent de tracer le déroulement d'un programme. On parle aussi parfois de *log* de l'exécution d'un programme. Chacune de ces classes présente les quatre méthodes `Write(string)`, `WriteLine(string)`, `WriteIf(bool,string)` et `WriteLineIf(bool,string)` qui peuvent par exemple s'utiliser comme ceci :

*Exemple 16-11 :*

```
using System.Diagnostics ;
class Program {
   static void Main() {
      Trace.Listeners.Add( new ConsoleTraceListener() ) ;
      Trace.WriteLine("Trace hello");
      for( int i=0;i<5;i++)
         Debug.WriteLineIf(i > 2, "debug i=" + i.ToString() );
   }
}
```

Voici l'affichage de ce programme sur la console :

```
Trace hello
debug i=3
debug i=4
```

La classe `Trace` est plutôt dédiée aux log d'information sur le domaine fonctionnel de l'application tandis que la classe `Debug` est destinée à tracer des informations utiles au débogage. De ce fait, elles sont relativement similaires bien qu'elles doivent être utilisées à des fins différentes. Ces deux classes ne sont opérationnelles que si l'assemblage qui les utilise est compilé avec respectivement la constante symbolique `TRACE` ou/et `DEBUG`.

## *Listener*

L'Exemple 16-11 commence par la ligne :

```
Trace.Listeners.Add( new ConsoleTraceListener() ) ;
```

Cela signifie que la liste des *listeners* va contenir un *listener* de type `ConsoleTraceListener`. Un *listener* est une instance d'une classe qui dérive de la classe `System.Diagnostics.TraceListener`. La liste de listener est commune aux deux classes `Debug` et `Trace`. Lorsqu'une trace est loguée par une de ces classes, chacun des listeners de la liste écrit la trace sur son propre moyen de persistance, en l'occurrence, la console pour un listener de type `ConsoleTraceListener`. D'autres classes de listener sont prévues par le framework (les classes en gras ont été ajoutées par la version 2.0) :

```
System.Diagnostics.TraceListener
   Microsoft.VisualBasic.Logging.FileLogTraceListener
   System.Diagnostics.DefaultTraceListener
   System.Diagnostics.EventLogTraceListener
   System.Diagnostics.TextWriterTraceListener
      System.Diagnostics.ConsoleTraceListener
      System.Diagnostics.DelimitedListTraceListener
      System.Diagnostics.XmlWriterTraceListener
   System.Web.WebPageTraceListener
```

- `DefaultTraceListener` : Trace dans le flot de données dédié au débogage. Par défaut, ce flot de données est dirigé vers la fenêtre de sortie de *Visual Studio*.
- `EventLogTraceListener` : Trace dans le journal de log de *Windows*.
- `TextWriterTraceListener` : Trace dans un flot de données tel qu'un fichier.
- `DelimitedListTraceListener` : Trace dans un flot de données tel qu'un fichier. À la différence de `TextWriterTraceListener`, les traces sont séparées par une chaine de caractères (tel qu'un point virgule par exemple) de façon à être facilement exploitables par un programme d'analyse de trace.
- `XMLWriterTraceListener` : Trace dans un fichier XML qui suit un certain schéma XML.
- `WebPageTraceListener` : Utilisée par ASP.NET. Trace directement sur la page web en cours de construction.

Vous avez la possibilité de construire la liste de listener par l'intermédiaire du fichier de configuration. Par exemple, ce fichier de configuration ajoute un listener de type `ConsoleTraceListener` à la liste :

*Exemple 16-12 :*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration
  xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <system.diagnostics>
      <trace>
         <listeners>
            <add name="TraceOnConsole"
                 type="System.Diagnostics.ConsoleTraceListener"/>
```

```
         </listeners>
      </trace>
   </system.diagnostics>
</configuration>
```

Si l'Exemple 16-11 est utilisé conjointement avec ce fichier, la liste de listeners contiendra deux listeners qui tracent sur la console et le programme affichera chaque trace en double :

```
Trace hello
Trace hello
debug i=3
debug i=3
debug i=4
debug i=4
```

Enfin, sachez que vous pouvez développer vos propres classes de listener dérivées de la classe `TraceListener`.

## *Sources de traçage et gravité des traces*

Le framework .NET 2.0 introduit la classe `System.Diagnostics.TraceSource` qui permet de gérer plusieurs *sources de traçage*. Chaque source de traçage est nommée et a sa propre liste de listeners. Les sources de traçage ne sont activées à l'exécution que si la constante symbolique `TRACE` est prévue à la compilation. En général, les sources de traçage sont accessibles à partir de champs statiques d'une classe visible à partir de toute l'application.

Un système de filtre sur la gravité d'une trace est implémenté. Voici un exemple d'utilisation de ce système. Ici, le filtre accepte toutes les traces (`SourceLevels.All`) et nos traces ont la gravité *information* (`TraceInformation()`) et *avertissement* (`TraceEventType.Warning`) :

*Exemple 16-13 :*

```
using System.Diagnostics ;
class Program {
   static void Main() {
      Trace.Listeners.Add(new ConsoleTraceListener()) ;
      TraceSource trace1 = new TraceSource("Trace1") ;
      trace1.Listeners.Add(new ConsoleTraceListener()) ;
      trace1.Switch.Level = SourceLevels.All;
      trace1.TraceInformation("Trace hello") ;
      for (int i = 0 ; i < 2 ; i++)
         trace1.TraceData(TraceEventType.Warning, 122,
                          "debug i=" + i.ToString()) ;
   }
}
```

Ce programme affiche ceci sur la console :

```
Trace1 Information: 0 : Trace hello
Trace1 Warning: 122 : debug i=0
Trace1 Warning: 122 : debug i=1
```

Il n'aurait pas affiché la trace `Trace Hello` si nous avions choisi `SourceLevels.Warning`. Le concept de *switch* peut être aussi implémenté avec des instances de la classe `SourceSwitch` :

*Exemple 16-14 :*

```
...
      TraceSource trace1 = new TraceSource("Trace1") ;
      SourceSwitch switch1 = new SourceSwitch("Switch1");
      switch1.Level = SourceLevels.All;
      trace1.Switch = switch1;
...
```

Une source de traçage est paramétrable après compilation grâce au fichier de configuration. Avec le fichier de configuration suivant, nous n'aurions qu'à créer une source de traçage nommée `Trace1` dans notre code :

*Exemple 16-15 :*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration
  xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <system.diagnostics>
      <sources>
         <source name="Trace1" switchName="Switch1">
            <listeners>
               <add name="TraceOnConsole"
                    type="System.Diagnostics.ConsoleTraceListener"/>
            </listeners>
         </source>
      </sources>
      <switches>
         <add name="Switch1" value="All" />
      </switches>
   </system.diagnostics>
</configuration><labelstart id="_Toc114113400"/>
```

## *Filtrage des sources de traçage au niveau des listeners*

Un second niveau de filtrage existe au niveau des listeners. Un listener a la possibilité de n'accepter que des traces issues d'une certaines source. Ici, le filtre affecté au listener de type `ConsoleTraceListener` n'accepte que les traces issues de la source fictive nommée `TraceXXX` et en conséquence, ce programme n'affiche rien :

*Exemple 16-16 :*

```
...
      TraceSource trace1 = new TraceSource("Trace1") ;
      ConsoleTraceListener listener = new ConsoleTraceListener() ;
      listener.Filter = new SourceFilter("TraceXXX");
      trace1.Listeners.Add(listener);
...
```

## *Indentation des traces*

Les classes Debug et Trace présentent des méthodes Indent() et Unindent() pour permettre d'incrémenter et de décrémenter le niveau d'indentation des traces. En outre la propriété IndentSize définit le nombre d'espaces pour un niveau d'indentation :

*Exemple 16-17 :*

```
using System.Diagnostics ;
class Program {
   static void Main() {
      Trace.Listeners.Add(new ConsoleTraceListener()) ;
      Trace.IndentSize = 3;
      Trace.WriteLine("Begin Main()") ;
      fct( 2 ) ;
      Trace.WriteLine("End Main()") ;
   }
   static void fct( int i ) {
      Trace.Indent();
      Trace.WriteLine( "Begin fct(" + i.ToString() + ")" ) ;
      if (i > 0)
         fct(i-1) ;
      Trace.WriteLine( "End fct(" + i.ToString() + ")" ) ;
      Trace.Unindent();
   }
}
```

Ce programme affiche ceci sur la console :

```
Begin Main()
   Begin fct(2)
      Begin fct(1)
         Begin fct(0)
         End fct(0)
      End fct(1)
   End fct(2)
End Main()
```

La classe TraceSource ne présente pas ces facilités. Le programme suivant montre comment pallier ce manque au moyen d'une propriété IndentLevel qui calcule le niveau d'indentation en fonction de la taille courante de la pile d'appel du thread courant. Ainsi, l'affichage de ce programme est identique à celui du précédent :

*Exemple 16-18 :*

```
using System.Diagnostics ;
class Program {
   static TraceSource trace1 ;
   const int IndentSize = 3;
   static string IndentLevel { get{
         StackTrace stackTrace = new StackTrace();
         return new string(' ',(stackTrace.FrameCount - 2)* IndentSize);
```

```
      }
   }
   static void Main() {
      trace1 = new TraceSource("Trace1") ;
      trace1.Listeners.Add(new ConsoleTraceListener()) ;
      trace1.Switch.Level = SourceLevels.All ;
      trace1.TraceInformation( IndentLevel + "Begin Main()") ;
      fct(2) ;
      trace1.TraceInformation( IndentLevel + "End Main()") ;
   }
   static void fct(int i) {
      Trace.Indent() ;
      trace1.TraceInformation( IndentLevel +
                              "Begin fct(" + i.ToString() + ")" ) ;
      if (i > 0)
         fct(i-1) ;
      trace1.TraceInformation( IndentLevel +
                              "End fct(" + i.ToString() + ")" ) ;
      Trace.Unindent() ;
   }
}
```

# *Les expressions régulières*

## *Introduction*

Les *expressions régulières* (*regular expression* en anglais, souvent nommées *regexp* ou *regex*) constituent un outil puissant pour réaliser trois grands types d'opérations sur les chaînes de caractères :

- Les expressions régulières permettent de **vérifier** qu'une chaîne de caractères vérifie une règle syntaxique (par exemple si une adresse mail contient bien des caractères avant et après le caractère @ et si le nom du serveur a une extension valide comme *.com*, *.fr* ou *.edu*).
- Les expressions régulières permettent de **modifier** une chaîne de caractères en remplaçant un élément par un autre.
- Les expressions régulières permettent d'**extraire** des éléments d'une chaîne de caractères.

La notion d'expression régulière a surtout été popularisée par la commande *grep* sous Unix qui permet le filtrage de données. Le filtre est alors paramétré par une expression régulière.

Nous allons maintenant présenter quelques points principaux sur les expressions régulières sans toutefois rentrer dans un exposé exhaustif qui sortirait du cadre de ce livre.

## *Syntaxe*

Une expression régulière se présente comme une chaîne de caractères. Chaque caractère de cette chaîne a une signification précise. Voici les plus communs :

| Caractère(s) | Signification |
|---|---|
| \ | Annule le sens d'un méta caractère. |
| ^ | Début de ligne. |
| . | N'importe quel caractère. |
| $ | Fin de ligne. |
| x\|y | x ou y. |
| () | Groupement. |
| [xyz] | Ensemble de caractères (x ou y ou z) |
| [x-z] | Intervalle de caractères (x ou y ou z) |
| [^x] | Tous sauf x |

Vous pouvez aussi spécifier le nombre d'occurrences d'une expression régulière avec les expressions suivantes :

| Caractère(s) | Signification |
|---|---|
| x* | Zéro, une ou plusieurs occurrences de x. |
| x+ | Une ou plusieurs occurrences de x. |
| x? | Zéro ou une occurrence de x. |
| x{n} | Exactement n occurrences de x. |
| x{n,} | Au moins n occurrences de x. |
| x{n,m} | Au moins n, au plus m occurrences de x. |

Enfin, il existe des alias bien pratiques :

| Caractère(s) | Alias pour | Signification |
|---|---|---|
| \n | | Un caractère de fin de ligne. |
| \r | | Un retour à la ligne. |
| \t | | Une tabulation. |
| \s | [\f\n\r\t\v] | Un espacement. |
| \S | [^\f\n\r\t\v] | Tout ce qui n'est pas un espacement |

| | | |
|---|---|---|
| \d | [0-9] | Un chiffre. |
| \D | [^0-9] | Tout, sauf un chiffre. |
| \w | [a-zA-Z0-9_] | Un caractère alphanumérique. |
| \W | [^a-zA-Z0-9_] | Tout sauf un caractère alphanumérique. |
| \067 | 7 | Un caractère en octal. |
| \x5A | Z | Un caractère en hexadécimal. |

## *Exemples*

Voici quelques exemples d'expressions régulières et de leurs significations :

- L'expression régulière ci-dessous vérifie si une chaîne de caractères commence par une lettre majuscule.

```
[A-Z].*
```

- L'expression régulière ci-dessous vérifie si une chaîne de caractères contient au moins un caractère non alphanumérique.

```
.*[^0-9A-Za-zÀ-ÿ].*
```

- L'expression régulière ci-dessous vérifie si une chaîne de caractères se termine par « el » ou « elle ».

```
.*(el|elle)
```

- L'expression régulière ci-dessous vérifie si une chaîne de caractères est une date au format yyyy-mm-dd entre 1900 01-01 et 2099 12-31.

```
(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])
```

## *Le framework .NET et les expressions régulières*

La classe `System.Text.RegularExpressions.RegEx` permet d'utiliser les expressions régulières pour vérifier une règle syntaxique sur une chaîne de caractères, pour modifier une chaîne de caractères ou pour extraire des éléments d'une chaîne de caractères. Le programme suivant expose ces possibilités :

*Exemple 16-19 :*

```
using System ;
using System.Text.RegularExpressions ;
public class Program {
   static void Main() {
      // Utilisation d'une regexp pour vérifier qu'une chaine de
      // caractères ne contient pas de chiffres.
      bool b ;
```

```
      Regex regex1 = new Regex("^[^0-9]*$") ;
      b = regex1.IsMatch("ab3de") ; // b vaut false
      b = regex1.IsMatch("abcde") ; // b vaut true
      // Remplacement du mot 'belle' par le mot 'jolie'.
      Regex regex2 = new Regex("belle") ;
      // Affiche 'Elle est jolie.'.
      Console.WriteLine(regex2.Replace("Elle est belle.", "jolie")) ;
      // Extrait chacun des mots séparés par un espace.
      Regex regex3 = new Regex("[ ]") ;
      string[] mots=regex3.Split("la terre est bleue comme une orange") ;
      foreach (string mot in mots)
         Console.WriteLine(mot) ;
   }
}
```

## *Optimiser l'évaluation d'expressions régulières*

Nous expliquons page 255 comment optimiser certaines opérations grâce à la génération d'assemblages. Les expressions régulières se prêtent particulièrement bien à ce type d'optimisation. Aussi, la classe `RegExp` présente la méthode statique `CompileToAssembly()` qui permet de réaliser simplement une telle optimisation. Le programme produit un assemblage nommé `regexp.dll` qui contient la classe `CompiledExpressions.PasDeChiffres` qui dérive de la classe `RegEx`.

*Exemple 16-20 :*

```
using System.Reflection ;
using System.Text.RegularExpressions ;
public class Program {
   static void Main() {
      string sExpression = "^[^0-9]*$" ;
      RegexCompilationInfo reci = new RegexCompilationInfo(
                  sExpression,
                  RegexOptions.Compiled,
                  "PasDeChiffres",
                  "CompiledExpressions",
                  true) ;
      RegexCompilationInfo[] recis = new RegexCompilationInfo[] {reci} ;
      AssemblyName assemblyName = new AssemblyName() ;
      assemblyName.Name = "Regex" ;
      Regex.CompileToAssembly(recis, assemblyName) ;
   }
}
```

L'exemple suivant montre comment mesurer le facteur d'optimisation d'une expression régulière compilée :

*Exemple 16-21 :*

```
using System ;
using System.Text.RegularExpressions ;
```

```
using CompiledExpressions ;
using System.Diagnostics ;

public class Program {
   const int NLOOP = 1000000 ;
   const string str = "abcdefghijklmnopqrstuvwxyz" ;
   static void Main() {
      bool b ;
      Regex regex1 = new Regex("^[^0-9]*$") ;
      Stopwatch sw = Stopwatch.StartNew();
      for (int i = 0 ; i < NLOOP ; i++)
         b = regex1.IsMatch(str) ;
      Console.WriteLine("Sans précompilation:{0}", sw.Elapsed );

      sw = Stopwatch.StartNew();
      PasDeChiffres regex2 = new PasDeChiffres() ;
      for (int i = 0 ; i < NLOOP ; i++)
         b = regex2.IsMatch(str) ;
      Console.WriteLine("Avec précompilation:{0}", sw.Elapsed );
   }
}
```

Nous avons effectué différents tests avec l'expression régulière pour vérifier qu'il n'y a pas de chiffre (`"^[^0-9]*$"`) et l'expression régulière pour vérifier une date (`@"(19|20)\d\d[-/.](0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])"`). Les chaînes testées étaient l'alphabet (`"abcdefghijklmnopqrstuvwxyz"`) et un double alphabet (`"abcdefghijklmnopqrstuvwxyz abcdefghijklmnopqrstuvwxyz"`). Voici les facteurs d'optimisation obtenus :

| Expression régulière | Chaîne testée | Facteur d'optimisation |
|---|---|---|
| Pas de chiffres | Alphabet | 1,21 |
| Pas de chiffres | double alphabet | 1,12 |
| Vérification de date | Alphabet | 3,99 |
| Vérification de date | double alphabet | 4,03 |

# *La console*

La classe `System.Console` permet de contrôler l'affichage et la saisie de données sur la console. Toutes les fonctionnalités de cette classe sont accessibles à partir de membres statiques puisqu'une application a au plus une console.

## *Le curseur de la console*

La console supporte un curseur qui représente l'endroit où le prochain affichage de données va s'effectuer.

- Par défaut le curseur s'affiche après le dernier caractère affiché mais vous pouvez fixer sa position avec la méthode `SetCursorPosition(int column,int row)`.
- Vous pouvez obtenir ou modifier la position courante du curseur avec les propriétés `int CursorTop{get;set;}` et `int CursorLeft{get;set;}`.
- Vous pouvez choisir de rendre le curseur visible ou non avec la propriété `int CursorVisible{get;set;}`.
- Vous pouvez obtenir ou positionner la taille du curseur en utilisant la propriété `int CursorSize{get ;set}` qui prend ses valeurs dans l'intervalle fermé `[1,100]`. La taille du curseur désigne le pourcentage du remplissage du rectangle du curseur.

## *Affichage de données*

- Vous pouvez afficher une chaîne de caractère à l'endroit où le curseur est positionné en appelant une des surcharges de la méthode `Write()`. La méthode `WriteLine()` oblige le curseur à passer à une nouvelle ligne après avoir afficher les données.
- La méthode `MoveBufferArea(int sourceLeft, int sourceTop, int sourceWidth, int sourceHeight, int targetLeft, int targetTop)` permet de copier un bloc de données. Le rectangle source spécifié doit être complètement visible. Il se peut qu'une partie des données soient tronquée si le rectangle destination n'est pas strictement contenu dans la partie visible de console.
- La méthode `Clear()` permet d'effacer toutes les données couramment affichées sur la console et de mettre le curseur en haut à gauche.

## *Taille et position de la console*

- Vous pouvez modifier la position de la console sur l'écran au moyen de la méthode `SetWindowsPosition(int left, int top)`. Les paramètres désignent le nombre de pixels par rapport au coin haut gauche de l'écran. Vous pouvez obtenir ou modifier la position de la console au moyen des propriétés `int WindowLeft{get;set;}` et `int WindowTop{get;set;}`.
- Vous pouvez modifier la taille de la console au moyen de la méthode `SetWindowsSize(int nColumn, int nRow)`. Vous pouvez obtenir ou modifier la taille de la console (en terme de nombre de colonnes et de lignes) au moyen des propriétés `int WindowWidth{get;set;}` et `int WindowHeight{get;set;}`.
- Vous pouvez obtenir le plus grand nombre de colonnes et de lignes possibles en fonction de la résolution de l'écran au moyen des propriétés `int LargestWindowWidth{get;}` et `int LargestWindowHeight{get;}`.
- Le nombre de colonnes et de lignes que la console contient peut dépasser la taille de la console. Vous pouvez obtenir ou modifier ce nombre de colonnes et de lignes au moyen des propriétés `int BufferWidth{get;set;}` et `int BufferHeight{get;set;}`.

## *Couleur*

La classe `Console` vous permet de modifier la couleur des données que vous affichez en précisant les couleurs avec l'énumération `System.ConsoleColor`. Dans la suite nous utiliserons le terme couleur *background* pour désigner la couleur de fond et le terme couleur *foreground* pour désigner la couleur des caractères.

- Les propriétés `ConsoleColor ForegroundColor{get;set;}` et `ConsoleColor BackgroundColor{get;set;}` permettent d'obtenir ou de positionner les couleurs *foreground* et *background* des prochaines données affichées.
- Vous pouvez repositionner les couleurs *foreground* et *background* à leurs valeurs par défaut en appelant la méthode `ResetColor()`.

## *Saisie de données*

- Vous pouvez obtenir le prochain caractère tapé sur le clavier par l'utilisateur en appelant la méthode `int ReadKey()`. L'appel à cette méthode est bloquant jusqu'à ce qu'un caractère soit effectivement tapé sur le clavier. Ensuite le caractère correspondant à la touche appuyée est affiché sur la console. Vous pouvez convertir l'entier retourné en une instance de `char` en appelant la méthode `char Convert.ToChar(int)`. La surcharge `int ReadKey(bool)` permet de ne pas afficher le caractère sur la console lorsqu'elle prend en argument la valeur `false`.
- La propriété `bool KeyAvailable{get;}` permet de savoir si le prochain appel à la méthode `ReadKey()` va être bloquant ou non.
- La méthode `string ReadLine()` retourne la ligne qui vient d'être saisie par l'utilisateur. Cette méthode est bloquante jusqu'à ce que l'utilisateur appuie sur la touche *Entrée*.
- La propriété `bool TreatControlCAsInput{get;set;}` permet de connaître ou d'activer/désactiver le fait que l'événement *Ctrl-C* soit traité comme une entrée normale ou non. Si vous positionnez cette propriété à `false` vous pouvez alors exploiter l'événement `CancelPressedKey`.
- Les propriétés `bool NumberLock{get;}` et `bool CapsLock{get;}` permettent de savoir si les modes du clavier *NumLock* et *CapsLock* sont couramment activés.

## *Redirection de flux de données*

Vous pouvez modifier et obtenir les flux de données entrant, sortant et d'erreur de la console. Un flux de données est une instance d'une classe dérivée de la classe abstraite `System.IO.Stream`. Par défaut le flux de données entrant est le clavier, le flux de données sortant est la console et le flux de données d'erreurs est aussi la console.

Vous pouvez obtenir un de ces flux au moyen d'une des trois propriétés `TextReader In{get;}`, `TextWriter Out{get;}` ou `TextWriter Error{get;}`.

Vous pouvez positionner un de ces flux au moyen d'une des trois méthodes `void SetError(TextWriter newError)`, `void SetIn(TextReader newIn)` ou `void SetOut(TextWriter newOut)`.

## *Autres fonctionnalités*

- La méthode `Beep()` permet de faire jouer un beep. Une surcharge permet de modifier la fréquence et la durée du beep.
- La propriété `string Title{get ;set ;}` permet d'obtenir ou de positionner le titre de la console.

# 17

# *Les mécanismes d'entrée/sortie*

Les classes de base du *framework*.NET permettant de réaliser des entrées/sorties de données se trouvent pour la plupart dans l'espace de noms `System.IO`.

## *Introduction aux flots de données*

### *Hiérarchie des classes modélisant un flot de données*

La classe `System.IO.Stream` est la classe de base pour toutes les classes qui permettent d'effectuer des entrées sorties en modélisant un flot de données. La classe `Stream` est une classe abstraite.

Il existe trois types de manipulation d'un flot de données qui sont :

- L'accès en lecture aux données.
- L'accès en écriture aux données.
- L'accès aléatoire aux données (*seek* en anglais). Cette possibilité autorise l'utilisateur à déplacer le *curseur* en avant et en arrière dans le flot de données. Le curseur définit l'endroit dans le flot de données à partir duquel la prochaine lecture/écriture se fera. Si cette possibilité n'est pas supportée par un flot de données, l'utilisation des propriétés `Length` et `Position` et des méthodes `SetLength()` et `Seek()` de la classe `Stream` provoque l'envoi de l'exception `NotSupportedException`. Si cette propriété n'est pas supportée par un flot de données, on dit que le flot de données est *forward-only*. Ce terme anglais signifie que le curseur ne peut être déplacé qu'en avant.

Vous pouvez tester quelles sont les manipulations supportées par un flot de données en utilisant les propriétés `CanRead`, `CanWrite` et `CanSeek` de la classe `Stream`. Voici la liste des classes dérivant de la classe `Stream` :

- `System.IO.FileStream`

Cette classe permet de modéliser un flot de données vers un fichier. Ce type de flots de données est décrit page 637. Il peut éventuellement supporter les trois types d'accès.

- `System.IO.IsolatedStorage.IsolatedStorageFileStream`
  Cette classe dérive de la classe `FileStream`. Elle permet de réaliser la fonctionnalité de *stockage isolé* décrit page 205, dans le chapitre relatif à la sécurité.
- `System.Net.Sockets.NetworkStream`
  Cette classe permet de modéliser un flot de données vers une entité distante en utilisant des sockets (voir page 641). Ce type de flots de données ne supporte jamais l'accès aléatoire.
- `System.IO.Ports.SerialPort`
  Cette classe permet de fournir un flot de données vers un port série. Ce type de flots de données ne supporte jamais l'accès aléatoire.

## *Services sur un flot de données*

Le *framework* .NET propose une architecture pour permettre l'ajout de services à des flots de données. Cette architecture est décrite en page 657. Voici la liste des classes représentant des services sur les flots de données. Elles sont toutes dérivées de la classe `Stream` :

- `System.Net.Security.AuthenticatedStream`
  Les classes dérivées de cette classe sont des implémentations de protocoles de sécurisations de flot de données par authentification et encryptions (voir page 660).
- `System.Security.Cryptography.CryptoStream`
  Cette classe permet de crypter les données d'un flot de données. Un flot qui utilise ce service ne supporte jamais l'accès aléatoire (voir page 664).
- `System.IO.BufferedStream`
  Cette classe est utilisée conjointement avec un flot de données pour lui fournir une mémoire tampon. L'utilisation de `BufferedStream` permet d'améliorer les performances dans certains cas, comme ceux exposés page 658. `BufferedStream` supporte les types d'accès du flot de données sous-jacent.
- `System.IO.MemoryStream` et `System.IO.UnmanagedMemoryStream`
  Cette classe permet d'obtenir une mémoire tampon partageable entre plusieurs flots de données (voir page 659). La version non gérée (*unmanaged*) permet d'éviter la copie des données sur le tas par le CLR et est donc plus efficace.
- `System.IO.Compression.DeflateStream System.IO.Compression.GZipStream`
  Ces classes permettent de compresser les données d'un flot (voir page 659).

## *Typer les données d'un flot de données*

En plus des classes modélisant un flot de données, l'espace de noms `System.IO` présente plusieurs couples de classes permettant de typer en entrée ou en sortie les octets d'un flot de données modélisé par une des classes présentées ci-dessus. Chaque couple contient une classe pour la lecture et une classe pour l'écriture. Voici la liste de ces couples de classes :

- `System.IO.BinaryWriter` et `System.IO.BinaryReader`
  Ces classes permettent de lire et d'écrire des données codées dans un des types primitifs .NET à partir d'un flot de données. Ces classes présentent des méthodes telles que `short ReadInt16()`, `void Write(short)`, `short ReadDouble()` ou `void Write(double)`.

- `System.IO.TextWriter` et `System.IO.TextReader`
  Ces classes sont abstraites. Elles servent de classes de base aux classes permettant de lire et d'écrire des caractères encodés dans un format particulier. La présentation de ces formats fait l'objet de la prochaine section. Ces classes présentent des méthodes telles que `string ReadLine()` ou `void WriteLine(string)`.
- `System.IO.StringWriter` et `System.IO.StringReader`
  Ces classes sont dérivées respectivement de `TextWriter` et de `TextReader`. Elles sont utilisées pour écrire et lire des caractères (ou des chaînes de caractères) vers une (ou à partir d'une) chaîne de caractères.
- `System.IO.StreamWriter` et `System.IO.StreamReader`
  Ces classes sont dérivées respectivement de `TextWriter` et de `TextReader`. Elles sont utilisées pour écrire et lire des caractères (ou des chaînes de caractères) encodés dans un format particulier, vers un (ou à partir d'un) flot de données quelconque.

## *Encodage des chaînes de caractères*

La problématique de l'*encodage des caractères* est d'associer un caractère tel qu'une lettre, un chiffre ou un symbole, à une représentation binaire. Un des premiers grands standard concernant l'encodage des caractères fut la norme *ASCII* 7 bits (*American Standard Code for Information Interchange*), qui représentait 128 caractères tels que les 26 lettres minuscules et les 26 lettres majuscules de notre alphabet ou encore les 10 chiffres. Dans la norme ASCII 7 bits, le $8^{\text{ième}}$ bit est soit positionné à zéro, soit utilisé comme bit de parité.

La norme ASCII 7 bits était suffisante pour des applications anglo-saxonnes. Rapidement, la nécessité de représenter les lettres accentuées des alphabets européens est apparue. Chaque langue ayant des lettres accentuées différentes, on ne pouvait pas encoder tous ces caractères avec 256 valeurs. Il est alors apparu différents jeux de 256 caractères *OEM* (*Original Equipment Manufacturer*). Ces jeux de caractères sont appelés *pages de code* (*codepage* en anglais). Il existe une page de code pour les langues nordiques, une page de code pour le français tel que l'écrivent les québécois etc.

Les pages de code OEM sont satisfaisantes pour les Européens et les Américains. Néanmoins, elles ne peuvent absolument pas représenter les alphabets des langues telles que le chinois ou l'arabe. On a donc inventé la norme *UNICODE*. Cette norme associe à (pratiquement) chaque combinaison possible de 16 bits, un caractère parmi les caractères des alphabets européens, asiatiques, arabes etc, mais aussi des symboles mathématiques, des symboles typographiques etc. Les 256 premiers caractères de UNICODE correspondent à ceux de la page de code ISO 8859-1. UNICODE va plus loin que les formats précédents en prenant en compte le fait que les processeurs ne gèrent pas les entiers de la même manière. En effet, certains processeurs (comme ceux d'*Intel*) travaillent d'abord avec les huit bits les moins significatifs (on dit qu'ils sont *little-endian*) alors que d'autres travaillent d'abord avec les huit bits les plus significatifs (on dit qu'ils sont *big-endian*). Un flot de données étant une suite de bits, il faut transformer certains octets pour certains processeurs. Cette transformation se nomme *UTF* (*Universal Transformation Format*). Il existe plusieurs normes UTF, UTF-7, UTF-8 et UTF-16.

La norme *UTF-8* est la plus utilisée à l'heure actuelle. Elle a la particularité d'avoir un nombre variable d'octets pour représenter les caractères UNICODE. Ce nombre d'octets peut aller de 1 à 5. Les caractères représentés par un octet sont les 128 caractères de la norme ASCII 7 bits. Nos caractères accentués ne faisant pas partie de ces 128 premiers caractères ASCII, ils sont donc

codés sur plus d'un octet. Si un éditeur de texte supporte l'encodage UTF-8, il reproduit correctement les caractères accentués. Sinon, plusieurs caractères étranges remplacent chaque caractère accentué. Vous avez sûrement déjà rencontré ce problème, notamment dans la lecture de vos mails.

Lors du traitement d'un flot de données contenant des caractères avec une des classes `StreamWriter` ou `StreamReader`, on a la possibilité de préciser le format d'encodage des caractères avec une des classes suivantes :

```
System.Object
   System.Text.Encoding
      System.Text.ASCIIEncoding
      System.Text.UnicodeEncoding
      System.Text.UTF7Encoding
      System.Text.UTF8Encoding
```

La classe `Encoding` contient des méthodes permettant de convertir une chaîne de caractères d'un format à un autre.

# *Lecture/écriture des données d'un fichier*

## *Simple lecture/écriture dans un fichier*

La classe `File` présente six méthodes permettant de lire ou d'écrire dans un fichier très simplement :

- Les méthodes `ReadAllText()` et `WriteAllText()` permettent d'écrire ou de lire une chaîne de caractères à partir de ou dans un fichier.
- Les méthodes `ReadAllLines()` et `WriteAllLines()` permettent d'écrire ou de lire un tableau de chaînes de caractères à partir de ou dans un fichier.
- Les méthodes `ReadAllBytes()` et `WriteAllBytes()` permettent d'écrire ou de lire un tableau de d'octets à partir de ou dans un fichier.

Les méthodes d'écriture créent le fichier si celui-ci n'existe pas et écrase les anciennes données si le fichier existe. En outre toutes ces méthodes ferment proprement le fichier utilisé. L'exemple suivant montre comment dupliquer un fichier texte tout en l'affichant à l'écran :

*Exemple 17-1 :*

```
using System.IO ;
public class Program {
   public static void Main() {
      string text = File.ReadAllText(@"G:\source\Test.txt") ;
      System.Console.Write(text) ;
      File.WriteAllText(@"G:\source\TestCopy.txt",text) ;
   }
}
```

## *Lecture/écriture de données binaires dans un fichier*

Nous montrons ici comment dupliquer un fichier binaire grâce aux flots de données. Voici les étapes à suivre :

- Créer un flot de données sortant du fichier source, instance de la classe `System.IO.FileStream`.
- Créer un flot de données entrant dans le fichier destination, instance de la classe `FileStream`.
- Faire transiter le contenu du fichier source vers le fichier destination. Comme vous n'avez pas à interpréter ce contenu, vous n'avez pas à utiliser les classes de typages d'un flot de données.

Pour faire transiter les données entre un flot de données entrant et un flot de données sortant, on utilise en général une *mémoire tampon* (*buffer* en anglais). Une mémoire tampon est une zone de mémoire de taille fixe, qui est alternativement remplie par le flot de données entrant puis vidée dans le flot de données sortant. Cette opération de remplissage/vidage est effectuée autant de fois qu'il le faut.

Le programme suivant copie le fichier exécutable de l'assemblage courant, vers le fichier `Copy.exe`, dans le répertoire de l'application. Naturellement l'assemblage courant ne doit pas s'appeler `Copy.exe`.

*Exemple 17-2 :*

```
using System.IO ;
public class Program {
   static readonly int tailleTampon = 512 ;
   public static void Main() {
      // Le nom du domaine d'application courant est égal...
      // ...au nom de l'assemblage courant (avec l'extension).
      string sExe = System.AppDomain.CurrentDomain.FriendlyName ;
      // Les deux fichiers sont dans le répertoire de l'application.
      FileStream inStream = File.OpenRead(sExe) ;
      FileStream outStream = File.OpenWrite("Copy.exe") ;
      // Construit une zone de mémoire tampon.
      byte[] tampon = new System.Byte[tailleTampon] ;
      int nBytesRead = 0 ;
      // Copie le fichier binaire.
      while ((nBytesRead = inStream.Read(tampon, 0, tailleTampon)) > 0)
         outStream.Write(tampon, 0, nBytesRead) ;
      // Ferme les flots de données.
      inStream.Close() ;
      outStream.Close() ;
   }
}
```

## *Lecture/écriture d'un fichier texte*

Nous présentons ici la copie d'un fichier texte en utilisant les classes `System.IO.StreamWriter` et `System.IO.StreamReader`. Nous allons copier ligne après ligne notre fichier texte. La longueur d'une ligne étant de taille variable non bornée, on ne peut utiliser une zone de mémoire

tampon de taille fixe. À chaque copie de ligne nous utilisons une nouvelle chaîne de caractères construite par la méthode `ReadLine()` de la classe `StreamReader`. Nous utilisons ensuite la méthode `WriteLine()` de `StreamWriter` pour copier cette chaîne dans le fichier destination.

Chaque ligne est affichée sur la console au moyen de la méthode `WriteLine()` de la classe `Console`. Une différence entre ces deux méthodes `WriteLine()` est que celle de la classe `Console` est statique alors que celle de la classe `StreamWriter` est d'instance.

*Exemple 17-3 :*

```
using System.IO ;
public class Program {
   public static void Main() {
      StreamReader inStream = File.OpenText(@"G:\source\Test.txt") ;
      StreamWriter outStream = new
                     StreamWriter(@"G:\source\TestCopy.txt") ;
      // Copie la source vers la destination.
      string sTmp ;
      do{
         sTmp = inStream.ReadLine() ;
         outStream.WriteLine(sTmp) ;
         System.Console.WriteLine(sTmp) ;
      }
      while (sTmp != null) ;
      // Ferme les flots de données.
      inStream.Close() ;
      outStream.Close() ;
   }
}
```

La notion d'encodage de caractères semble totalement absente de cet exemple, et pourtant cet exemple marche quel que soit le format d'encodage du fichier texte source. En fait, si l'utilisateur ne spécifie pas d'encodage particulier à la classe `StreamReader`, cette classe détecte automatiquement le format à utiliser en analysant les premiers caractères du fichier texte. Chaque type d'encodage prévoit de laisser une marque spéciale sur les premiers octets d'un fichier texte. Vous pouvez obtenir cette marque avec la méthode `byte[] GetPreamble()` de la classe présentée par chaque classe d'encodage dérivée de la classe `Encoding`. Par exemple cette marque est les trois octets `0xEF 0xBB 0xBF'` pour l'encodage UTF8, et `0xFF 0xFE` pour l'encodage UNICODE et rien pour l'encodage ASCII. Cette marque n'est pas forcément présente dans tous les fichiers textes et vous avez la possibilité de désactiver l'émission de cette marque lorsque vous écrivez dans un fichier texte au moyen de `StreamWriter`. Dans ce cas, l'implémentation *Microsoft* utilise par défaut l'encodage UTF-8.

La propriété `Encoding CurrentEncoding{get;}` de la classe `StreamReader`, vous permet de connaître l'encodage utilisé. Cette propriété est automatiquement positionnée à la première lecture d'un fichier texte. De même la classe `StreamWriter` présente la propriété `Encoding Encoding` accessible en lecture seule.

## *Traitement asynchrone d'un flot de données*

Le traitement d'un flot de données dépend linéairement de la taille de la source de données. Le développeur d'une application qui traite un ou plusieurs flots de données peut avoir un ordre

de grandeur de cette taille, mais en général, il ne la connaît pas. Le temps du traitement d'un flot de données est donc inconnu du développeur. Si un tel traitement est effectué par le thread principal d'une application, le résultat peut être catastrophique puisque l'application peut potentiellement se bloquer pendant un temps inconnu. De plus si l'application a de nombreux flots de données à traiter d'une manière indépendante, il serait judicieux qu'elles puissent effectuer ces traitements en parallèle.

Le traitement asynchrone d'un flot de données résout ces problèmes. Il permet de déléguer le traitement sur un autre thread que celui qui décide de lancer le traitement. Le développeur n'a absolument pas besoin de s'occuper de cet autre thread. Ce dernier fait partie du *pool de threads* d'entrée/sortie du CLR. Comme nous allons le voir dans le prochain exemple, il est aisé de lancer en parallèle plusieurs traitements asynchrones d'un flot de données. Une telle architecture pour vos applications peut être plusieurs dizaines de fois plus performante que le simple traitement séquentiel.

L'exemple suivant illustre le lancement en parallèle de `N` traitements asynchrones. Tout d'abord, nous créons un fichier volumineux d'environ 10 Mo avec la méthode `CreateBlobFile()`. Ensuite `nTraitements` lectures de ce fichier sont lancées en parallèle avec l'appel à la méthode `BeginRead()` de `FileStream`, dans une boucle. Les threads affectés à ces traitements sont ceux du pool de threads. La méthode `BeginRead()` permet de préciser une *procédure de finalisation* qui est appelée lorsque la lecture est finie. C'est le même thread qui effectue la lecture et qui exécute la procédure de finalisation. Dans l'exemple, la méthode `OnLectureFinie()` constitue la procédure de finalisation. L'exemple montre comment cette méthode récupère les données lues et éventuellement des informations provenant du thread principal (comme l'ID du traitement) grâce à la classe `EtatFichier`.

Contrairement aux sections précédentes, la zone de mémoire tampon est égale à la taille du fichier à lire. Nous n'avons donc pas à nous occuper de la gestion du tampon. En fait la classe `FileStream` gère son propre mécanisme de tampon en interne d'une manière transparente pour le développeur. Le développeur peut optionnellement fixer la taille de cette mémoire tampon interne dans un argument du constructeur (ici nous spécifions 4096 octets).

*Exemple 17-4 :*

```
using System ;
using System.IO ;
using System.Threading ;
public class Program {
   class EtatFichier {
      public byte []      tampon ;
      public FileStream   fs ;
      public int          idTraitement ;
   }
   // Taille du fichier : presque 10Mo.
   static readonly int   nOctetsParFichier  = 10000000 ;
   static readonly int   nTraitements = 5 ;
   static string         sFileName = "blob.bin" ;
   // Crée un fichier de nOctetsParFichier  octets.
   static void CreateBlobFile(){
      byte [] Blob = new byte[nOctetsParFichier ] ;
      FileStream fs = new FileStream(
```

```
            sFileName,
            FileMode.Create,FileAccess.Write,FileShare.None,
            4096, false) ;
         fs.Write(Blob, 0, nOctetsParFichier ) ;
         fs.Flush() ;
         fs.Close() ;
      }
      public static void Main() {
         CreateBlobFile() ;
         for(int i = 0 ; i< nTraitements ; i++ ){
            FileStream fs = new FileStream(
               sFileName,
               FileMode.Open,FileAccess.Read,FileShare.Read,
               // Taille du tampon interne.
               4096,
               true) ;
            // Initialise les données à communiquer à la procédure
            // de finalisation.
            EtatFichier etat = new EtatFichier() ;
            etat.idTraitement = i ;
            etat.fs = fs ;
            etat.tampon = new Byte[nOctetsParFichier ] ;
            Console.WriteLine(
                "Thread #{0} : Lance lecture asynchrone #{1}",
                Thread.CurrentThread.ManagedThreadId ,i) ;
            // Lance la lecture asynchrone de données.
            fs.BeginRead(
               etat.tampon,
               0,nOctetsParFichier ,
               new AsyncCallback(OnLectureFinie),
               etat);
         }
         // Pour ne pas compliquer, nous n'implémentons pas de mécanisme
         // de synchronisation pour attendre la fin des traitements
         // qui sont réalisés sur des threads background.
         Thread.Sleep(10000) ;
      }
      // La procédure de finalisation effectue
      // un traitement sur les données lues.
      static void OnLectureFinie(IAsyncResult asyncResult) {
         // Récupère les données pour le traitement.
         EtatFichier etat = (EtatFichier)asyncResult .AsyncState ;
         byte[] data = etat.tampon ;
         FileStream fs = etat.fs ;
         // Finalise le flot de données qui a servi à lire les données.
         int nOctetsLu = fs.EndRead(asyncResult ) ;
         fs.Close() ;
         Console.WriteLine(
            "Thread #{0} : Début traitement #{1}",
```

```
            Thread.CurrentThread.ManagedThreadId ,etat.idTraitement) ;
        // Simule un traitement des données d'une seconde.
        Thread.Sleep(1000) ;
        Console.WriteLine(
            "Thread #{0} : Fin traitement #{1}",
            Thread.CurrentThread.ManagedThreadId ,etat.idTraitement) ;
    }
}
```

Cet exemple affiche ceci sur ma machine. On voit bien que le thread principal a pour ID #8 et que les deux threads qui ont pour IDs #6 et #10, sont impliqués dans le traitement asynchrone. Ce comportement peut varier d'une exécution à l'autre et il peut y avoir entre 1 et `min` (`Ntraitement`, Nombre de threads IO dans le pool) threads affectés au traitement de données.

```
Thread #8 : Lance lecture asynchrone #0
Thread #8 : Lance lecture asynchrone #1
Thread #6 : Début traitement #0
Thread #8 : Lance lecture asynchrone #2
Thread #8 : Lance lecture asynchrone #3
Thread #8 : Lance lecture asynchrone #4
Thread #6 : Fin traitement #0
Thread #6 : Début traitement #1
Thread #10 : Début traitement #2
Thread #6 : Fin traitement #1
Thread #6 : Début traitement #3
Thread #10 : Fin traitement #2
Thread #6 : Fin traitement #3
Thread #10 : Début traitement #4
Thread #10 : Fin traitement #4
```

# *Support du protocole TCP/IP et des sockets*

## *Introduction aux sockets et au protocole TCP/IP*

Nous montrons dans cette section que la manipulation des flots de données distants, c'est-à-dire entre machines distantes et par l'intermédiaire d'un réseau, est très similaire à ce que nous avons présenté plus haut. Les instances de la classe `System.Net.Sockets.Socket` représentent des *sockets*. La présentation détaillée des sockets et du protocole TCP/IP dépasse le cadre de cet ouvrage, mais nous allons rappeler les points importants concernant ce sujet.

Le protocole *IP* (*Internet Protocol*) propose un modèle d'adressage d'une machine. Le protocole IP est aujourd'hui globalement répandu. Bien que ce soit une vision simpliste, on peut dire qu'une *adresse IP* identifie une machine (ou un groupe de machines) sur le réseau mondial *Internet*. Une adresse IP est un nombre codé sur quatre octets. En plus d'avoir une adresse IP, chaque machine connectée au réseau Internet peut présenter jusqu'à 65536 *ports*. Un port est un nombre codé sur deux octets qui représente un point de communication sur une machine. Chaque machine peut donc présenter plusieurs points de communication, et un couple *adresse IP/ numéro de port* représente un point de communication unique sur le réseau internet. Pour l'utilisateur du protocole IP, tout se passe comme s'il y avait effectivement jusqu'à 65536 flots de

données entrants et sortants. Ce sont les couches basses réseau, qui modulent et démodulent le flot physique de bits entrant et sortant du câble réseau. Ces couches sont conceptuellement en dessous du protocole IP.

Une socket matérialise un point de communication sur une machine. Les sockets étaient à la base, il y a 20 ans, un ensemble de fonctions bas niveau, écrites en C. Elles ont été inventées à l'université de *Berkeley* en Californie, pour répondre au besoin de communication inter machines. L'ensemble de ces fonctions est devenu standard et a été encapsulé dans des classes. Les sockets supportent plusieurs protocoles de communication. Notamment le protocole *UDP/IP* (*User Datagram Protocol*) et surtout le protocole *TCP/IP* (*Transfert Control Protocol*) qui comme son nom l'indique, utilise le protocole IP et son mode d'adressage.

Concrètement, un processus, nommé serveur, crée une socket et appelle une méthode bloquante (en général nommée `Accept()`) sur cet objet, pour attendre une requête. Un autre processus crée une autre socket. Ce processus est nommé client. Il existe sur la même machine ou sur une autre machine que celle qui contient le processus serveur. Le client fournit à sa socket le couple adresse IP/Port identifiant la socket du serveur. Grâce à cette information, la socket du client se connecte à la socket serveur, ce qui a pour effet de débloquer le serveur de son attente sur la méthode `Accept()` et de créer un flot de données entre le client et le serveur. À partir de cette étape, deux scénarios peuvent se produire sur le serveur :

- Soit le serveur traite complètement les besoins du client avant de se remettre en mode d'attente client en appelant `Accept()`. On dit que le serveur travaille d'une manière synchrone, car il sert les clients les uns après les autres.
- Soit le serveur crée le plus rapidement possible une nouvelle socket qu'il passe à un autre thread pour qu'il traite les besoins du client. Le serveur principal rappelle donc `Accept()` presque immédiatement après la connexion client. On dit que le serveur travaille d'une manière asynchrone.

Bien entendu, l'utilisation du mode asynchrone est en général beaucoup plus performante, puisqu'elle minimise les temps d'attente globaux des clients. Nous allons présenter ces deux façons de travailler dans les deux prochaines sections.

## *Utilisation synchrone des sockets*

Nous exposons ici un exemple d'utilisation synchrone des sockets, avec le code du serveur et le code du client. Ne vous étonnez pas de ne pas y voir la moindre trace de l'utilisation de la classe `Socket` ! Le *framework* .NET présente les deux classes `System.Net.Sockets.TcpListener` et `System.Net.Sockets.TcpClient` qui utilisent la classe `Socket` en interne. L'utilisation de ces classes simplifie grandement la programmation socket avec le protocole TCP/IP. Pour les lecteurs qui connaissent déjà bien la programmation avec sockets, sachez que l'utilisation de ces classes reste assez similaire. De plus rien ne vous empêche d'utiliser la classe `Socket` si vous préférez.

### *Côté serveur*

Notre serveur crée un point de communication sur le port 50000 avec une instance de la classe `TcpListener`. Le serveur attend la connexion d'un client en appelant la méthode `AcceptTcpClient()`. Lorsqu'un client se connecte, le serveur obtient une instance de la classe `System.Net.Sockets.NetworkStream` qui représente un flot de données avec le client. Dans la méthode `TraiteClient()`, le serveur ouvre un fichier texte et le copie, ligne après ligne, dans le flot de

données vers le client. Une fois le fichier copié, le serveur ferme le flot de données vers le client et le flot de données en provenance du fichier et retourne en attente d'un client.

Notez que les opérations de fermeture de flot ont lieu au sein d'une clause `finally`. En effet, ce sont typiquement des opérations de libération de ressources critiques. Nous aurions pu opter plutôt pour l'utilisation du *pattern using/Dispose()* puisque toutes les classes de flots de données implémentent l'interface `IDisposable`.

*Exemple 17-5 : Le code du serveur : Serveur.cs*

```
using System ;
using System.Net.Sockets ;
using System.Net ;
using System.IO ;
class ProgServeur {
   static readonly ushort port = 50000 ;
   static void Main() {
      IPAddress ipAddress = new IPAddress(new byte[] { 127, 0, 0, 1 }) ;
      TcpListener tcpL = new TcpListener(ipAddress,port) ;
      tcpL.Start() ;
      // Chaque passage dans la boucle = envoi d'un fichier à un client.
      while(true){
         try{
            Console.WriteLine( "Attente d'un client..." ) ;
            // L'appel à cette méthode est bloquant, càd, on ne sort
            // de cette méthode que lorsqu'un client est connecté.
            TcpClient tcpC = tcpL.AcceptTcpClient() ;
            Console.WriteLine( "Client connecté." ) ;
            TraiteClient( tcpC.GetStream() ) ;
            Console.WriteLine( "Client déconnecté." ) ;
         }
         catch(Exception e){
            Console.WriteLine(e.Message) ;
         }
      }
   }
   static void TraiteClient (NetworkStream networkStream){
      // Flot de données d'envoi sur le réseau.
      StreamWriter  streamWriter = new StreamWriter(networkStream) ;
      // Flot de données de lecture du fichier source.
      StreamReader  streamReader = new StreamReader(
                                 @"C:/Text/Fichier.txt" ) ;

      // Pour chaque ligne du fichier source : envoi sur le réseau.
      string sTmp = streamReader.ReadLine() ;
      try{
         while(sTmp != null ){
            Console.WriteLine( "Envoi de : {0}" , sTmp ) ;
            streamWriter.WriteLine(sTmp);
            streamWriter.Flush();
```

```
            sTmp = streamReader.ReadLine();
         }
      }
      finally {
         // Fermeture des flots de données
         streamReader.Close() ;
         streamWriter.Close() ;
         networkStream.Close() ;
      }
   }
}
```

Voici l'affichage du serveur lorsqu'un client se connecte :

*Exemple :* *Affichage du serveur*

```
Attente d'un client...
Client connecté.
Envoi de : ####################
Envoi de : Voici le contenu...
Envoi de : ...de Fichier.txt
Envoi de : ####################
Client déconnecté.
Attente d'un client...
```

## *Côté client*

Le code du client est encore plus simple que celui du serveur. Le client crée une instance de la classe `TcpClient` et lui fournit le couple adresse IP/Port du serveur. Lorsque le client est connecté, il obtient une instance de la classe `System.Net.Sockets.NetworkStream` qui représente un flot de données avec le serveur. Le client n'a plus qu'à lire les données à partir de ce flot. Nous savons que le serveur envoie des lignes de caractères. Aussi, le client lit les lignes les unes après les autres, et les affiche sur sa console.

*Exemple 17-6 :* *Le code du client : Client.cs*

```
using System ;
using System.Net.Sockets ;
using System.IO ;

class ProgClient {
   static readonly string host = "localhost" ;
   static readonly ushort port = 50000 ;
   static void Main() {
      TcpClient tcpC ;
      try{
         // L'appel au constructeur de TcpClient envoie une
         // exception si la connexion avec le serveur échoue.
         tcpC = new TcpClient(host,port);
         Console.WriteLine(
               "Connexion établie avec {0}:{1}" ,host , port) ;
```

```
            NetworkStream networkStream = tcpC.GetStream();
            StreamReader streamReader = new StreamReader(networkStream) ;
            try{
               // Chaque passage dans la boucle = une ligne récupérée.
               string sTmp = streamReader.ReadLine();
               while( sTmp != null ){
                  Console.WriteLine( "Réception de : {0}" ,sTmp) ;
                  sTmp = streamReader.ReadLine();
               }
            }
            finally{
               // Ferme les flots de données.
               streamReader.Close() ;
               networkStream.Close() ;
               Console.WriteLine( "Déconnexion de {0}:{1}" ,host , port ) ;
            }
         }
         catch(Exception e){
            Console.WriteLine(e.Message) ;
            return ;
         }
      }
   }
```

Voici l'affichage du client :

*Exemple :* *Affichage du client*

```
Connexion établie avec localhost:50000
Réception de : ####################
Réception de : Voici le contenu...
Réception de : ...de Fichier.txt
Réception de : ####################
Déconnexion de localhost :50000
```

L'adresse IP est ici `"localhost"` car nous supposons que le serveur et le client sont sur la même machine. Dans un cas réel, l'adresse IP du serveur serait un paramètre du programme client, sûrement placé dans le fichier de configuration de l'application ou dans les arguments en ligne de commande. Sachez qu'à la place d'une adresse IP vous pouvez fournir un nom de domaine comme `www.smacchia.com`. La classe `System.Net.Dns` permet de transformer un nom de domaine en une liste d'adresses IP et d'obtenir un nom de domaine à partir d'une adresse IP. Nous n'utilisons pas cette classe ici.

## *Utilisation asynchrone des sockets*

### *Côté serveur*

À l'instar de la section précédente, nous n'utilisons pas la classe `Socket` mais seulement les classes `TcpListener` et `TcpClient`. Toute l'implémentation du modèle asynchrone tient dans le fait que dans le code du serveur, nous utilisons les méthodes `BeginRead()` et `BeginWrite()`. Comme nous l'avons vu un peu plus haut lors du traitement asynchrone d'un flot de données

avec un fichier, les méthodes `BeginRead()` et `BeginWrite()` permettent de fournir une *procédure de finalisation*. Cette procédure est automatiquement appelée à la fin de la lecture ou de l'écriture des données. Cette procédure est appelée par le même thread qui a effectué la lecture ou l'écriture. Vous n'avez pas à vous occupez de ce thread. Ce dernier fait partie du *pool de threads* d'entrée/sortie du CLR.

Contrairement à l'exemple du traitement asynchrone d'un flot de données avec un fichier (Exemple 17-4), nous n'utilisons pas la possibilité de partager un objet entre le thread principal et le thread du pool effectuant les entrée/sorties. En général, l'architecture qui permet le traitement asynchrone des connexions clientes distantes contient une classe dont les instances traitent les clients (une instance de cette classe par client). Nous avons nommé cette classe `TraitementClient`. Une instance de cette classe n'a besoin que du flot de données vers le client. Ce flot de données est obtenu de la même manière que dans la section précédente, en appelant successivement les méthodes `AcceptTcpClient()` puis `GetStream()`.

Notre exemple est construit de manière à ce que le serveur commence par lire les données du client. Une fois réceptionnées, le serveur renvoie ces données au client. Le serveur effectue ces deux opérations jusqu'à ce que le client ne lui envoie plus de données. À ce moment, le serveur arrête de traiter ce client. Bien entendu, dans un cas réel, le serveur ne renverrait pas les données directement au client sans traitements, mais nous avons tenté de simplifier au maximum l'exemple.

*Exemple 17-7 : Code du serveur :Serveur.cs*

```
using System ;
using System.Net.Sockets ;
using System.Net ;
using System.IO ;
using System.Text ;

class ProgServeur {
   static readonly ushort port = 50000 ;
   static void Main() {
      IPAddress ipAddress = new IPAddress(new byte[] { 127, 0, 0, 1 }) ;
      TcpListener tcpL = new TcpListener(ipAddress,port);
      tcpL.Start() ;
      while(true){
         try{
            Console.WriteLine( "Main:Attente d'un client..." ) ;
            TcpClient tcpC = tcpL.AcceptTcpClient();
            Console.WriteLine( "Main:Client connecté." ) ;
            TraitementClient tc =new TraitementClient(tcpC.GetStream()) ;
            tc.Traite() ;
         }
         catch(Exception e){
            Console.WriteLine(e.Message) ;
         }
      }
   }
}
// Une instance de cette classe est créée pour chaque connexion client.
```

```
class TraitementClient{
   static readonly int      tailleTampon = 512 ;
   private byte []            m_Tampon ;
   private NetworkStream      m_NetworkStream ;
   private AsyncCallback      m_ProcLecture ;
   private AsyncCallback      m_ProcEcriture ;
   // Le constructeur initialise :
   //   - m_NetworkStream :le flot de données avec le client.
   //   - m_ProcLecture : la procédure de finalisation d'une lecture.
   //   - m_ProcEcriture : la procédure de finalisation d'une écriture.
   //   - m_Tampon : la zone de mémoire tampon utilisée pour la lecture
   //                et l'écriture.

   public TraitementClient(NetworkStream networkStream){
      m_NetworkStream  = networkStream ;
      m_ProcLecture    = new AsyncCallback(this.OnLectureFini) ;
      m_ProcEcriture   = new AsyncCallback(this.OnEcritureFini) ;
      m_Tampon         = new byte[tailleTampon] ;
   }

   public void Traite(){
      m_NetworkStream.BeginRead(
         m_Tampon, 0 , m_Tampon.Length , m_ProcLecture , null ) ;
   }
   // Cette méthode est la procédure de finalisation appelée lorsqu'une
   // lecture en mode asynchrone déclenchée par l'appel à la méthode
   // BeginRead() se termine.

   private void OnLectureFini( IAsyncResult asyncResult  ){
      int nOctets = m_NetworkStream.EndRead(asyncResult ) ;
      // Renvoie au client, la chaine reçue.
      if( nOctets > 0 ){
         string s = Encoding.ASCII.GetString(m_Tampon,0,nOctets) ;
         Console.Write(
           "Async:{0} octets recus du client : {1}" , nOctets, s ) ;
         m_NetworkStream.BeginWrite(
            m_Tampon, 0 , nOctets , m_ProcEcriture , null ) ;
      }
      // Le client n'a rien renvoyé donc on ne s'occupe plus de lui.
      else{
         Console.WriteLine( "Async:Traitement client fini." ) ;
         m_NetworkStream.Close() ;
         m_NetworkStream = null ;
      }
   }
   // Cette méthode est la procédure de finalisation appelée lorsqu'une
   // écriture en mode asynchrone déclenchée par l'appel à la méthode
   // BeginWrite() se termine.
```

```
   private void OnEcritureFini( IAsyncResult asyncResult  ){
      m_NetworkStream.EndWrite(asyncResult ) ;
      Console.WriteLine( "Async:Ecriture finie." ) ;
      m_NetworkStream.BeginRead(
         m_Tampon, 0 , m_Tampon.Length , m_ProcLecture , null ) ;
   }
}
```

Voici l'affichage du serveur lorsqu'il est connecté à un client dont le code est présenté ci-après. Notez que le serveur se remet en attente d'un autre client avant même de recevoir des données du premier client.

*Exemple :* *Affichage du serveur*

```
Main:Attente d'un client...
Main:Client connecté.
Main:Attente d'un client...
Async:33 octets recus du client : Vous avez le bonjour du client !
Async:Ecriture finie.
Async:33 octets recus du client : Vous avez le bonjour du client !
Async:Ecriture finie.
Async:33 octets recus du client : Vous avez le bonjour du client !
Async:Ecriture finie.
Async:Traitement client fini.
```

## *Côté client*

Le fait que le serveur travaille en mode asynchrone n'a aucun impact sur le code du client. Le client est ainsi très similaire au client présenté dans la section précédente. La différence est qu'ici le client ne se contente pas d'attendre des données en provenance du serveur. Il envoie des données et reçoit des données trois fois successivement. Voici le code :

*Exemple 17-8 :* *Code du Client : Client.cs*

```
using System ;
using System.Net.Sockets ;
using System.IO ;

class ProgClient {
   static readonly string host = "localhost" ;
   static readonly ushort port = 50000 ;
   static void Main() {
      TcpClient tcpC ;
      try{
         // L'appel au constructeur de TcpClient envoie une
         // exception si la connexion avec le serveur échoue.
         tcpC = new TcpClient(host,port);
         Console.WriteLine(
            "Connexion établie avec {0}:{1}",host,port) ;

         // Initialise le flot de données vers le serveur
```

```
            // accessible en écriture et en lecture.
            NetworkStream networkStream = tcpC.GetStream();
            StreamWriter streamWriter = new StreamWriter(networkStream) ;
            StreamReader streamReader =new StreamReader(networkStream) ;
            try{
               string sEnvoie = "Vous avez le bonjour du client !" ;
               string sRecu ;
               for(int i=0;i<3;i++)
               {
                  // Envoi de données au serveur.
                  Console.WriteLine( "Client  -> Serveur:" + sEnvoie ) ;
                  streamWriter.WriteLine(sEnvoie) ;
                  streamWriter.Flush() ;
                  // Réception de données du serveur.
                  sRecu = streamReader.ReadLine() ;
                  Console.WriteLine( "Serveur -> Client :" + sRecu ) ;
               }
            }
            finally{
               streamWriter.Close() ;
               streamReader.Close() ;
               networkStream.Close() ;
            }
         }
         catch(Exception e){
            Console.WriteLine(e.Message) ;
            return ;
         }
      }
   }
```

Voici l'affichage du client :

*Exemple :* *Affichage du client*

```
Connexion établie avec localhost:50000
Client  -> Serveur:Vous avez le bonjour du client !
Serveur -> Client :Vous avez le bonjour du client !
Client  -> Serveur:Vous avez le bonjour du client !
Serveur -> Client :Vous avez le bonjour du client !
Client  -> Serveur:Vous avez le bonjour du client !
Serveur -> Client :Vous avez le bonjour du client !
```

# *Obtenir des informations concernant le réseau*

## *Découvrir les interfaces réseaux disponibles*

Les instances de la classe `System.Net.NetworkInformation.NetworkInterface` représentent une interface réseau. La méthode statique `GetAllNetworkInterfaces()` de cette classe permet de lister toutes les interfaces réseaux disponibles sur la machine. Par exemple :

*Exemple 17-9 :*

```
using System ;
using System.Net.NetworkInformation ;
public class Program {
   public static void Main() {
      NetworkInterface[] nis=NetworkInterface.GetAllNetworkInterfaces();
      foreach (NetworkInterface ni in nis)      {
         Console.WriteLine("Nom : " + ni.Name) ;
         Console.WriteLine("Description : " + ni.Description) ;
         Console.WriteLine("Etat : " + ni.OperationalStatus.ToString()) ;
         Console.WriteLine("Vitesse : {0} Kb", ni.Speed / 1024) ;
         Console.WriteLine("Media Acess Control (MAC):" +
                           ni.GetPhysicalAddress().ToString()) ;
         Console.WriteLine("--------------------------------") ;
      }
   }
}
```

## *Ping*

Vous pouvez utiliser la classe `System.Net.NetworkInformation.Ping` pour déterminer si une machine distante est accessible par le réseau. En fait cette classe est l'équivalente de la commande `Ping`.

*Exemple 17-10 :*

```
using System ;
using System.Net.NetworkInformation ;
public class Program {
   public static void Main() {
      using (Ping ping = new Ping()) {
         PingReply pingReply = ping.Send("www.smacchia.com", 10000) ;
         System.Console.WriteLine("IP:{0} Etat:{1}",
                           pingReply.Address , pingReply.Status) ;
      }
   }
}
```

Cette classe implémente l'interface `IDispose`. En outre la méthode `Ping.SendAsync()` vous permet de démarrer un ping d'une manière asynchrone. Il faut alors se brancher à l'événement `Ping.PingCompleted` pour obtenir le résultat du ping.

## *Etre averti d'un changement réseau*

L'adresse IP affectée à une interface réseau peut changer dans le temps pour de multiples raisons. Vous pouvez être averti le cas échéant en vous abonnant à l'événement `NetworkAddressChanged` de la classe `System.Net.NetworkInformation.NetworkChange`.

## *Statistiques réseau*

Vous pouvez programmatiquement avoir accès aux statistiques des protocoles IP, ICMP, TCP, et UDP. Ces statistiques sont les mêmes que celles obtenues avec les différentes options de la commande `netstat`. Par exemple :

*Exemple 17-11 :*

```
using System ;
using System.Net.NetworkInformation ;
public class Program {
   public static void Main() {
      IPGlobalProperties ipProp =
         IPGlobalProperties.GetIPGlobalProperties() ;
      IPGlobalStatistics ipStat = ipProp.GetIPv4GlobalStatistics();
      Console.WriteLine("Host name:" + ipProp.HostName) ;
      Console.WriteLine("Domain name:" + ipProp.DomainName) ;
      Console.WriteLine("IPv4 # packets recus:" +
                        ipStat.ReceivedPackets) ;
      Console.WriteLine("IPv4 # packets envoyés:" +
                        ipStat.OutputPacketRequests) ;
      TcpConnectionInformation[] tcpConns =
                                    ipProp.GetActiveTcpConnections() ;
      foreach (TcpConnectionInformation tcpConn in tcpConns) {
         Console.WriteLine("localhost:{0} <-> {1}:{2} state:{3}",
                           tcpConn.LocalEndPoint.Port,
                           tcpConn.RemoteEndPoint.Address.ToString(),
                           tcpConn.RemoteEndPoint.Port, tcpConn.State) ;
      }
   }
}
```

Vous pouvez n'obtenir que les statistiques relatives à une interface réseau en utilisant la méthode `NetworkInterface.GetIPInterfaceProperties()`.

# *Clients HTTP et FTP*

## *La notion d'URI*

Un *URI* (*Universal Ressource Identifier*) est une chaîne de caractères permettant de localiser une ressource. Un URI est formé de trois parties ;

- Les premiers caractères d'un URI représentent le *mode d'accès* de l'URI (*scheme* en anglais). Voici les modes d'accès les plus courants :
  - Le mode d'accès `file` indique que la ressource est un fichier en local ou sur un intranet.
  - Le mode d'accès `http` indique que la ressource est gérée par un serveur web. Le mode d'accès `https` indique qu'il faut utilisé le protocole HTTP sécurisé (i.e HTTP au dessus du protocole SSL décrit un peu plus bas page 661).
  - Le mode d'accès `mailto` indique que la ressource est une adresse e-mail.

  - Le mode d'accès `ftp` indique que la ressource est un fichier géré par un serveur FTP (*File Transfert Protocol*) (même remarque pour `ftps` que pour `https`).

- Le nom du serveur.
- Le nom de la ressource.

Voici quelques exemples d'URI :

```
http://www.smacchia.com/ConstructeursAutomobiles.html
ftp://ftp.lip6.fr/pub/gnu/a2ps/a2ps-4.10.4.tar.gz
file://localhost/
mailto:patrick@smacchia.com
```

À l'instar des chemins, un URI peut être relatif à un autre URI. Les *URI relatifs* n'ont donc pas de mode d'accès au début des chaînes de caractères qui les représentent. La classe `System.Uri` a été conçue pour stocker et manipuler les URI.

## *La classe WebClient*

La classe `System.Net.WebClient` permet d'écrire ou de récupérer des données à partir d'un URI. Ces données peuvent être stockées sur le système de fichier local, sur un intranet ou sur internet. La classe `WebClient` est très pratique puisque son implémentation décide automatique du protocole de transfert de données à utiliser en fonction de l'URI (`file`, `http`, `https`, `ftp` ou `ftps`). L'exemple suivant télécharge une page HTML à partir de son URI puis l'affiche au format texte sur la console :

*Exemple 17-12 :*

```
using System.Net ;
class Program {
   static void Main() {
      WebClient webClient = new WebClient() ;
      string s =
           webClient.DownloadString("http://www.microsoft.com/france");
      System.Console.Write(s) ;
   }
}
```

La classe `WebClient` présente des méthodes telles que `void DownloadFile(string uri,string fileName)` ou `byte[] DownloadData(string uri)` pour faciliter le téléchargement de données stockées dans des fichiers ou au format binaire. Chacune de ces méthodes a une homologue nommée `UploadXXX(string uri, data)` permettant d'écrire des données à un emplacement indiqué par un URI. Notez aussi la présence du couple de méthodes `Stream OpenWrite(string uri)` et `Stream OpenRead(string uri)` permettant de manipuler une ressource accédée par son URI au moyen d'un flot de données. Enfin, toutes ces méthodes (`DownloadXXX()` et `UpLoadXXX()`) sont disponibles en une version asynchrone dont le nom est suffixé par `Async`. Dans ce cas, l'opération est réalisée par un des threads du pool de threads du CLR. Vous pouvez être alors prévenu de la terminaison d'une opération asynchrone en vous abonnant à un événement tel que `UpdloadDataCompleted` de votre instance de la classe `WebClient`.

## *Autres classes pour manipuler les ressources à partir de leurs URI*

La classe `WebClient` est très pratique mais elle ne suffit pas lorsque vous devez exploiter certaines spécificités du protocole de transfert de données sous jacent (tel que les *cookies* du protocole HTTP). Les classes `System.Net.FileWebRequest`, `System.Net.FtpWebRequest` et `System.Net.HttpWebRequest` permettent de modéliser une requête de données en tenant compte du protocole de transfert. Elles dérivent toutes les trois de la classe `WebRequest`. Pour obtenir une instance d'une de ces classes il est conseillé d'invoquer la méthode statique `WebRequest WebRequest.Create(string uri)`. Vous pouvez alors personnaliser votre requête. Par exemple la classe `HttpWebRequest` présente la propriété `CookieContainer` permettant d'associer des cookies à votre requête.

Pour effectuer la requête il faut appeler la méthode `WebResponse GetResponse()` sur l'objet représentant votre requête. Cette méthode retourne une instance d'une des classes dérivées de `System.Net.WebResponse` à savoir `System.Net.FileWebResponse`, `System.Net.FtpWebResponse` et `System.Net.HttpWebResponse`. Vous pouvez alors obtenir les données en appelant la méthode d'instance `Stream WebResponse.GetResponseStream()`. Notez que certaines méthodes `BeginGetXXX()` et `EndGetXXX()` de la classe `WebRequest` permettent d'effectuer la requête d'une manière asynchrone. Enfin, sachez qu'en interne la classe `WebClient` utilise les classes que l'on vient de présenter.

L'exemple suivant affiche sur la console le contenu d'un fichier téléchargé sur un serveur FTP :

*Exemple 17-13 :*

```
using System.Net ;
using System.Text ;
using System.IO ;
class Program{
   static void Main(){
      FtpWebRequest ftpReq = WebRequest.Create(
         "ftp://smacchia.com/test.txt") as FtpWebRequest;
      ftpReq.Method = WebRequestMethods.Ftp.UploadFile;
      ftpReq.Credentials = new NetworkCredential( "login", "password" );
      FtpWebResponse ftpResp = ftpReq.GetResponse() as FtpWebResponse;
      StreamReader streamReader = new StreamReader(
                        ftpResp.GetResponseStream() , Encoding.ASCII ) ;
      System.Console.WriteLine( streamReader.ReadToEnd() ) ;
      streamReader.Close() ;
   }
}
```

Vous pouvez définir un cache de ressources. Pour cela, il vous suffit de créer une instance de la classe `System.Net.Cache.RequestCachePolicy` avec une valeur de l'énumération `System.Net.Cache.RequestCacheLevel` précisant le type de cache désiré. Vous pouvez alors positionner le cache au niveau global en utilisant la propriété statique `RequestCachePolicy WebRequest.DefaultCachePolicy{get;set;}` ou au niveau d'une requête en utilisant la propriété d'instance `RequestCachePolicy WebRequest.CachePolicy{get;set;}`. Plus d'information à ce sujet sont disponibles dans la section **Cache Management For Network Applications** des **MSDN**.

# Serveur HTTP avec HttpListener et HTTP.SYS

## La couche noyau HTTP.SYS

Les systèmes d'exploitation *Windows XP SP2* et *Windows Server 2003* contiennent une couche noyau nommée *HTTP.SYS* spécialisée dans le traitement du protocole HTTP. Jusqu'à cette innovation, les serveurs HTTP s'exécutant sur les OS *Windows* exploitaient la couche utilisateur *Winsock* (*Windows Socket API*) qui elle-même consommait les services de la couche noyau TCP/IP. Dans ce modèle, chaque requête/réponse HTTP requiert une transition coûteuse entre le mode noyau et le mode utilisateur. HTTP.SYS est une couche qui s'exécute entièrement en mode noyau. Notamment, HTTP.SYS prend en charge le cache de réponse. Ainsi, lorsque HTTP.SYS retourne une page directement à partir du cache il n'y a pas de transition coûteuse vers le mode utilisateur. Ce modèle est donc plus performant.

IIS (le serveur HTTP de *Windows*) exploite HTTP.SYS depuis la version 6.0. Cette couche sait gérer le routage d'une requête HTTP vers le processus adéquate notamment parce que IIS 6.0 lui communique ses informations de routage à chaque démarrage. HTTP.SYS sait aussi gérer le stockage des requêtes HTTP en attendant le démarrage du processus qui saura les traiter.

## La classe System.Net.HttpListener

Le *framework* .NET 2.0 présente la classe `System.Net.HttpListener` qui permet d'exploiter la couche HTTP.SYS pour développer un serveur HTTP. Bien entendu, vous ne pouvez pas utiliser cette classe lorsque votre application s'exécute sur un OS qui ne supporte pas HTTP.SYS. Le programme suivant montre comment utiliser cette classe afin de retourner une page HTML qui contient la date et l'heure courante ainsi que l'URL demandée. Une telle page HTML est retournée pour chaque requête HTTP entrante par le port 8008 à destination du chemin `/hello` :

*Exemple 17-14 :*

```
using System.Net ;
using System.IO ;
class Program {
   static void Main() {
      HttpListener httpListener = new HttpListener() ;
      string uri = string.Format("http://localhost:8008/hello/") ;
      httpListener.Prefixes.Add(uri);
      httpListener.Start();
      while (true) {
         // Ne pas effectuer la requête si le client
         HttpListenerContext ctx = httpListener.GetContext() ;
         ctx.Response.ContentType = "text/html" ;
         TextWriter writer = new StreamWriter(
           ctx.Response.OutputStream, System.Text.Encoding.Unicode) ;
         writer.WriteLine(
   "<html><body>Page demandée à {0}<br/>URL:{1}</body></html>",
           System.DateTime.Now, ctx.Request.Url) ;
         writer.Flush() ;
         writer.Close() ;
      }
```

```
    }
}
```

Le démarrage du serveur HTTP se fait à l'appel de la méthode `Start()`. Il faut au préalable enregistrer toutes les URLs que votre instance de `HttpListener` va traiter à l'aide de la propriété `Prefixes`. HTTP.SYS supporte le protocole HTTPS. En conséquence, vous pouvez préciser des URLs ayant le mode d'accès `https://`. Si une URL est déjà traitée par HTTP.SYS, une exception est levée lors de l'ajout. Une erreur classique est de ne pas préciser de numéro de port dans vos URLs. Le port 80 est alors pris par défaut mais en général ce port est déjà utilisé par le serveur IIS si celui-ci est présent sur la machine en production. Une exception est alors levée.

Une fois le serveur web démarré, on entre dans une boucle sans fin. Chaque passage dans la boucle représente le traitement d'une requête HTTP. La méthode `HttpListener.GetContext()` est bloquante jusqu'à la réception d'une requête. Le traitement de la requête est alors matérialisé par une instance de `System.Web.HttpListenerContext`. On construit alors notre page HTML que l'on inclut dans la réponse HTTP à l'aide du flot de données `Stream HttpListenerContext.Response.OutputStream`.

En plus des propriétés `Request` et `Response`, la classe `HttpListenerContext` présente la propriété `IPrincipal User{get;}` qui précise l'utilisateur *Windows* responsable de la requête si celui-ci a été authentifié. HTTP.SYS supporte les quatre modes d'authentification : Connexion Anonyme, Authentification Digest, Authentification de base et Authentification intégrée *Windows*. Ces modes sont décrits en page 957.

## *Traitement asynchrone des requêtes*

Dans l'Exemple 17-14, toutes les requêtes HTTP sont traitées les unes à la suite des autres par le même thread. Clairement cette façon de faire n'est pas efficace. Aussi, la classe `HttpListener` présente les deux méthodes `BeginGetContext()` et `EndGetContext()` qui permettent de traiter plusieurs requêtes simultanément sur les threads du pool du CLR. Voici notre exemple réécrit avec cette possibilité (notez qu'à chaque exécution de la boucle nous attendons une seconde pour éviter que le thread principal ne monopolise le processeur puisque la méthode `BeginGetContext()` n'est pas bloquante) :

*Exemple 17-15 :*

```
using System ;
using System.Net ;
using System.IO ;
class Program {
   static void Main() {
      HttpListener httpListener = new HttpListener() ;
      string uri = string.Format("http://localhost:8008/hello/") ;
      httpListener.Prefixes.Add(uri) ;
      httpListener.Start() ;
      while (true) {
         IAsyncResult result =
            httpListener.BeginGetContext(ProcessResponse, httpListener);
         System.Threading.Thread.Sleep(1000) ;
      }
   }
```

```
   private static void ProcessResponse(IAsyncResult result) {
      HttpListener httpListener = result.AsyncState as HttpListener;
      HttpListenerContext ctx = httpListener.EndGetContext(result);
      ctx.Response.ContentType = "text/html" ;
      TextWriter writer = new StreamWriter(
         ctx.Response.OutputStream, System.Text.Encoding.Unicode) ;
      writer.WriteLine(
   "<html><body>Page demandée à {0}<br/>URL:{1}</body></html>",
         System.DateTime.Now, ctx.Request.Url ) ;
      writer.Flush() ;
      writer.Close() ;
   }
}
```

# *Support des protocoles d'envoi de mails SMTP et MIME*

En .NET 2.0, les classes de l'espace de noms `System.Web.Mail` sont maintenant obsolètes. Pour envoyer des mails, il faut utiliser les classes de l'espace de noms `System.Net.Mail`. Notamment, une instance de la classe `SmtpClient` représente un point d'accès vers un serveur SMTP. Voici un exemple qui se sert de cette classe pour envoyer un mail :

*Exemple 17-16 :*

```
using System.Net ;
using System.Net.Mail ;
class Program {
   static void Main() {
      SmtpClient client = new SmtpClient("smtp.myserver.fr",25);
      client.Credentials = new
         NetworkCredential("patrick.smacchia@myserver.fr", "mypwd") ;
      client.Send("patrick@smacchia.com", "destinataire@xyz.com",
                  "my subject", "my body") ;
   }
}
```

La classe `SmtpClient` prévoit aussi une méthode `SendAsync()` et un événement `SendCompleted` pour envoyer un mail d'une manière asynchrone, sans bloquer le thread courant. Des facilités sont aussi proposées pour communiquer avec un serveur SMTP en SLL.

Une instance de la classe `MailMessage` représente un mail, une instance de la classe `MailAddress` représente une adresse mail et une instance de la classe `Attachment` représente un attachement à un mail. Reprenons notre programme avec ces classes pour envoyer notre mail en lui attachant le fichier `C:\file.txt` :

*Exemple 17-17 :*

```
using System.Net ;
using System.Net.Mail ;
class Program {
   static void Main() {
      SmtpClient client = new SmtpClient("smtp.myserver.fr",25) ;
```

```
        client.Credentials = new
           NetworkCredential("patrick.smacchia@myserver.fr", "mypwd") ;
        MailMessage msg = new MailMessage("patrick@smacchia.com",
            "destinataire@xyz.com", "my subject", "my body");
        msg.Attachments.Add(new Attachment(@"C:\file.txt"));
        client.Send(msg) ;
    }
}
```

L'espace de noms `System.Net.Mime` contient des classes pour le support de la norme *Multipurpose Internet Mail Exchange* (MIME). Cette norme décrit le formatage d'entêtes dans le corps d'un mail. Ces entêtes permettent de décrire la manière dont le corps du mail ainsi que ses attachements doivent êtres affichés. Par exemple, certains mails sont au format HTML et contiennent des images.

# *Bufférisation et compression d'un flot de données*

Cette section montre comment vous pouvez intervenir sur la façon dont les données d'un flot sont lues et écrites à l'aide de services. Ces services sont matérialisés par certaines classes du *framework* qui permettent par exemple de mettre en mémoire tampon ou de compresser/décompresser les données d'un flot. Nous verrons dans la section suivante que d'autres classes permettent aussi la sécurisation (i.e l'authentification et/ou l'encryptions) des données d'un flot.

## *Le design pattern décorateur et la mise en séries de services sur un flot de données*

Le *framework* .NET se base sur le design pattern Gof nommé *décorateur* pour réaliser l'architecture qui permet de mettre en série des services sur un flot de données. L'idée est simple : chaque classe représentant un flot de données et chaque classe représentant un service sur un flot dérive de la classe abstraite `Stream`. Une instance d'une classe représentant un service possède une référence de type `Stream` vers le flot de données sur lequel elle agit. Le client de cette instance la manipule par exemple en appelant les méthodes `Read([out]buffer)` et `Write([in]buffer)`. Le client n'a pas nécessairement conscience qu'il manipule en fait un service qui encapsule l'accès à un flot puisqu'il peut très bien se contenter d'une référence de type `Stream`. L'extrait de code suivant illustre une utilisation de cette architecture où trois services sont mis en séries sur un flot de données réseau :

```
...
  NetworkStream  networkStream = new NetworkStream(...) ;
  BufferedStream bufferedStream = new BufferedStream(networkStream,...) ;
  CryptoStream   cryptoStream = new CryptoStream(bufferedStream,...) ;
  GZipStream     gzipStream = new GZipStream(cryptoStream,...) ;
  Stream         clientStream = gzipStream as Stream ;
...
  clientStream.Read(...) ;
  clientStream.Write(...) ;
...
```

Dans cet exemple, si le code de création du flot et de ses trois services est isolé, le fait que les données du flot soient compressées/décompressées, encryptées/décryptées puis mises en mémoire tampon est complétement transparent du point de vue du code qui appelle les méthodes `Read()` et `Write()`. Voici un diagramme de classe UML pour formaliser cette architecture :

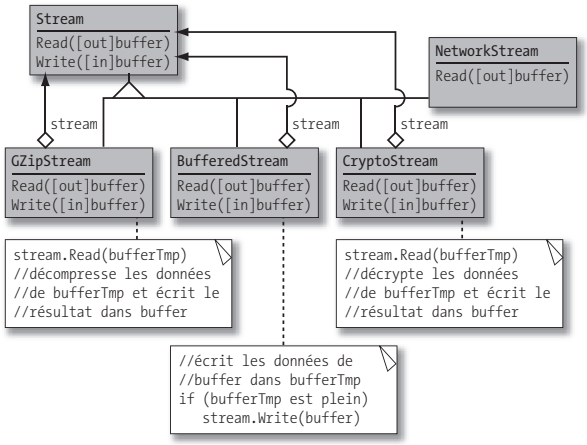


*Figure 17-1 : Le design pattern décorateur et les services sur les flots de données*

## *Bufférisation d'un flot de données*

La classe `NetworkStream` ne gère pas de zone de mémoire tampon. Concrètement, lorsque vous appelez une des méthodes `Write()` ou `WriteByte()`, les données sont directement envoyées aux clients. Un gros problème de performance peut apparaître si vous envoyez beaucoup de données sous la forme de petits paquets. En effet, chaque écriture sur le réseau consomme un volume incompressible de ressources, indépendant du nombre d'octets envoyés. Si vous envoyez beaucoup de données sous la forme de petits paquets, sans avoir à attendre de réponses du client, il serait judicieux d'envoyer tous ces petits paquets en une seule fois. C'est-à-dire qu'il serait judicieux que plusieurs appels à la méthode `Write()` ou à la méthode `WriteByte()` ne génère qu'un accès réseau. Il faut donc que vous gériez une zone de mémoire tampon pour accueillir ces paquets au fur et à mesure de leur production par le serveur. Le contenu de cette zone de mémoire tampon sera vidé en une seule fois au moment opportun.

La classe `System.IO.BufferedStream` a été conçue spécialement dans ce but. Cette classe est extrêmement simple d'utilisation. Lors de la création d'une instance, vous fournissez au constructeur une référence vers l'instance de `NetworkStream` à laquelle vous souhaitez associer une zone de mémoire tampon. La classe `BufferedStream` dérive de la classe `Stream`. Ceci implique que toutes les méthodes présentées pour utiliser un flot de données sont accessibles. L'instance de `BufferedStream` transmet les données à envoyer à l'instance de `NetworkStream` que lorsque la zone de mémoire de tampon est pleine. Vous pouvez aussi forcer l'envoi de données vers l'instance de `NetworkStream`, en appelant la méthode `Flush()` sur l'instance de `BufferedStream`.

Lors de la création d'une instance de `BufferedStream`, vous avez la possibilité de spécifier une taille pour la zone de mémoire tampon. Cette taille est le paramètre sur lequel vous devez jouer

pour obtenir les meilleures performances possibles. Empiriquement, la taille de 4Ko constitue un bon choix, mais nous vous conseillons de faire plusieurs tests avec plusieurs valeurs.

La classe `BufferedStream` peut être aussi utilisée pour lire les données. Notez qu'il est conseillé de ne pas utiliser une instance de `BufferedStream` à la fois pour lire et écrire des données. Si vous utilisez une instance de `BufferedStream` avec des données toujours plus grandes que la taille interne de la zone de mémoire tampon, il se peut que la zone de mémoire tampon ne soit jamais allouée.

Une instance de la classe `System.IO.MemoryStream` peut être utilisée dans le même esprit que `BufferedStream` pour améliorer les performances d'un flot de données réseau. La différence d'utilisation est que vous avez une marge de manœuvre plus large avec `MemoryStream`. Une instance de `MemoryStream` n'est pas explicitement liée avec une instance de `NetworkStream`. Vous ne précisez le flot de données sous-jacent que lors de l'appel à la méthode `WriteTo()`. Ainsi, vous pouvez par exemple envoyer les mêmes données contenues dans la même zone de mémoire tampon, à plusieurs clients.

La version non gérée `System.IO.UnmanagedMemoryStream` de la classe `MemoryStream` permet d'éviter la copie des données sur le tas des objets du CLR. Elle est donc plus efficace.

Notez enfin qu'il n'est pas nécessaire d'utiliser la classe `BufferedStream` pour utiliser une zone de mémoire tampon avec une instance de `FileStream`. En effet, comme nous l'avons vu, la classe `FileStream` gère nativement en interne une zone de mémoire tampon.

## *Compression d'un flot de données*

Le nouvel espace de noms `System.IO.Compression` contient les deux classes `GZipStream` et `DeflateStream` permettant de compresser de décompresser un flot de données en utilisant les algorithmes *GZip* (spécification RFC 1952 : GZIP 4.3) et *Deflate* (spécification RFC 1951 : DEFLATE 1.3). L'exemple suivant montre comment compresser un fichier :

*Exemple 17-18 :*

```
using System.IO ;
using System.IO.Compression ;
public class Program {
   public static void Main() {
      string sFileIn = @"C:\Test.txt" ;
      string sFileOut = @"C:\Test.bin" ;
      byte[] contenu = File.ReadAllBytes(sFileIn) ;
      FileStream fileStream = new FileStream(sFileOut,
                                    FileMode.Create, FileAccess.Write);
      GZipStream gzipStream = new GZipStream(fileStream,
                                      CompressionMode.Compress);
      gzipStream.Write(contenu, 0, contenu.Length) ;
      gzipStream.Close() ;
      fileStream.Close() ;
      FileInfo fileOut = new FileInfo(sFileOut) ;
      System.Console.WriteLine("Avant {0} octets", contenu.Length) ;
      System.Console.WriteLine("Après {0} octets", fileOut.Length) ;
   }
}
```

Il est regrettable que l'on ne puisse pas utiliser simplement la classe `GZipStream` pour manipuler les fichiers d'extension `.zip`. Aussi, voici un lien vers la bibliothèque gratuite et open-source *SharpZipLib* qui permet d'effectuer ceci : `http://www.icsharpcode.net/OpenSource/SharpZipLib/Default.aspx`

# *Lecture/écriture des données sur un port série*

La classe `System.IO.Ports.SerialPort` permet d'exploiter un port série d'une manière synchrone ou par événement.

Une instance de cette classe n'est pas un flot de données puisque celle-ci ne dérive pas de la classe `Stream`. En revanche, la classe `SerialPort` présente la propriété `Stream BaseStream{get;}` qui fournit un flot de données en lecture et en écriture vers le port série sous-jacent. Bien entendu, l'accès aléatoire n'est pas supporté par un tel flot de données.

La propriété `string PortName{get;set;}` spécifie le nom du port série sous-jacent. La méthode statique `string[] GetPortNames()` permet d'obtenir tous les ports séries disponibles sur la machine courante.

Les méthodes `ReadLine()` et `WriteLine()` sont prévues pour l'envoie et la réception de chaînes de caractères. La propriété `Encoding SerialPort.Encoding{get;set;}` permet de positionner l'encodage de ces chaînes de caractères.

La classe `SerialPort` permet de travailler à un niveau bas grâce aux nombreuses propriétés permettant de manipuler les bits du port série sous-jacent.

# *Support des protocoles SSL, NTLM et Kerberos de sécurisation des données échangées*

## *Présentation des protocoles*

Le protocole *SSL* (*Secure Socket Layer*) a été développé à l'origine par *Netscape* pour sécuriser l'échange de données sur internet. Ses dernières versions sont nommés *TLS* (*Transport Layer Security*). Ce protocole fournit trois services : l'authentification du serveur, l'authentification optionnelle du client et l'échange sécurisé de données. Ce protocole n'est pas lié à un protocole internet particulier. Il peut se placer en dessous des protocoles HTTP, TCP, FTP, TELNET etc.

Les protocoles *NTLM* (*NT Lan Manager*) et *Kerberos* sont les protocoles d'authentification et de sécurisation de données échangées entre plateformes *Windows*. NTLM est supporté par tous les systèmes *Windows* depuis *Windows 95/98* tandis que Kerberos est supporté par les versions plus récentes de *Windows*. Si au moins un des deux partis d'une communication ne supporte pas Kerberos, le protocole NTLM est utilisé par défaut.

## *Implémentations des protocoles dans System.Net.Security*

Le nouvel espace de noms `System.Net.Security` présentent des classes permettant d'utiliser les protocoles SSL, NTLM et Kerberos pour sécuriser les données d'un flot. Ces classes encapsulent l'API win32 relatives à ces protocoles qui est connue sous l'acronyme *SSPI* (*Security Support Provider Interface*). Les classes `NegociateSteam` et `SslStream` dérivent de la classe `AuthenticatedStream`.

## *La classe SslStream*

La classe `SslStream` est utilisée pour exploiter le protocole SSL. Reprenons le code du client/serveur TCP de la page 642 afin d'illustrer la sécurisation d'un flot de données TCP avec le protocole SSL :

*Exemple 17-19 :* *Serveur.cs*

```
...
   private static X509Certificate getServerCert() {
      X509Store store = new X509Store(StoreName.My,
                                      StoreLocation.CurrentUser) ;
      store.Open(OpenFlags.ReadOnly) ;
      X509CertificateCollection certs = store.Certificates.Find(
         X509FindType.FindBySubjectName, "CN=SslSvrCertif", true) ;
      return certs[0] ;
   }
   static void TraiteClient(NetworkStream networkStream) {
      SslStream streamSsl = new SslStream( networkStream );
      streamSsl.AuthenticateAsServer( getServerCert() );
      StreamWriter streamWriter = new StreamWriter( streamSsl ) ;
      StreamReader streamReader = new StreamReader(
                                  @"C:/Text/Fichier.txt") ;
      string sTmp = streamReader.ReadLine() ;
      try {
         while (sTmp != null) {
            Console.WriteLine("Envoi de : {0}", sTmp) ;
            streamWriter.WriteLine(sTmp) ;
            streamWriter.Flush() ;
            sTmp = streamReader.ReadLine() ;
         }
      } finally {
         streamReader.Close() ;
         streamWriter.Close() ;
         streamSsl.Close();
         networkStream.Close() ;
      }
   }
...
```

On voit que du coté serveur, un nouveau flot de données de type `SslStream` se place entre le flot de données réseau `networkStream` et le flot de données `streamWriter` utilisé pour envoyer des données.

Pour créer ce flot de données SSL, nous récupérons un certificat X509 nommé `SslSvrCertif` stocké dans le répertoire `Personnel` de l'utilisateur actuel. Le fait de passer ce certificat à la méthode `AuthenticateAsServer()` permet l'authentification SSL du serveur avec ce certificat. Voici le code du client :

*Exemple 17-20 :* `Client.cs`

```
...
      NetworkStream networkStream = tcpC.GetStream() ;
      SslStream streamSsl = new SslStream (
         networkStream,false, ValidateSvrCertificateCallback );
      streamSsl.AuthenticateAsClient( "SslSvrCertif" );
      StreamReader streamReader = new StreamReader( streamSsl );
      try {
         string sTmp = streamReader.ReadLine() ;
         while (sTmp != null) {
            Console.WriteLine("Réception de : {0}", sTmp) ;
            sTmp = streamReader.ReadLine() ;
         }
      } finally {
         streamReader.Close() ;
         streamSsl.Close();
         networkStream.Close() ;
         Console.WriteLine("Déconnexion de {0}:{1}", host, port) ;
      }
...
   static bool ValidateSvrCertificateCallback(object sender,
                                X509Certificate certificate,
                                X509Chain chain,
                                SslPolicyErrors sslPolicyErrors) {
      if ( sslPolicyErrors != SslPolicyErrors.None ) {
         Console.WriteLine(
            "Erreur dans la validation du SSL Certificate !") ;
         Console.WriteLine( sslPolicyErrors.ToString() ) ;
         return false ;
      } else return true ;
   }
...
```

Ici aussi un flot de données de type `SslStream` se place entre le flot de données réseau `networkStream` et le flot de données `streamWriter` utilisé pour recevoir des données du serveur. La méthode `ValidateSvrCertificateCallback()` est appelée lors de l'authentification du certificat du serveur. Si cette méthode retourne `true`, le serveur est considéré comme authentifié. Cette méthode est connue de notre instance de `SslStream` car elle est référencée par un délégué passé à son constructeur.

On passe le nom du certificat du serveur à la méthode `AuthenticateAsClient()`. Une autre surcharge de cette méthode permet de passer un certificat spécifique au client pour son authentification. L'authentification du client est optionnelle avec le protocole SSL et comprenez bien qu'ici, nous n'y avons pas recours.

## *La classe NegociateStream*

La classe `NegociateStream` est utilisée pour exploiter les protocoles *Windows* NTLM et Kerberos. Reprenons nos exemples afin d'illustrer l'utilisation de ces protocoles sur un flot de données TCP. Le niveau de protection le plus élevé, à savoir `EncryptAndSign`, est choisi :

*Exemple 17-21 :* *Serveur.cs*

```
...
   static void TraiteClient(NetworkStream networkStream) {
      NegotiateStream streamAuth = new NegotiateStream(networkStream);
      streamAuth.AuthenticateAsServer(
            CredentialCache.DefaultNetworkCredentials,
            ProtectionLevel.EncryptAndSign,
            TokenImpersonationLevel.Impersonation);
      WindowsPrincipal principal = new WindowsPrincipal(
            streamAuth.RemoteIdentity as WindowsIdentity);
      // Ne pas effectuer la requête si le client
      // n'est pas un administrateur.
      if( !principal.IsInRole( @"BUILTIN\Administrators" ) ){
         networkStream.Close();
         return;
      }
      StreamWriter streamWriter = new StreamWriter(streamAuth) ;
      StreamReader streamReader = new StreamReader(
            @"C:/Text/Fichier.txt") ;
      string sTmp = streamReader.ReadLine() ;
      try {
         while (sTmp != null) {
            Console.WriteLine("Envoi de : {0}", sTmp) ;
            streamWriter.WriteLine(sTmp) ;
            streamWriter.Flush() ;
            sTmp = streamReader.ReadLine() ;
         }
      } finally {
         streamReader.Close() ;
         streamWriter.Close() ;
         streamAuth.Close();
         networkStream.Close() ;
      }
   }
...
```

On observe qu'un flot de données de type `NegociateStream` vient s'intercaler entre le flot de données réseau `networkStream` et le flot de données `streamWriter` utilisé pour envoyer les données. Nous récupérons l'utilisateur *Windows* utilisé par le client avec la propriété `IIdentity NegociateStream.RemoteIdentity{get;}`. L'identité de cet utilisateur a été acheminée automatiquement par le protocole sous-jacent (NTLM ou Kerberos). Si cet utilisateur n'est pas un administrateur, nous choisissons de ne pas satisfaire sa requête. Une autre alternative aurait été par exemple d'affecter cet utilisateur au thread courant durant la durée de la requête (impersonification) afin que les vérifications des permissions soient réalisées implicitement. Ici, seule la permission de lire le fichier `C:/Text/Fichier.txt` aurait été requise.

Voici le code du client. On remarque que c'est l'identité de l'utilisateur sous lequel ce code s'exécute qui est envoyée au serveur par l'implémentation de `NegociateStream` :

*Exemple 17-22 :* *Client.cs*

```
...
   NetworkStream networkStream = tcpC.GetStream() ;
   NegotiateStream streamAuth=new NegotiateStream(networkStream);
   streamAuth.AuthenticateAsClient(
          CredentialCache.DefaultNetworkCredentials,
          WindowsIdentity.GetCurrent().Name,
          ProtectionLevel.EncryptAndSign,
          TokenImpersonationLevel.Impersonation);
   StreamReader streamReader = new StreamReader( streamAuth ) ;
   try {
      // Chaque passage dans la boucle = une ligne récupérée.
      string sTmp = streamReader.ReadLine() ;
      while (sTmp != null) {
         Console.WriteLine("Réception de : {0}", sTmp) ;
         sTmp = streamReader.ReadLine() ;
      }
   } finally {
      // Ferme les flots de données.
      streamReader.Close() ;
      streamAuth.Close();
      networkStream.Close() ;
      Console.WriteLine("Déconnexion de {0}:{1}", host, port) ;
   }
...
```

## *Cryptage d'un flot de données*

La classe `System.Security.Cryptography.CryptoStream` permet de crypter et de décrypter un flot de données, sans authentification d'aucun des deux partis. Le constructeur de la classe `CryptoStream` permet de préciser l'algorithme à utiliser pour l'encryption ou la décryption en passant une instance d'une classe qui implémente l'interface `ICryptoTransform`. Les algorithmes de cryptographie sont exposés en page 219.

# 18

# *Les applications fenêtrées (Windows Forms)*

## *Les applications fenêtrées sous les systèmes d'exploitation Windows*

Comme le nom l'indique, les applications avec fenêtres jouent un rôle prépondérant dans les systèmes d'exploitation *Windows. Microsoft* a toujours tenu à ce que le développement d'applications avec fenêtres sous ses systèmes d'exploitation, soit à la fois simple et standard. Dans ce but, *Microsoft* distribue la plupart des techniques qu'il utilise en interne pour développer ses propres interfaces homme/machine. Ceci explique la cohérence dans le style des fenêtres qui contribue grandement à la convivialité qui ressort de l'utilisation des systèmes d'exploitation *Windows*.

### *Console Application et Windows Application*

Avant les systèmes d'exploitation *Windows*, il y avait le système d'exploitation *DOS*. Sous les systèmes d'exploitation *Windows* il y a deux types d'applications :

- Les applications en mode console : on les exécute dans une fenêtre de commande, style DOS.
- Les applications avec fenêtres : on peut les exécuter à partir d'une ligne de commande ou à partir d'une icone, dans un explorateur.

Cette dualité applications en mode console/applications avec fenêtre existe encore sous .NET.

- Le compilateur C# `csc.exe` a besoin de savoir si l'exécutable qu'il produit est une application en mode console (option `/target:exe`) ou une application avec fenêtre (option `/target:winexe`).

- L'environnement de développement *Visual Studio* vous demande dès la création d'un projet C#, s'il représente une application en mode console (*Console Application*) ou une application avec fenêtres (*Windows Application*).

## *Notion de messages Windows*

Le passage de DOS aux systèmes d'exploitation *Windows* a vu naître la notion de *messages Windows*. À chaque événement (mouvement de la souris, touche clavier enfoncée ou relâchée etc), le système d'exploitation *Windows* fabrique un message qu'il envoie à l'application graphique concernée. Chaque message contient :

- Un identifiant message qui indique le type d'événement (clic droit souris, touche clavier enfoncée, etc).
- Des paramètres dont les types et le nombre varient en fonction de l'entrée (position souris sur l'écran, code touche clavier etc).

À chaque type d'événement correspond une *procédure de rappel* (*callback procedure*). Pour une fenêtre donnée, le développeur a la possibilité d'écrire, ou de réécrire, une procédure de rappel pour un événement particulier (par exemple un clic gauche sur un bouton).

Dans les applications avec fenêtres, chaque fenêtre a un thread et un seul qui attend la réception de messages dans une file d'attente propre au thread. Un tel thread peut gérer plusieurs fenêtres. Lorsqu'un message arrive, le thread exécute la procédure de rappel adéquate. Le code principal d'un tel thread est donc constitué d'une boucle qui est exécutée à chaque réception d'un message. Cette boucle contient entre autres, un gigantesque *switch* (plusieurs centaines de cas) qui associe les procédures de rappel (les *callbacks*) aux événements.

## *Évolution du développement des applications avec fenêtres*

L'évolution des techniques de développement d'applications avec fenêtres sous *Windows* a toujours convergée vers la simplification de la maintenance de ce gigantesque switch. L'introduction des *MFC* (*Microsoft Fundation Classes*) est allée dans ce sens. Le développeur n'avait plus qu'à définir les associations événements/procédures de rappel qui le concernait. Une table d'association était définie avec des macros. Elle pouvait ressembler à ceci :

```
BEGIN_MESSAGE_MAP(CDlgWizard, CDialog)
 ON_NOTIFY_EX( TTN_NEEDTEXT, 0, OnToolTips )      // Tool Tips

 ON_WM_SHOWWINDOW()   // Événement 'fenêtre vue ou cachée'.
                      // Correspond à la procédure de rappel
                      // standard OnShowWindow().

 ON_WM_MOUSEMOVE()    // Événement 'La souris bouge'
                      // Correspond à la procédure de rappel
                      // standard OnMouseMove().

 ON_WM_PAINT()        // Événement 'La région visible de la fenêtre est
                      // modifiée'. Correspond à la procédure de rappel
                      // standard OnPaint().
```

```
 ON_BN_CLICKED(IDC_BN_VALIDATE, OnBnClickedValidate)
        // Événement 'Le bouton d'ID IDC_BN_VALIDATE est cliqué'.
        // Correspond à la procédure de rappel propriétaire
        // OnBnClickedValidate().
END_MESSAGE_MAP()
```

L'IDE *Visual Studio* simplifiait cette gestion en permettant de placer les contrôles de la fenêtre à l'aide d'un éditeur *WYSIWYG* (*What You See Is What You Get*). Cet éditeur ajoutait automatiquement les associations événements/procédures de rappel dans la table. Cet éditeur avait cependant l'énorme défaut de ne pas pouvoir modifier ou supprimer les associations qu'il produisait. Il fallait une certaine expérience pour réaliser ces opérations à la main. De plus l'éditeur se servait d'un fichier unique pour décrire toutes les fenêtres d'une application. De nombreux conflits d'accès en écriture à ce fichier survenaient entre les développeurs de l'application.

Cette section est écrite au passé, car pour le développeur .NET, tout ceci est « presque » du passé. Le travail du développeur d'applications avec fenêtres est grandement simplifié sous .NET grâce à la technologie *Windows Forms*. Comme nous allons l'exposer dans la prochaine section, la description du contenu des fenêtres ainsi que les associations événements/procédures de rappel sont très intuitives, et ne font pas appels à des fichiers ou à du code superflu. Ceci implique notamment qu'une application avec fenêtres peut tout à fait être développée sans éditeur particulier. Un simple éditeur de texte comme le bloc-notes suffit.

## *L'espace de noms System.Windows.Forms*

L'espace de noms `System.Windows.Forms` contient de nombreuses classes utilisées pour créer des applications graphiques *Windows*. Ces classes peuvent être réparties en cinq groupes :

- Les *formulaires* :
  Ce sont les classes qui contiennent les comportements de base des formulaires. Il vous suffit de construire une classe qui hérite d'une de ces classes formulaires pour créer votre propre formulaire. On peut citer la classe `System.Windows.Forms.Form`, qui représente la classe de base pour tous les formulaires.
- Les *contrôles* :
  Un contrôle est un élément d'un formulaire. Une trentaine de types de contrôles classiques sont fournis par le *framework* .NET, comme le bouton ou la zone d'édition de texte. Ils ont tous la particularité de dériver de la classe `System.Windows.Forms.Control` qui décrit le comportement de base d'un contrôle. La section suivante liste toutes les classes de contrôles standard et explique comment vous pouvez aussi créer vos propres classes de contrôles.
- Les *composants* :
  Dans le domaine des applications graphiques, les composants permettent de rajouter des fonctionnalités à vos formulaires Par exemple la classe `System.Windows.Forms.ToolTip` permet d'ajouter la fonctionnalité de description par tooltip des contrôles, à un formulaire. On peut aussi citer la fonctionnalité de menu et d'aide à la demande de l'utilisateur.
  La classe de base de tous les composants est la classe `System.ComponentModel.Component`. Les classes `Control` et `Form` dérivent de la classe `Component`. La notion de composant est donc plus globale que les notions de contrôle et de formulaire. De plus, la notion de composant n'est pas cantonnée au domaine des applications graphiques. Plus de détails sur la notion de composant sont disponibles dans l'article **Class vs. Component vs. Control** des **MSDN**.

Concrètement un composant est un champ d'un formulaire. L'idée de renfermer le comportement global du formulaire dans une classe formulaire, tout en définissant les contrôles du formulaire comme des champs de cette classe, est en fait un *design pattern* (Gof) appelé *médiateur*. L'avantage de ce *design pattern* est que les composants n'ont absolument pas besoin de se connaître pour interagir entre eux. Ceci implique que les classes des composants sont totalement découplées.

- Les fenêtres de dialogues usuels :
  De nombreux dialogues usuels sont directement encapsulés dans des classes de base du framework .NET. On peut citer le dialogue de choix d'un fichier pour le charger, le dialogue de choix de fontes, le dialogue de choix d'une couleur ou le dialogue de configuration d'un dossier à imprimer.
- Les classes d'aide au développement de formulaires :
  De nombreux types sont nécessaires pour développer des formulaires. Vous avez par exemple des délégations qui représentent la signature des procédures de rappel ou des énumérations dont les valeurs affinent le style des contrôles ou des formulaires.

# *Introduction aux formulaires Windows Forms*

Nous allons créer un formulaire de conversion francs/euros basé sur le taux de conversion 1 euro = 6.55957 francs. Pour cela nous allons utiliser *Visual Studio*. À la fin de cette section nous allons montrer que nous aurions pu utiliser un simple éditeur de texte et le compilateur `csc.exe` pour arriver au même résultat.

Il faut d'abord créer un nouveau projet de type *Visual C#* ▶ *Windows Application* avec l'éditeur *Visual Studio*. Vous arrivez immédiatement sur l'éditeur de formulaire. Notez qu'une classe C# associée au formulaire est aussi créée.

L'éditeur de formulaire permet d'ajouter des contrôles à vos formulaires. Vous n'avez qu'à choisir le type de contrôle dans la boite à outil (*toolbox*) puis placer et dimensionner le contrôle sur votre formulaire. En utilisant deux contrôles de type `Button`, deux contrôles de type `TextBox` et deux contrôles de type `Label` nous avons pu créer le formulaire de la Figure 18-1.

Vous pouvez maintenant modifier les propriétés *Text* des six contrôles et du formulaire de façon à arriver au résultat de la figure précédente.

Pour avoir accès aux propriétés d'un contrôle, il suffit de cliquer droit sur le contrôle dans l'éditeur et de choisir le menu *Properties* ou de sélectionner le contrôle et de taper F4. Une fenêtre similaire à celle de la Figure 18-2 apparaît, et permet d'éditer les propriétés du contrôle. L'ensemble des propriétés est propre à chaque type de contrôle (mis à part le sous-ensemble des propriétés de la classe `Control`).

Chaque contrôle du formulaire représente un champ dans la classe du formulaire. L'éditeur a ajouté ces champs automatiquement dans la classe du formulaire.

```
...
   public class Form1 : System.Windows.Forms.Form {
      private System.Windows.Forms.Button   Francs2Euros ;
      private System.Windows.Forms.Button   Euros2Francs ;
      private System.Windows.Forms.TextBox  textBoxEuros ;
      private System.Windows.Forms.Label    label1 ;
      private System.Windows.Forms.Label    label2 ;
```

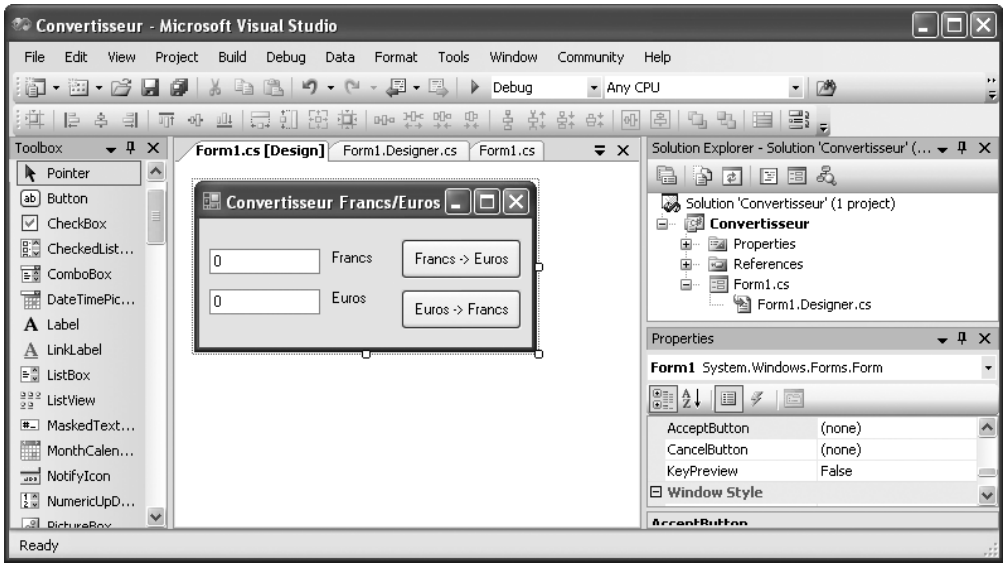


*Figure 18-1 : L'éditeur de formulaire Visual Studio .NET*

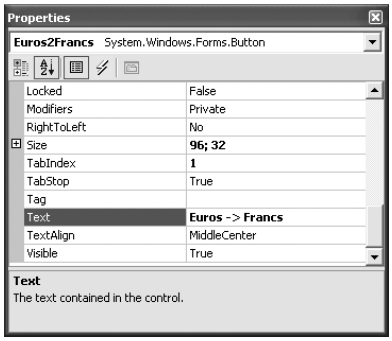


*Figure 18-2 : Edition des propriétés d'un contrôle*

```
private System.Windows.Forms.TextBox  textBoxFrancs ;
...
```

Avec *Visual Studio 2003*, un formulaire *Windows Forms* était complètement défini sur un même fichier source C#. Le code généré par cet IDE était séparé de votre propre code au moyen de régions. *Visual Studio .2005* exploite la notion de classe partielle de C# 2.0 au niveau de la gestion des formulaires. Par défaut, à chaque formulaire nommé `FormXXX` correspond deux fichiers : `FormXXX.cs` qui contient votre code et `FormXXX.Designer.cs` qui contient le code généré par *Visual Studio 2005*. Par souci de simplicité, les exemples de code des formulaires du présent ouvrage tiennent chacun sur un seul fichier source C#.

Les noms des champs du formulaire référençant les contrôles, ont été modifiés aussi dans la fenêtre des propriétés de la Figure 18-2. Dans la méthode `InitializeComponent()` de la classe du formulaire, l'éditeur a ajouté automatiquement du code qui permet de créer les contrôles, de positionner les contrôles et de configurer la taille et les champs des contrôles. Par exemple le code ajouté pour un bouton est :

```
...
   //
   // Euros2Francs
   //
   this.Euros2Francs.Location = new System.Drawing.Point(160, 56) ;
   this.Euros2Francs.Name     = "Euros2Francs" ;
   this.Euros2Francs.Size     = new System.Drawing.Size(96, 32) ;
   this.Euros2Francs.TabIndex = 1 ;
   this.Euros2Francs.Text     = "Euros -> Francs" ;
...
```

Comme vous le voyez, contrairement à la gestion des fenêtres avec les MFC, il n'y a aucune ligne de code superflue et tout ce qui concerne un formulaire se trouve dans la classe de ce formulaire.

## *Ajout d'événements*

Chaque type de contrôle présente un ensemble d'événements. On parle bien du même concept d'événement que celui présenté par C# car on parle d'événements de la classe du contrôle. À chaque événement d'un contrôle, vous pouvez associer une méthode de la classe du formulaire contenant le contrôle. Cette méthode constitue alors la procédure de rappel de l'événement. Par exemple le type de contrôle `Button` présente l'événement `Click` qui déclenche l'appel à la procédure de rappel associée, lorsque le bouton est cliqué. En double-cliquant sur le bouton de conversion *Euros vers Francs* dans l'éditeur, *Visual Studio* ajoute automatiquement la méthode suivante à la classe formulaire :

```
...
private void Euros2Francs_Click(object sender, System.EventArgs e) {
   // à remplir avec votre propre code.
}
...
```

L'éditeur associe cette méthode à l'événement `Click` avec le code suivant placé dans la méthode `InitializeComponent()` après l'initialisation des champs du contrôle `Euros2Francs` :

```
...
   this.Euros2Francs.Click += new
                              System.EventHandler(this.Euros2Francs_Click) ;
...
```

Plutôt que de double-cliquer le bouton dans l'éditeur, on peut facilement associer les événements d'un contrôle aux méthodes d'un formulaire, à partir des propriétés du contrôle comme le montre la Figure 18-3. Notez que pour accéder à ce sous-menu des propriétés, il faut cliquer l'icone qui ressemble à un éclair.

Notez enfin que le prototype des méthodes qui servent de procédures de rappel d'un événement est immuable. Il doit être égal au prototype de la délégation `System.EventHandler`.

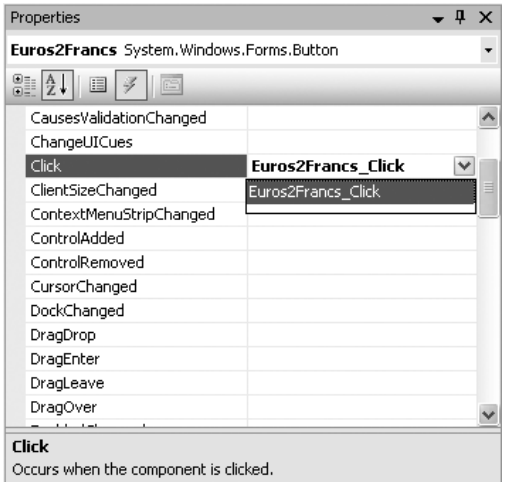


*Figure 18-3 : Edition des événements d'un contrôle*

## *Remplissage des méthodes*

Il ne nous reste plus qu'à remplir les deux méthodes `Euros2Francs_Click` et `Francs2Euros_Click` avec le code suivant :

```
...
      const double Taux = 6.55957;
      private void Francs2Euros_Click(object sender, System.EventArgs e) {
         double sommeF;
         if( double.TryParse(textBoxFrancs.Text, out sommeF) )
            textBoxEuros.Text = (sommeF / Taux).ToString();
      }
      private void Euros2Francs_Click(object sender, System.EventArgs e) {
         double sommeE;
         if ( double.TryParse(textBoxEuros.Text, out sommeE) )
            textBoxFrancs.Text = (sommeE * Taux).ToString();
      }
   ...
```

Notez la possibilité d'utiliser la méthode `double.TryParse()` pour éviter qu'une exception ne survienne pour dans le cas d'une saisie d'un double invalide.

## *Comment se passer de Visual Studio ?*

Nous aurions pu complètement nous passer de *Visual Studio* pour créer cette application fenêtrée. Il suffit d'écrire le code suivant dans un fichier source C# (par exemple `Convertisseur.cs`) puis de compiler ce fichier avec le compilateur `csc.exe` comme ceci :

```
>csc.exe /target:winexe Convertisseur.cs
```

Si l'on avait choisi `/target:exe`, le compilateur n'aurait pas produit d'erreur. La différence est que l'exécutable généré aurait besoin d'une console pour s'exécuter. Si nous le lancions à partir d'un explorateur, cet exécutable se créerait sa propre console.

Il est évident que l'éditeur de formulaire de *Visual Studio* est surtout utile pour pré visualiser et régler l'aspect du formulaire. La majorité du code suivant est constitué par l'initialisation des contrôles. Seul le code en gras a effectivement été écrit à la main.

*Exemple 18-1 :* *Convertisseur.cs*

```
using System.Drawing ;
using System.ComponentModel ;
using System.Windows.Forms ;

namespace WindowsFrancEuro {
   public class MyForm : Form {
      private Button Francs2Euros ;
      private Button Euros2Francs ;
      private TextBox textBoxEuros ;
      private Label label1 ;
      private Label label2 ;
      private TextBox textBoxFrancs ;
      public MyForm() { InitializeComponent() ; }

      private void InitializeComponent() {
         this.Francs2Euros = new Button() ;
         this.Euros2Francs = new Button() ;
         this.label1 = new Label() ;
         this.label2 = new Label() ;
         this.textBoxEuros = new TextBox() ;
         this.textBoxFrancs = new TextBox() ;
         this.SuspendLayout() ;

         // Francs2Euros
         this.Francs2Euros.Location = new Point(160, 16) ;
         this.Francs2Euros.Name = "Francs2Euros" ;
         this.Francs2Euros.Size = new System.Drawing.Size(96, 32) ;
         this.Francs2Euros.TabIndex = 0 ;
         this.Francs2Euros.Text = "Francs -> Euros" ;
         this.Francs2Euros.Click += this.Francs2Euros_Click ;

         // Euros2Francs
         this.Euros2Francs.Location = new Point(160, 56) ;
         this.Euros2Francs.Name = "Euros2Francs" ;
         this.Euros2Francs.Size = new Size(96, 32) ;
         this.Euros2Francs.TabIndex = 1 ;
         this.Euros2Francs.Text = "Euros -> Francs" ;
         this.Euros2Francs.Click += this.Euros2Francs_Click ;

         // label1
```

```
            this.label1.Location = new Point(104, 24) ;
            this.label1.Name = "label1" ;
            this.label1.Size = new Size(40, 16) ;
            this.label1.TabIndex = 3 ;
            this.label1.Text = "Francs" ;

            // label2
            this.label2.Location = new Point(104, 56) ;
            this.label2.Name = "label2" ;
            this.label2.Size = new Size(40, 16) ;
            this.label2.TabIndex = 5 ;
            this.label2.Text = "Euros" ;

            // textBoxEuros
            this.textBoxEuros.Location = new Point(8, 56) ;
            this.textBoxEuros.Name = "textBoxEuros" ;
            this.textBoxEuros.Size = new Size(88, 20) ;
            this.textBoxEuros.TabIndex = 4 ;
            this.textBoxEuros.Text = "0" ;

            // textBoxFrancs
            this.textBoxFrancs.Location = new Point(8, 24) ;
            this.textBoxFrancs.Name = "textBoxFrancs" ;
            this.textBoxFrancs.Size = new Size(88, 20) ;
            this.textBoxFrancs.TabIndex = 6 ;
            this.textBoxFrancs.Text = "0" ;

            // Form1
            this.AutoScaleDimensions = new SizeF(5, 13) ;
            this.ClientSize = new Size(264, 102) ;
            this.Controls.AddRange(new Control[] {
                     this.textBoxFrancs,
                     this.label2,
                     this.textBoxEuros,
                     this.label1,
                     this.Euros2Francs,
                     this.Francs2Euros}) ;
            this.Name = "Form1" ;
            this.Text = "Convertisseur Francs/Euros" ;
            this.ResumeLayout(false) ;
        }
        [System.STAThread]
        static void Main() { Application.Run(new MyForm()) ; }
        const double TAUX = 6.55957;
        private void Francs2Euros_Click(object sender,System.EventArgs e){
            double sommeF;
            if( double.TryParse(textBoxFrancs.Text, out sommeF) )
               textBoxEuros.Text = (sommeF / TAUX).ToString();
        }
```

```
      private void Euros2Francs_Click(object sender,System.EventArgs e){
         double sommeE;
         if ( double.TryParse(textBoxEuros.Text, out sommeE) )
            textBoxFrancs.Text = (sommeE * TAUX).ToString();
      }
   }
}
```

## *Notion de fenêtres modales*

Dans une interface graphique, il est souvent judicieux de faire apparaître une nouvelle fenêtre pour saisir ou présenter des données spécifiques. Il y a deux types de scénario lorsque l'apparition d'une nouvelle fenêtre fille est une conséquence d'une action faite sur la fenêtre mère :

- Soit la fenêtre mère reste en arrière-plan et est gelée, jusqu'à ce que la fenêtre fille disparaisse. Dans ce cas on dit que la fenêtre fille est *modale*. Pour créer un formulaire modal il faut utiliser la méthode `ShowDialog()` de la classe `Form` :

```
...
MaClasseDeFormulaire UnFormulaire = new MaClasseDeFormulaire()
// La méthode ShowDialog() ne retourne que lorsque le formulaire
// 'UnFormulaire' disparaît.
UnFormulaire.ShowDialog() ;
...
```

- Soit on peut continuer à utiliser la fenêtre mère, même si la fenêtre fille est encore présente. Dans ce cas on dit que la fenêtre fille est non modale (*modeless* en anglais). Pour créer un formulaire non modal il faut utiliser la méthode `Show()` de la classe `Form` :

```
...
MaClasseDeFormulaire UnFormulaire = new MaClasseDeFormulaire()
// La méthode Show() retourne immédiatement après la création du
// formulaire.
UnFormulaire.Show() ;
...
```

Il est plus courant d'utiliser des fenêtres modales que des fenêtres non modales. En effet, les fenêtres modales sont très adaptées à la saisie de données de façon séquentielle. Cependant, dans certains scénarios où plusieurs fenêtres sont nécessaires à l'exploitation de l'application, on ne peut qu'utiliser des fenêtres non modales. Par exemple *Visual Studio* utilise de très nombreuses fenêtres non modales, une pour la vue des fichiers, une pour la vue du projet, une pour afficher les messages du compilateurs etc. Mais comprenez bien qu'une application avec trop de fenêtres non modales est souvent très confuse pour l'utilisateur (rappelez-vous votre réaction lors de la première ouverture d'un IDE).

# *Facilités pour développer des formulaires*

## *Événements clavier et souris*

Vous avez la possibilité d'intercepter les événements produits par les périphériques d'entrées standard que sont le clavier et la souris. Tous les événements présentés existent aussi dans la classe `Form`.

Un contrôle intercepte les événements clavier à partir du moment où il a le *focus* du clavier. Un contrôle obtient le focus :

- Soit parce que l'utilisateur a cliqué sur le contrôle.
- Soit parce que l'utilisateur fait défiler le focus d'un contrôle à l'autre en appuyant sur la touche TAB ou MAJ+TAB. Vous pouvez modifier l'ordre de défilement des contrôles en affectant des valeurs croissantes aux propriétés `TabIndex` des contrôles.

La classe `System.Windows.Forms.Control`, dont hérite toutes les classes de contrôles standard *Windows Forms*, présente notamment les événements `KeyDown`, `KeyUp`, et `KeyPress` qui provoquent l'appel de leurs procédures de rappel respectivement : lorsqu'une touche est enfoncée, relâchée ou lorsqu'une touche correspondant à un caractère est enfoncée. Bien évidemment, l'événement n'est déclenché que si le contrôle a le focus. Le paramètre de cet événement est le code de la touche clavier enfoncée.

Un contrôle intercepte les événements souris à partir du moment où la souris se trouve sur la zone du contrôle. Vous pouvez néanmoins décider de capturer les événements souris même lorsque la souris est hors de la zone du contrôle en positionnant la propriété `capture` à `true`.

Plusieurs événements souris sont disponibles et il est important de comprendre quand ils sont déclenchés. Chacun de ces événements est paramétré, notamment par le bouton souris sur lequel le clic est fait (s'il y a clic) et par la position de la souris.

| Événement | Description |
|---|---|
| `MouseDown` | Un bouton de la souris est enfoncé. |
| `MouseUp` | Un bouton de la souris est relâché. |
| `Click` | Un bouton de la souris est cliqué. |
| `DoubleClick` | Un bouton de la souris est double-cliqué (il vaut mieux ne pas implémenter cet événement en même temps que l'événement `Click`. Dans ce cas *Windows* appelle les deux procédures de rappel lors d'un double-clic). |
| `MouseMove` | La souris se déplace au dessus du contrôle (ou ailleurs si la propriété `capture` est positionnée à `true`). |
| `MouseEnter` | La souris entre dans la zone du contrôle. |
| `MouseLeave` | La souris quitte la zone du contrôle. |
| `MouseHover` | La souris marque un temps d'arrêt dans le contrôle. |

| | |
|---|---|
| `MouseWheel` | La molette de la souris est actionnée. |

## *L'événement Paint*

L'événement `Paint`, avec sa procédure de rappel `OnPaint()`, est très important dans les applications graphiques sous les systèmes d'exploitation *Windows*. En effet, les affichages des fenêtres ne sont pas persistants. Cela veut dire que si une fenêtre (ou une partie d'une fenêtre) devient visible alors qu'elle ne l'était pas, il est nécessaire que le thread responsable de cette fenêtre reconstruise cette partie. Lorsqu'une fenêtre (ou une partie d'une fenêtre) devient visible, les systèmes d'exploitation *Windows* envoient le message `WM_PAINT`. Sous .NET cela se traduit par le déclenchement de l'événement `Paint` du formulaire concerné et par l'appel de la procédure de rappel `OnPaint()`. Parmi les arguments de cet événement, il y a notamment les coordonnées du rectangle à dessiner. La plupart du temps, vous ne vous occuperez pas de cet événement car par défaut, l'événement est automatiquement transmis aux contrôles qui doivent se réafficher. Néanmoins, lorsque vous utilisez la bibliothèque GDI+ ou lorsque vous faites vos propres contrôles, il faut réécrire la méthode `OnPaint()` avec votre propre code.

## *Opérations asynchrones*

Dans une application *Windows Forms*, lorsque le thread qui traite les messages *Windows* est bloqué par l'exécution d'une opération longue, l'utilisateur est en général exaspéré de voir sa fenêtre gelée. Pour pallier ce problème, on délègue en général l'opération longue soit à un thread du pool soit à un thread créé spécialement pour l'occasion.

*Windows Forms 2.0* présente la classe `BackgroundWorker` qui permet de standardiser le développement d'opération asynchrone au sein d'un formulaire. Cette classe présente notamment des facilités pour signaler périodiquement l'avancement et pour annuler l'opération. Son utilisation est illustrée par l'exemple suivant qui délègue le calcul de nombres premiers au moyen d'une instance de la classe `System.ComponentModel.BackgroundWorker`. L'avancement du traitement est signalé par une barre de progression et vous avez la possibilité d'annuler l'opération au moyen d'un bouton. Dans cet exemple, seules les méthodes `DoWork()` et `IsPrime()` sont exécutée par un thread background du pool :


*Figure 18-4 : Formulaire de calcul de nombres premiers*

*Exemple 18-2 :*

```
...
public class PrimeForm : Form {
   public PrimeForm() {
      InitializeComponent() ;
      InitializeBackgoundWorker() ;
```

```
    }
    private void InitializeBackgoundWorker() {
       backgroundWorker.DoWork += DoWork;
       backgroundWorker.RunWorkerCompleted += Complete;
       backgroundWorker.ProgressChanged += ProgressChanged;
       backgroundWorker.WorkerReportsProgress = true;
       backgroundWorker.WorkerSupportsCancellation = true;
    }
    private void DoWork(object sender,DoWorkEventArgs e) {
       BackgroundWorker worker = sender as BackgroundWorker ;
       e.Result = IsPrime((int)e.Argument, worker, e) ;
    }
    private void ProgressChanged(object sender,
                                 ProgressChangedEventArgs e) {
       progressBar.Value = e.ProgressPercentage;
    }
    private void Complete(object sender, RunWorkerCompletedEventArgs e) {
       textBoxInput.Enabled = true ;
       buttonStart.Enabled = true ;
       buttonCancel.Enabled = false ;
       if ( e.Error != null )    // Cas ou une exception a été lancée :
          MessageBox.Show(e.Error.Message) ;
       else if ( e.Cancelled )   // Cas ou annulation :
          textBoxResult.Text = "Opération annulée !" ;
       else                      // Opération réussie :
          textBoxResult.Text = e.Result.ToString() ;
    }
    private void buttonStart_Click(object sender, EventArgs e) {
       int number = 0 ;
       if (int.TryParse(textBoxInput.Text, out number)) {
          textBoxResult.Text = String.Empty ;
          textBoxInput.Enabled = false ;
          buttonStart.Enabled = false ;
          buttonCancel.Enabled = true ;
          progressBar.Value = 0 ;
          backgroundWorker.RunWorkerAsync(number);
       } else textBoxResult.Text = "entrée invalide !" ;
    }
    private void buttonCancel_Click(object sender, EventArgs e) {
       backgroundWorker.CancelAsync();
       buttonCancel.Enabled = false ;
    }
    private string IsPrime(int number, BackgroundWorker worker,
                           DoWorkEventArgs e) {
       int racine = ((int)System.Math.Sqrt(number))+1 ;
       int highestPercentageReached = 0 ;
       for (int i = 2 ; i < racine ; i++) {
          if ( worker.CancellationPending ) {
             e.Cancel = true ;
```

```
            return String.Empty ;
         } else {
            if (number % i == 0)
               return "est divisible par " + i.ToString() ;
            int percentComplete =
               (int)((float)i / (float)racine * 100) ;
            if (percentComplete > highestPercentageReached) {
               highestPercentageReached = percentComplete ;
               worker.ReportProgress(percentComplete);
            }
         }
      }
      return "est premier" ;
   }
...
```

## *Autres possibilités des formulaires*

La création d'applications graphiques avec *Windows Forms* présente un grand nombre d'autres possibilités que celles qui viennent d'être expliquées. Nous avons essayé ci-dessus, de sélectionner quelques possibilités à la fois non triviales et souvent sollicitées. Voici d'autres possibilités dont l'utilisation est détaillée dans les **MSDN** :

- La particularité de la classe `Component` est d'implémenter l'interface `System.Compo\-nent\-Model.IComponent`. Cette interface permet à un composant d'être encapsulé dans une instance d'une classe qui implémente l'interface `System.ComponentModel.IContainer`. Un *conteneur* (*container* en anglais) est un ensemble de composants et chaque composant à la connaissance de son conteneur. Cet ensemble d'interfaces propose un modèle pour l'implémentation du *design pattern Mediator*. Une conséquence de ce modèle est la possibilité d'éditer vos formulaires d'une manière WYSIWYG avec *Visual Studio*. Pour nommer cette possibilité d'édition composant/conteneur durant la phase de conception/développement d'un projet, les documentations *Microsoft* utilisent le terme « édition durant le *design-time* ». Plus de détails à propos de ce sujet sont disponibles dans l'article **Design-Time Architecture** des **MSDN**.
- La classe `System.Windows.Forms.MessageBox` permet d'afficher très simplement une fenêtre avec du texte et les combinaisons de boutons classiques :

  ```
  OK ;
  OK/Cancel ;
  Abort/Retry/Ignore ;
  Retry/Cancel ;
  Yes/No ;
  Yes/No/Cancel.
  ```

  L'affichage se fait grâce à la méthode statique `Show()` de cette classe qui est surchargées en de nombreuse versions.
- La gestion du *drag & drop* se fait grâce aux événements `System.Windows.Forms.Control.DragDrop`, `DragEnter` et `DragOver`. Chacun de ces événements accepte un argument de type `System.Windows.Forms.DragEventArgs`.

- Vous pouvez utiliser différents types de menus grâce à la classe `System.Windows.Forms.Menu` et à ses classes dérivées, `ContextMenu`, `MainMenu` et `MenuItem`.
- Vous pouvez utiliser les classes de dialogue standard suivantes :

```
System.Windows.Forms.ColorDialog
System.Windows.Forms.FileDialog
System.Windows.Forms.FontDialog
System.Windows.Forms.OpenFileDialog
System.Windows.Forms.PageSetupDialog
System.Windows.Forms.PrintControllerWithStatusDialog
System.Windows.Forms.PrintDialog
System.Windows.Forms.PrintPreviewDialog
System.Windows.Forms.SaveFileDialog
```

  La classe `System.Windows.Forms.Timer` permet d'utiliser simplement un *timer*, c'est-à-dire un objet qui déclenche un événement à intervalle de temps régulier. Plus de détails à ce sujet sont disponibles en page 169.
- La classe `System.Windows.Forms.ToolTips` permet d'afficher une fenêtre d'aide donnant des informations sur l'utilisation d'un contrôle. Pour obtenir cette aide, l'utilisateur n'a qu'à laisser la souris qu'un petit instant sur le contrôle. L'aide disparaît automatiquement lorsque la souris bouge ou après un certain moment. Tous ces délais sont configurables par des propriétés de la classe.
- La classe `System.Windows.Forms.Help` permet d'afficher l'aide complète de l'application au format HTML dans un navigateur. Cette aide est beaucoup plus générale que celle dispensée par les *tooltips*. Vous pouvez afficher l'aide (respectivement l'index de l'aide) en utilisant la méthode statique `ShowHelp()` (respectivement `ShowHelpIndex()`). Vous pouvez télécharger l'outil *HTML Help WorkShop* sur le site de *Microsoft*. Cet outil permet de créer des fichiers d'aide au format `.chm` ou `.htm`.
- La classe `System.Windows.Forms.ClipBoard` permet de placer ou de récupérer des informations sur le *clipboard*. N'utilisez jamais cette technique pour communiquer des informations entre processus. Le clipboard doit être strictement réservé aux données utilisateurs.
- La classe `System.Windows.Forms.NotifyIcon` permet de développer des applications type *tray icon*. Ces applications ont pour principe d'être résidente en mémoire et de présenter un icone dans la barre de *Windows* qui indique leur état courant. L'application *Windows Messenger* est un exemple d'application *tray icon*.
- En appelant la méthode statique `Application.EnableVisualStyles()` au démarrage d'une application *Windows Forms*, vous fait en sorte que vos contrôles ayant leur propriété `FlatStyle` positionnée à `Standard` ou `System` auront l'apparence définie par le système d'exploitation sous-jacent

# *Les contrôles standards*

## *Hiérarchie des contrôles standard disponibles*

La hiérarchie des contrôles standard est présentée ci-dessous. Les contrôles dont le nom est écrit en gras sont nouveaux par rapport à *Windows Forms 1.x* :

```
System.Object
   System.MarshalByRefObject
      System.ComponentModel.Component
         System.Windows.Forms.Control
            Microsoft.WindowsCE.Forms.DocumentList
            System.ComponentModel.Design.ByteViewer
            System.Windows.Forms.AxHost
            System.Windows.Forms.ButtonBase
              System.Windows.Forms.Button
              System.Windows.Forms.CheckBox
              System.Windows.Forms.RadioButton
            System.Windows.Forms.DataGrid
            System.Windows.Forms.DataGridView
            System.Windows.Forms.DateTimePicker
            System.Windows.Forms.GroupBox
            System.Windows.Forms.Label
               System.Windows.Forms.LinkLabel
            System.Windows.Forms.ListControl
               System.Windows.Forms.ComboBox
               System.Windows.Forms.ListBox
            System.Windows.Forms.ListView
            System.Windows.Forms.MdiClient
            System.Windows.Forms.MonthCalendar
            System.Windows.Forms.PictureBox
            System.Windows.Forms.PrintPreviewControl
            System.Windows.Forms.ProgressBar
            System.Windows.Forms.ScrollableControl
               System.Windows.Forms.ContainerControl
                  System.Windows.Forms.Form
                  System.Windows.Forms.PropertyGrid
                  System.Windows.Forms.SplitContainer
                  System.Windows.Forms.ToolStripContainer
                  System.Windows.Forms.ToolStripPanel
                  System.Windows.Forms.UpDownBase
                     System.Windows.Forms.DomainUpDown
                     System.Windows.Forms.NumericUpDown
                  System.Windows.Forms.UserControl
               System.Windows.Forms.Design.ComponentTray
               System.Windows.Forms.Panel
                  System.Windows.Forms.FlowLayoutPanel
                  System.Windows.Forms.SplitterPanel
                  System.Windows.Forms.TableLayoutPanel
                  System.Windows.Forms.TabPage
                  System.Windows.Forms.ToolStripContentPanel
               System.Windows.Forms.ToolStrip
                  System.Windows.Forms.BindingNavigator
                  System.Windows.Forms.DataNavigator
                  System.Windows.Forms.MenuStrip
                  System.Windows.Forms.StatusSTrip
```

```
                System.Windows.Forms.ToolStripDropDown
                  System.Windows.Forms.ToolStripDropDownMenu
                     System.Windows.Forms.ContextMenuStrip
          System.Windows.Forms.ScrollBar
             System.Windows.Forms.HScrollBar
             System.Windows.Forms.VScrollBar
          System.Windows.Forms.Splitter
          System.Windows.Forms.StatusBar
          System.Windows.Forms.TabControl
          System.Windows.Forms.TextBoxBase
             System.Windows.Forms.RichTextBox
             System.Windows.Forms.TextBox
             System.Windows.Forms.MaskedTextBox
          System.Windows.Forms.ToolBar
          System.Windows.Forms.TrackBar
          System.Windows.Forms.TreeView
          System.Windows.Forms.WebBrowserBase
             System.Windows.Forms.WebBrowser
```

Nous vous invitons à consulter les **MSDN** et à faire vos propres essais pour avoir plus de détails sur chacune de ces classes.

## *Les nouveaux contrôles*

Voici un rapide descriptif de chacun des nouveaux contrôles :

- Les contrôles `ToolStrip`, `MenuStrip`, `StatusStrip` et `ContextMenuStrip` :
  Ces contrôles remplacent respectivement les contrôles `ToolBar`, `MainMenu`, `StatusBar` et `ContextMenu` (ces derniers sont toujours présents pour préserver la compatibilité ascendante). En plus d'un aspect visuel plus travaillé, ces nouveaux contrôles se révèlent être particulièrement pratiques à manipuler au moment du design d'une fenêtre, notamment grâce à une API cohérente. Des nouvelles fonctionnalités ont été ajoutées telles que la possibilité de partager un rendu entre contrôles, le support des GIF animées, l'opacité, la transparence et des facilités pour sauver l'état courant (position, taille etc) dans le fichier de configuration. La hiérarchie de classes dérivées de la classe `System.Windows.Forms.ToolStripItem` constitue autant d'éléments insérables dans ce type de contrôle. Plus d'informations à ce sujet sont disponibles dans les articles **ToolStrip Technology Summary** et **ToolStrip Control Overview (Windows Forms)** des **MSDN**.
- Les contrôles `DataGridView` et `BindingNavigator` :
  Ces contrôles font partie d'un nouveau *framework* pour développer des fenêtres de présentation et d'édition de données. Ce *framework* fait l'objet de la section 18 « Présentation et édition des données » un peu plus loin dans ce chapitre. Sachez qu'il est maintenant préférable d'avoir recours à un contrôle `DataGridView` pour tout affichage de données type table ou liste d'objets.
- Les contrôles `FlowLayoutPanel` et `TableLayout` :
  Ces contrôles permettent d'agencer le positionnement des contrôles enfants qu'ils contiennent lorsque l'utilisateur en modifie taille. La philosophie d'agencement du contrôle `FlowLayoutPanel` est de lister les contrôles enfants horizontalement ou verticalement de ma-

nière à ce qu'ils soient déplacés lors du redimensionnement. Cette philosophie est comparable à ce que l'on voit lorsque l'on redimensionne un document HTML affiché par un navigateur. La philosophie d'agencement du contrôle `TablePanel` est comparable au mécanisme d'ancrage (*anchoring* en anglais) ou les contrôles enfants sont redimensionnés selon la taille du contrôle parent. Cependant, ici les contrôles enfant se trouvent dans les cellules d'une table.

- Les contrôles `SplitterPanel` et `SplitContainer` :
  L'utilisation conjuguée de ces contrôles permet d'implémenter plus simplement le partitionnement d'une fenêtre d'une manière redimensionnable que ce que l'on avait en 1.1 avec le contrôle `Splitter`.
- Le contrôle `WebBrowser` :
  Ce contrôle permet d'insérer un navigateur web directement dans vos fenêtres Windows Forms.
- Le contrôle `MaskedTextBox` :
  Ce contrôle affiche une `TextBox` dans laquelle le format du texte à insérer est contraint. Plusieurs types de masques sont présentés par défaut tel que la contrainte d'entrer une date ou un numéro de téléphone au format US. Naturellement, vous pouvez fournir vos propres masques.
- Les contrôles `SoundPlayer` et `SystemSounds` :
  La classe `SoundPlayer` permet de jouer des sons au format `.wav` tandis que la classe `SystemSounds` permet de récupérer les sons systèmes associés à l'utilisateur courant du système d'exploitation.

## *Créer vos propres contrôles*

Vous avez la possibilité de créer très facilement vos propres contrôles graphiques. Tous ceux qui ont eu à créer leurs propres *contrôles ActiveX* vont être très agréablement surpris par la simplicité de la création d'un contrôle graphique avec .NET. En général, un contrôle graphique est réutilisable. Aussi, il est judicieux de créer un assemblage partagé pour contenir le code d'un contrôle (ou un ensemble de contrôles). Vous avez la possibilité d'insérer dans vos propres contrôles :

- Des contrôles standard .NET.
- D'autres contrôles propriétaires.
- Des tracés graphiques grâce à l'utilisation de la bibliothèque GDI+, qui fait l'objet de la dernière section du présent chapitre.

Un contrôle propriétaire dérive nécessairement de la classe `System.Windows.Forms.UserControl`.

Nous vous proposons un tutorial qui montre comment créer un contrôle graphique d'affichage de pourcentages avec des colonnes. Le contrôle créé contient des contrôles standard et des tracés avec la bibliothèque GDI+. Pour plus de simplicité nous créons le contrôle et le formulaire qui l'utilise dans le même assemblage, mais il vaut mieux créer ses contrôles dans des assemblages partagés. Pour simplifier notre exposé, nous utilisons *Visual Studio*. Cependant un éditeur de texte aurait suffi pour créer ce contrôle.

Créez un nouveau projet de type *Windows Application*. Ajoutez-y un nouveau contrôle utilisateur (cliquer droit sur le projet *Add* ▶ *Add New Item...* ▶ *User Control*) que vous pouvez appeler « *ControlPourCent* ». L'idée est que le contrôle ressemble à celui de la Figure 18-5. Un client

du contrôle peut changer la valeur du pourcentage à l'exécution. Un développeur client du contrôle peut choisir s'il souhaite afficher la grille et/ou le contour, lors du développement de l'application (*design-time*) et à l'exécution.

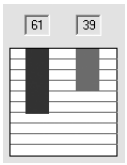


*Figure 18-5 : Aspect de notre contrôle propriétaire*

Voici le code. Notez que dans la méthode `Dessine()` nous utilisons des classes de la bibliothèque GDI+ que nous détaillerons dans la prochaine section :

*Exemple 18-3 :*

```
using System.ComponentModel ;
using System.Drawing ;
using System.Windows.Forms ;

public class ControlPourCent : UserControl {
   private TextBox PourCentLeft ;
   private TextBox PourCentRight ;

   // Propriété pour régler la valeur du pourcentage (entre 0.0 et 1.0).
   private byte m_Valeur = 30 ;
   public double Valeur {
      set {
         if (value < 0.0) m_Valeur = 0 ;
         else if (value > 1.0) m_Valeur = 100 ;
         else m_Valeur = (byte)(100.0 * value) ;
         Dessine() ;
      }
   }

   // Propriété pour décider si la grille doit être dessinée.
   private bool m_bGrille = true ;
   [Category("Appearance")]
   public bool bGrille {
      set { m_bGrille = value ; Dessine() ; }
      get { return m_bGrille ; }
   }

   // Propriété pour décider si le contour doit être dessiné.
   private bool m_bContour = true ;
   [Category("Appearance")]
   public bool bContour {
      set { m_bContour = value ; Dessine() ; }
```

```
      get { return m_bContour ; }
   }

   // Constructeur.
   public ControlPourCent() { InitializeComponent() ; }

   // Code d'initialisation du contrôle.
   #region Component Designer generated code
   private void InitializeComponent() {
      this.PourCentLeft = new TextBox() ;
      this.PourCentRight = new TextBox() ;
      this.SuspendLayout() ;

      // PourCentLeft
      this.PourCentLeft.Location = new Point(16, 8) ;
      this.PourCentLeft.Name = "PourCentLeft" ;
      this.PourCentLeft.ReadOnly = true ;
      this.PourCentLeft.Size = new Size(24, 20) ;
      this.PourCentLeft.TabIndex = 0 ;
      this.PourCentLeft.Text = "" ;
      this.PourCentLeft.TextAlign = HorizontalAlignment.Center ;

      // PourCentRight
      this.PourCentRight.Location = new Point(64, 8) ;
      this.PourCentRight.Name = "PourCentRight" ;
      this.PourCentRight.ReadOnly = true ;
      this.PourCentRight.Size = new Size(24, 20) ;
      this.PourCentRight.TabIndex = 1 ;
      this.PourCentRight.Text = "" ;
      this.PourCentRight.TextAlign = HorizontalAlignment.Center ;

      // ControlPourCent
      this.Controls.AddRange(new Control[] {
                  this.PourCentRight,
                  this.PourCentLeft}) ;
      this.Name = "ControlPourCent" ;
      this.Size = new Size(104, 168) ;
      // Affecte une procédure de rappel à l'événement Paint.
      this.Paint += new PaintEventHandler(this.ControlPourCent_Paint) ;
      this.ResumeLayout(false) ;
   }
   #endregion

   private void Dessine() {
      // Remplit les boîtes de texte.
      PourCentLeft.Text = m_Valeur.ToString() ;
      PourCentRight.Text = (100 - m_Valeur).ToString() ;

      // Il faut que la méthode Dispose() de toute instance de la
```

```
      // classe Graphics soit appelé dès que possible.
      using (Graphics g = CreateGraphics()) {
         // Remplit un rectangle de blanc
         // pour effacer l'affichage précédent.
         g.FillRectangle(Brushes.White, 3, 39, 100, 101) ;

         // Si m_bContour dessine le contour.
         if (m_bContour)
            g.DrawRectangle(new Pen(Color.Black), 2, 39, 100, 101) ;

         // Si m_bGrille dessine la grille.
         if (m_bGrille) {
            Pen p = new Pen(Color.Gray) ;
            for (int i = 1 ; i < 10 ; i++)
               g.DrawLine(p, 3, 39 + i * 10, 102, 39 + i * 10) ;
         }

         // Dessine les deux rectangles de pourcentage.
         g.FillRectangle(Brushes.Blue, 17, 40, 22, m_Valeur) ;
         g.FillRectangle(Brushes.Red, 64, 40, 22, 100 - m_Valeur) ;
      }
   }

   // Procédure de rappel de l'événement Paint
   // Pour simplifier, le contrôle est entièrement redessiné à chaque
   // réception d'un événement Paint.
   private void ControlPourCent_Paint(object sender,PaintEventArgs e) {
      Dessine() ;
   }
}
```

L'utilisation de l'attribut `CategoryAttribute` sur les propriétés `bGrille` et `bContour` permet au client du contrôle, de configurer ces propriétés à l'aide de *Visual Studio* comme le montre la Figure 18-6. Notez que vous pouvez créer vos propres catégories avec l'attribut `CategoryAttribute`. Vous pouvez aussi utiliser la douzaine de catégories proposées par *Visual Studio*. Ces catégories sont exposées dans l'article **CategoryAttribute Class** des **MSDN**.

Notez enfin que vous pouvez prévoir des facilités pour stocker directement l'état d'un contrôle dans les paramètres de l'application. Ainsi, il gardera le même aspect à chaque exécution de l'application. Cette possibilité fait l'objet de l'article **Application Settings for Custom Controls** des **MSDN**.

## *Utilisation du contrôle*

Notre contrôle s'utilise exactement de la même manière qu'un contrôle standard. Il faut juste référencer l'assemblage dans lequel est défini notre contrôle (dans notre exemple, le contrôle est dans le même assemblage que le code qui l'utilise mais ce n'est pas le cas en général).

Le formulaire client de notre contrôle est montré à la Figure 18-7. Un clic sur le bouton « *Nouveau tirage* » produit un pourcentage aléatoire qui est affecté aux trois instances du contrôle.

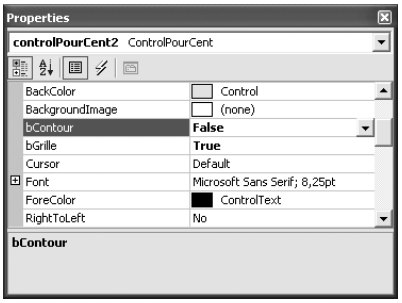


*Figure 18-6 : Configuration des propriétés d'un contrôle propriétaire avec Visual Studio .NET*

Nous avons utilisé trois instances pour illustrer le fait que vous pouvez choisir de dessiner la grille et/ou le contour.

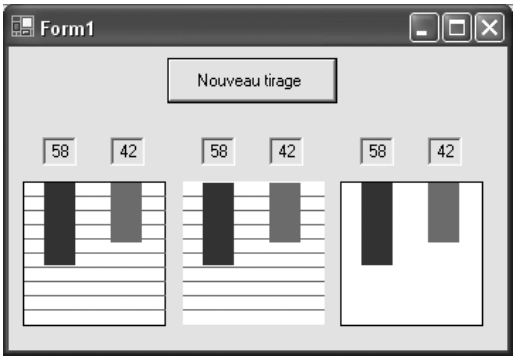


*Figure 18-7 : Formulaire client de notre contrôle propriétaire*

Voici le code :

*Exemple 18-4 :*

```
using System.Drawing ;
using System.ComponentModel ;
using System.Windows.Forms ;

public class MyForm : Form {
   private Button Tirage ;
   private ControlPourCent controlPourCent1 ;
   private ControlPourCent controlPourCent2 ;
   private ControlPourCent controlPourCent3 ;

   public MyForm() { InitializeComponent() ; }

#region Windows Form Designer generated code
```

```
private void InitializeComponent() {
   this.controlPourCent1 = new ControlPourCent() ;
   this.Tirage = new Button() ;
   this.controlPourCent2 = new ControlPourCent() ;
   this.controlPourCent3 = new ControlPourCent() ;
   this.SuspendLayout() ;

   // Tirage
   this.Tirage.Location = new Point(112, 8) ;
   this.Tirage.Name = "Tirage" ;
   this.Tirage.Size = new Size(120, 32) ;
   this.Tirage.TabIndex = 0 ;
   this.Tirage.Text = "Nouveau tirage" ;
   this.Tirage.Click += new System.EventHandler(this.Tirage_Click) ;

   // controlPourCent1
   this.controlPourCent1.bContour = true;
   this.controlPourCent1.bGrille = true;
   this.controlPourCent1.Location = new Point(8, 56) ;
   this.controlPourCent1.Name = "controlPourCent1" ;
   this.controlPourCent1.Size = new Size(104, 152) ;
   this.controlPourCent1.TabIndex = 1 ;

   // controlPourCent2
   this.controlPourCent2.bContour = false;
   this.controlPourCent2.bGrille = true;
   this.controlPourCent2.Location = new Point(120, 56) ;
   this.controlPourCent2.Name = "controlPourCent2" ;
   this.controlPourCent2.Size = new Size(104, 144) ;
   this.controlPourCent2.TabIndex = 2 ;

   // controlPourCent3
   this.controlPourCent3.bContour = true;
   this.controlPourCent3.bGrille = false;
   this.controlPourCent3.Location = new Point(232, 56) ;
   this.controlPourCent3.Name = "controlPourCent3" ;
   this.controlPourCent3.Size = new Size(104, 144) ;
   this.controlPourCent3.TabIndex = 3 ;

   // Form1
   this.AutoScaleDimensions = new SizeF(5, 13) ;
   this.ClientSize = new Size(352, 214) ;
   this.Controls.AddRange(new Control[] {
      this.controlPourCent3,
      this.controlPourCent2,
      this.controlPourCent1,
      this.Tirage}) ;
   this.Name = "MyForm" ;
   this.Text = "MyForm" ;
```

```
      this.ResumeLayout(false) ;
   }
#endregion
   [System.STAThread]
   static void Main() { Application.Run(new MyForm()) ; }

   private void Tirage_Click(object sender, System.EventArgs e) {
      // Calcul d'un valeur aléatoire entre 0.0 et 1.0.
      System.Random random = new System.Random() ;
      double d = random.NextDouble() ;
      controlPourCent1.Valeur = d;
      controlPourCent2.Valeur = d;
      controlPourCent3.Valeur = d;
   }
}
```

# *Présentation et édition des données*

## *Créer rapidement une fenêtre de présentation et d'édition de données avec Visual Studio 2005*

Grâce à certains outils de *Visual Studio 2005*, il est possible de développer une fenêtre d'édition et de présentation des données évoluée telle que celle de la Figure 18-8 en deux minutes montre en main. C'est ce que l'on appelle le *RAD* (*Rapid Application Development*).

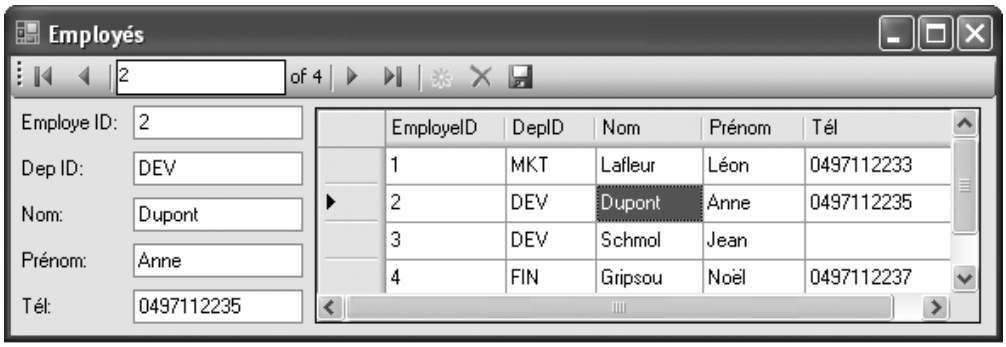


*Figure 18-8 : Edition et présentation d'une table*

Voici un petit tutorial d'utilisation de *Visual Studio 2005* qui explique comment développer cette fenêtre. Il faut au préalable avoir inséré le `DataSet` typé présenté en page dans votre projet (il est d'ailleurs nécessaire d'avoir assimilé cette notion de `DataSet` typé pour aborder la présente section). Prévoyez aussi une fenêtre vierge nommée `FormEmployes` dans votre projet :

Rendre visible la vue *Data Source* (menu *Data* ▶ *Show Data Sources*) ▶ Dépliez la source de données `ORGANISATIONDataSet` ▶ Sélectionnez le nœud `EMPLOYES` ▶ Dépliez la combobox qui apparaît ▶ Choisissez l'option *DataGridView* ▶ Glissez/Déposez le nœud EMPLOYES sur la fenêtre `FormEmployes` en cours d'édition ▶ *Visual Studio 2005* a ajouté plusieurs entités à votre fenêtre :

- Un DataSet typé de type ORGANISATIONDataSet qui représente la source de données ainsi qu'un TableAdapter de type EMPLOYESTableAdapter.
- Un contrôle non visuel de type BindingSource qui représente la liaison entre notre source de données (i.e le DataSet typé) et les contrôles visuels de présentation et d'édition des données.
- Un contrôle visuel de type DataGridView qui présente la liste des employés et qui permet aussi d'éditer chaque cellule.
- Un contrôle visuel de type BindingNavigator qui contient des boutons style VCR pour naviguer dans la liste des employés. Ce contrôle présente aussi un bouton pour ajouter un nouvel employé, un bouton pour détruire l'employé couramment sélectionné et un bouton pour sauver les changements réalisés sur la liste des employés.

En plus de tous le code nécessaire à la création et à l'initialisation de ces contrôles et objets, *Visual Studio* a généré deux méthodes qui se trouvent dans le fichier source FormEmployes.cs :

```
...
private void FormEmployes_Load(object sender, EventArgs e) {
  this.eMPLOYESTableAdapter.Fill(this.oRGANISATIONDataSet.EMPLOYES);
}
private void bindingNavigatorSaveItem_Click(object sender, EventArgs e){
  if (this.Validate()) {
    this.eMPLOYESBindingSource.EndEdit();
    this.eMPLOYESTableAdapter.Update(this.oRGANISATIONDataSet.EMPLOYES);
  } else {
    System.Windows.Forms.MessageBox.Show(this,
                  "Validation errors occurred.", "Save",
                  System.Windows.Forms.MessageBoxButtons.OK,
                  System.Windows.Forms.MessageBoxIcon.Warning) ;
  }
}
...
```

Il est clair que la méthode FormEmployes_Load() appelée lors du chargement de la fenêtre contient le code qui permet remplir la table EMPLOYES de notre DataSet typé à partir de la base de données. Comme expliqué en page , ceci se fait à l'aide du TableAdapter associé à la table EMPLOYES.

Il est clair la méthode bindingNavigatorSaveItem_Click() appelée lors du click du bouton de sauvegarde du contrôle BindingNavigator, contient le code qui permet de sauver dans la base les modifications effectués. Ici aussi, cette opération se fait à l'aide du TableAdapter associé à la table EMPLOYES.

Si vous vous intéressez au fichier FormEmployes.Designer.cs vous vous apercevrez que tout est configurable et notamment l'ensemble des boutons contenus dans le contrôle BindingNavigator et l'ensemble des colonnes du contrôle DataGridView.

Pour obtenir la fenêtre de la Figure 18-8 il faut maintenant ajouter les cinq contrôles de type Label et les cinq contrôles de type EditBox qui permettent de présenter une vue détaillée des cinq champs de l'employé couramment sélectionné. Sélectionnez le nœud EMPLOYES dans la source de données ORGANISATIONDataSet ▶ Dépliez la combobox qui apparaît ▶ Choisissez l'option *Details* ▶ Glissez/Déposez le nœud EMPLOYES sur la fenêtre FormEmployes en cours

d'édition ▶ *Visual Studio 2005* a ajouté les dix contrôles. On obtient ainsi une vue maîtresse détaillée très pratique pour l'édition et l'insertion des données des lignes d'une table. Notez qu'en page 942 nous exposons comment obtenir ce type de vue dans le cadre d'une page web.

Il est intéressant d'analyser les lignes de code dédiées à la création des liaisons. Elles se trouvent toutes dans la méthode généré `InitializeComponent()` du fichier `FormEmployes.Designer.cs`. Nous avons ajouté des commentaires explicatifs :

```
private void InitializeComponent() {
   ...
   // Crée le BindingSource.
   this.eMPLOYESBindingSource =
      new System.Windows.Forms.BindingSource(this.components) ;
   ...
   ((System.ComponentModel.ISupportInitialize)
      (this.eMPLOYESBindingSource)).BeginInit() ;
   ...
   // Lie le BindingSource avec la table EMPLOYES du DataSet typé.
   this.eMPLOYESBindingSource.DataMember = "EMPLOYES" ;
   this.eMPLOYESBindingSource.DataSource = this.oRGANISATIONDataSet ;
   ...
   // Lie le BindingNavigator avec le BindingSource.
   this.eMPLOYESBindingNavigator.BindingSource =
                                          this.eMPLOYESBindingSource ;
   ...
   // Lie la DataGridView avec le BindingSource.
   this.eMPLOYESDataGridView.DataSource = this.eMPLOYESBindingSource ;
   ...

   // Pour chaque colonne de la DataGridView, indique le nom de
   // la colonne associée dans la table des employés.
   this.dataGridViewTextBoxColumn1.DataPropertyName = "EmployeID" ;
   ...
   this.dataGridViewTextBoxColumn2.DataPropertyName = "DepID" ;
   ...
   this.dataGridViewTextBoxColumn3.DataPropertyName = "Nom" ;
   ...
   this.dataGridViewTextBoxColumn4.DataPropertyName = "Prénom" ;
   ...
   this.dataGridViewTextBoxColumn5.DataPropertyName = "Tél" ;
   ...

   // Crée les liaisons entre les TextBox et le DataSet typés.
   this.employeIDTextBox.DataBindings.Add(
                  new System.Windows.Forms.Binding("Text",
                     this.eMPLOYESBindingSource, "EmployeID", true)) ;
   ...
   this.depIDTextBox.DataBindings.Add(
                  new System.Windows.Forms.Binding("Text",
                     this.eMPLOYESBindingSource, "DepID", true)) ;
```

```
   ...
   this.nomTextBox.DataBindings.Add(
                   new System.Windows.Forms.Binding("Text",
                      this.eMPLOYESBindingSource, "Nom", true)) ;
   ...
   this.prénomTextBox.DataBindings.Add(
                   new System.Windows.Forms.Binding("Text",
                      this.eMPLOYESBindingSource, "Prénom", true)) ;
      ...
   this.télTextBox.DataBindings.Add(
                   new System.Windows.Forms.Binding("Text",
                      this.eMPLOYESBindingSource, "Tél", true)) ;
   ...
   ((System.ComponentModel.ISupportInitialize)
      (this.eMPLOYESBindingSource)).EndInit() ;
   ...
}
```

## *Utilité du contrôle BindingSource*

Vous pouvez vous passer d'un contrôle de type `BindingSource` pour lier un contrôle de type `DataGridView` directement à une source de données tel qu'une `DataTable`. L'exemple suivant montre une fenêtre avec une `DataGridView` remplie avec le contenu de la table `EMPLOYES` et une `DataGridView` rempli avec le contenu de la table `DEPARTEMENTS` :

*Exemple 18-5 :*

```
...
public class MyForm : Form {
...
   private void Form1_Load(System.Object sender, System.EventArgs e) {
      depDataGridView.DataSource = dSet.DEPARTEMENTS;
      empDataGridView.DataSource = dSet.EMPLOYES;
      depTableAdapter.Fill(dSet.DEPARTEMENTS) ;
      empTableAdapter.Fill(dSet.EMPLOYES) ;
   }
...
   private DataGridView empDataGridView ;
   private DataGridView depDataGridView ;
   private DEPARTEMENTSTableAdapter depTableAdapter ;
   private EMPLOYESTableAdapter empTableAdapter ;
   private ORGANISATIONDataSet dSet ;
...
```

Bien que l'on puisse s'en passer, les objets de type `BindingSource` peuvent se révéler utiles pour plusieurs raisons :

- *By design*, ils représentent un niveau d'indirection entre les contrôles de présentation et d'édition de données et les sources de donnés. Cette indirection permet de changer simplement de source de données sans toucher au code d'une fenêtre.

- Comme nous l'avons vu, les contrôles `BindingNavigator`, `DataGridView` et `Binding\-Source` ont été conçus de manière à collaborer facilement.
- Comme nous l'avons vu, on peut se servir d'un contrôle `BindingSource` pour lier des données aux contrôles de présentation au moment du design de la fenêtre avec *Visual Studio 2005*.
- Les contrôles de type `BindingSource` vous permettent de fournir votre propre logique de création d'un nouvel élément. Plus de détails à ce sujet sont disponibles dans l'article **How to: Customize Item Addition with the Windows Forms BindingSource** des **MSDN**.
- Grâce aux propriétés `Filter` et `Sort` de la classe `BindingSource`, vous pouvez filtrer ou trier les données d'une source avant de les présenter.
- Les contrôles de type `BindingSource` permettent d'exploiter simplement une relation « *one to many* ». Considérons la relation nommée `Est_Employé_Par` entre les tables `EMPLOYES` et `DEPARTEMENTS` de notre `DataSet` typé (cette relation est définie en page 710). Créons une fenêtre avec deux `DataGridView` et deux `BindingSource`. La première `BindingSource` est liée avec la table `DEPARTEMENT`. La seconde est liée avec la première avec pour valeur de la propriété `DataMember` le nom de la relation. La seconde `BindingSource` est alors capable de détecter les changements de sélection de département sur la première. Le cas échéant, elle applique la relation et ne retourne que les employés appartenant au département sélectionné :

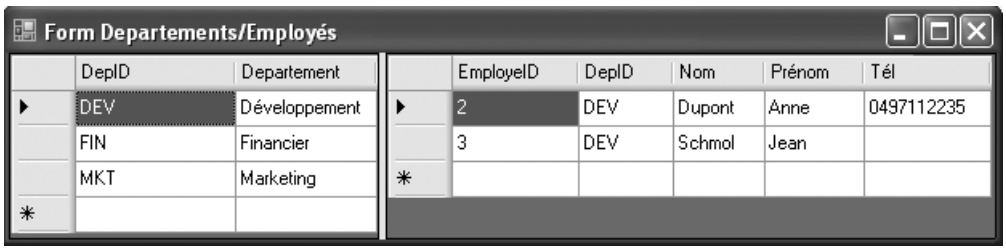


*Figure 18-9 : BindingSource et relations*

*Exemple 18-6 :*

```
...
   private void Form1_Load(System.Object sender, System.EventArgs e) {
      depDataGridView.DataSource = depBindingSource;
      empDataGridView.DataSource = empBindingSource;
      depTableAdapter.Fill(dSet.DEPARTEMENTS) ;
      empTableAdapter.Fill(dSet.EMPLOYES) ;
      depBindingSource.DataSource = dSet ;
      depBindingSource.DataMember = "DEPARTEMENTS" ;
      empBindingSource.DataSource = depBindingSource;
      empBindingSource.DataMember = "Est_Employé_Par";
   }
...
```

## *Liaisons entre BindingSource et une source de données*

Un avantage à utiliser des instances de la classe `BindingSource` est la possibilité qu'ils offrent pour s'abstraire des types de source de données. Quelque soit la source de données sous jacente, le consommateur d'une instance de `BindingSource` peut toujours exploiter les données au moyen de l'interface `IBindingList`. Cette interface implémente les interfaces `IEnumerables`, `IList` et `ICollection` et elle est implémentée par la classe `BindingSource`. En plus des méthodes d'accès aux données des interfaces implémentées, l'interface `IBindingList` présente des propriétés pour renseigner un client sur les possibilités supportées par la source (trie, édition, insertion etc), des méthodes telles que `AddNew()` pour ajouter un élément ou `Find()` pour rechercher un élément et un événement `ListChanged` déclenché à chaque modification des données ou lorsque la source elle-même est modifiée.

Comme nous l'avons vu, on peut affecter une source de données de type `DataTable` à un objet de type `BindingSource` par l'intermédiaire de la propriété `object DataSource{get;set;}`. On peut aussi affecter un `DataSet` à cette propriété à condition de préciser le nom de la table qui fait office de source de données par l'intermédiaire de la propriété `string DataMember{get;set;}`. On peut aussi affecter à cette propriété :

- N'importe quel objet qui implémente l'interface `IEnumerable`.
- N'importe quel objet qui représente un tableau (i.e dont la classe dérive de la classe `System.Array`).
- Une instance de la clases `System.Type` pour préciser le type des objets qui peuvent être ajoutés.
- N'importe quel objet. Le type d'un tel objet constituera alors le type des objets qui peuvent être ajoutés.

## *Avoir une liste d'objets pour source de données*

Nous venons de voir que nous ne sommes pas contraint de nous lier avec un objet ADO.NET tel qu'un `DataSet` ou une `DataTable`. N'importe quelle liste d'objets de données peut faire office de source de données. Cette particularité est essentielle pour développer des architectures 3-Tiers ou N-Tiers où les données subissent un traitement entre la couche de persistance et la couche de présentation. L'exemple suivant illustre ceci. Nous avons une liste d'objets de type `Employe` que nous souhaitons pouvoir présenter et éditer dans une `DataGridView`. Cette liste dérive du type `BindingList<Employe>` aussi, nous pouvons nous passer d'un objet de type `BindingSource` pour la lier avec la `DataGridView`. Remarquez que nous implémentons la méthode `BindingList<>.AddNewCore()`. Elle est appelée automatiquement par la `DataGridView` lors de l'ajout d'un employé. Nous implémentons aussi l'événement `UserDeletingRow` pour détecter la suppression d'un employé. Enfin, puisque notre classe `Employe` implémente l'interface `IEditableObject`, la `DataGridView` nous informe de la sélection d'un employé avec `BeginEdit()`, de la fin de l'édition d'un `Employe` avec `EndEdit()` ou de l'annulation de l'édition d'un employé avec `CancelEdit()`. L'annulation de l'édition se fait lorsque l'utilisateur appuie sur la touche ESC. Lors de l'appel de `EndEdit()`, les propriétés `Nom` et `Prenom` de l'objet employé concerné ont déjà été positionnées aux nouvelles valeurs saisies par l'utilisateur.

*Exemple 18-7 :*

```
using System ;
using System.ComponentModel ;
```

```
using System.Windows.Forms ;
public partial class MyForm : Form {
   private DataGridView dataGridView ;
   EmployeList list = new EmployeList() ;
   public MyForm() {
      InitializeComponent() ;
      list.Add(new Employe("Lafleur", "Léon")) ;
      list.Add(new Employe("Dupont", "Anne")) ;
      dataGridView.DataSource = list ;
   }
   private void InitializeComponent() {
      this.dataGridView = new System.Windows.Forms.DataGridView() ;
      dataGridView.Dock = DockStyle.Fill ;
      dataGridView.AutoGenerateColumns = true ;
      dataGridView.UserDeletingRow += UserDeletingRowHandler;
      this.Controls.Add(this.dataGridView) ;
   }
   protected virtual void UserDeletingRowHandler(object s,
         DataGridViewRowCancelEventArgs e) {
      if ( MessageBox.Show("Êtes vous sûr(e) ?", string.Empty,
               MessageBoxButtons.YesNo) == DialogResult.Yes)
         ((Employe)e.Row.DataBoundItem).Deleting() ;
   }
   [STAThread]
   static void Main() { Application.Run(new MyForm()) ; }
}
class EmployeList : BindingList<Employe> {
   public EmployeList() { this.AllowNew = true ; }
   protected override object AddNewCore() {
      Employe emp = new Employe("-", "-") ;
      Add(emp) ;
      return emp ;
   }
}
public class Employe : IEditableObject {
   private string m_Nom ;
   private string m_Prenom ;
   public Employe(string nom, string prenom) {
      m_Nom = nom ; m_Prenom = prenom ;
   }
   public string Nom    { get{return m_Nom;}    set{m_Nom = value;} }
   public string Prenom { get{return m_Prenom;} set{m_Prenom = value;} }
   void IEditableObject.BeginEdit() { }
   void IEditableObject.CancelEdit() { }
   void IEditableObject.EndEdit() { }
   public void Deleting() { }
}
```

# *Internationaliser les fenêtres*

Avons d'aborder la présente section, il faut avoir assimilé les concepts d'internationalisation d'une application .NET qui font l'objet de la section 2 « Internationalisation et assemblages satellites », page 34.

Dans le cas d'une application *Windows Forms*, *Visual Studio 2005* présente des facilités pour éditer plusieurs versions localisées d'une même fenêtre. Dans les propriétés d'une fenêtre, vous trouverez dans la catégorie Design les propriétés Localizable qui vaut false par défaut et Language qui vaut (Default) par défaut. Si vous souhaitez éditer plusieurs versions d'une même fenêtre XXX.cs relatives à plusieurs cultures, il faut d'abord positionner la propriété Localizable à true. Un fichier XXX.resx est alors automatiquement associé à la fenêtre. Si vous sélectionner une culture yy-ZZ pour la propriété Language de votre fenêtre, *Visual Studio* se met automatiquement en mode édition pour cette culture. Lors du premier changement sur un contrôle, un fichier XXX.yy-ZZ.resx est alors automatiquement associé à la fenêtre. Il contient bien entendu les valeurs des ressources spécifiques à la culture yy-ZZ. Pour ajouter des contrôles à votre fenêtre il faut revenir à la valeur (Default) pour la propriété Language de la fenêtre.

Il est intéressant de remarquer que le code d'initialisation des contrôles (i.e dans la méthode InitializeComponents()) est différent selon qu'une fenêtre est « localisée » ou pas. Dans le cas localisé, vous remarquerez des appels à une méthode ComponentResourceManager.ApplyRessource() qui s'occupent de charger les bonnes versions des ressources à l'exécution selon la culture courante.

# *La bibliothèque GDI+*

La bibliothèque *GDI+* (GDI pour *Graphical Device Interface*) contient de nombreuses classes qui permettent d'effectuer toutes sortes de tracés : des tracés de lignes, des tracés de courbes, des dégradés, afficher des images etc. GDI+ remplace l'ancienne bibliothèque GDI utilisées par les développeurs sous *Windows*. En plus de bénéficier de toute la convivialité des langages .NET, cette bibliothèque présente de nouvelles fonctionnalités, notamment en ce qui concerne les formats des fichiers images et l'affichage de dégradés. La bibliothèque GDI+ supporte les formats JPG, PNG, GIF et BMP alors que la bibliothèque GDI ne supportait que le format BMP (à moins d'utiliser d'autres librairies non standard).

## *La classe System.Drawing.Graphics*

La bibliothèque GDI+ permet de dessiner, c'est-à-dire de faire toutes sortes de formes géométriques tel que des rectangles, des lignes et même des courbes de Bézier. Dans les méthodes de chaque contrôle ou formulaire, on peut se procurer une instance de la classe `System.Drawing.Graphics` en appelant la méthode `CreateGraphics()` de la classe `Control` ou `Form`. Les instances de cette classe constituent le support sur lequel on peut dessiner, de la même façon qu'une feuille de papier est le support du dessinateur. Pour ceux qui ont déjà travaillé avec GDI, les instances de cette classe sont les équivalentes des instances *Device Context* (DC).

Il est important d'appeler la méthode `Dispose()` dès que possible, sur toutes les instances de la classe `Graphics` obtenues par l'appel à une méthode `CreateGraphics()`.

## La classe System.Drawing.Pen

De la même manière qu'un dessinateur a besoin d'un crayon pour tracer une courbe, les méthodes de la classe `Graphics` pour le tracé de lignes, de courbes ou de contour de figures, ont besoin d'une instance de la classe `System.Drawing.Pen`.

- Les constructeurs de la classe `Pen` acceptent soit une instance de la structure `System.Drawing.Color` pour spécifier la couleur du trait, soit une instance de la classe `System.Drawing.Brush` pour spécifier le type de remplissage d'un trait épais.
- Vous pouvez indiquer l'épaisseur, en pixel, du trait avec la propriété `Width` de la classe `Pen`.
- Vous pouvez indiquer si vous souhaitez un tracé plein, en tiré ou en pointillé avec la propriété `DashStyle` de la classe `Pen`.
- Vous pouvez indiquer quel type de dessin doit être placé aux extrémités du trait avec la propriété `StartCap` et `EndCap` de la classe `Pen`.

Par exemple le code suivant affiche la courbe de Bézier de la Figure 18-10 :

*Exemple 18-8 :*

```
...
      using (Graphics g = CreateGraphics()) {
         Pen pen = new Pen( Color.Black ) ;
         pen.Width = 5;
         pen.DashStyle = DashStyle.Dash;
         pen.StartCap = LineCap.RoundAnchor;
         pen.EndCap = LineCap.ArrowAnchor;
         g.DrawBezier(pen,
                      new Point(10, 30),
                      new Point(30, 200),
                      new Point(50, -100),
                      new Point(70, 100)) ;
      }
...
```

*Figure 18-10 : Affichage d'une courbe de Bézier avec GDI+*

## La classe System.Drawing.Brush

De la même manière qu'un dessinateur a besoin d'un pinceau pour remplir une surface, les méthodes de la classe `Graphics` pour le remplissage d'une surface, ont besoin d'une instance d'une classe dérivée de la classe `System.Drawing.Brush`. Ces classes dérivées sont les suivantes (elles sont toutes dans l'espace de noms `System.Drawing.Drawing2D`) :

- `SolidBrush` : Brosse à utiliser pour un remplissage uniforme.
- `HatchBrush` : Brosse à utiliser pour un remplissage avec hachures.
- `TextureBrush` : Brosse à utiliser pour un remplissage avec une image en arrière-plan.
- `LinearGradientBrush` : Brosse à utiliser pour un remplissage avec un dégradé de couleur.
- `PathGradientBrush` : Brosse à utiliser pour un remplissage avec un dégradé de couleur (plus élaboré qu'avec `LinearGradientBrush`).

Par exemple le code suivant affiche le pentagone de la Figure 18-11, rempli d'une manière alternée par un croisement de diagonales :

*Exemple 18-9 :*

```
...
      using (Graphics g = CreateGraphics()) {
         Brush brush = new HatchBrush( HatchStyle.DiagonalCross,
                                       Color.White, Color.Black) ;
         Point[] pts = new Point[5] ;
         pts[0] = new Point(50, 3) ;
         pts[1] = new Point(30, 100) ;
         pts[2] = new Point(80, 30) ;
         pts[3] = new Point(4, 35) ;
         pts[4] = new Point(70, 100) ;
         g.FillClosedCurve(brush, pts, FillMode.Alternate) ;
      }
...
```

*Figure 18-11 : Remplissage d'une courbe avec GDI+*

## *Afficher du texte dans vos dessins*

La méthode `DrawString()` de la classe `Graphics` permet d'afficher du texte dans vos dessins. Par exemple le code suivant affiche le « hi » de la Figure 18-12 :

*Exemple 18-10 :*

```
...
      using (Graphics g = CreateGraphics()) {
         Brush brush = new HatchBrush( HatchStyle.DiagonalBrick ,
                                       Color.White , Color.Black) ;
         g.DrawString("hi", new Font("Times", 70), brush,
                      new Point(5, 5));
      }
...
```


*Figure 18-12 : Affiche de texte avec GDI+*

## Gérer des images avec GDI+

La classe abstraite `System.Drawing.Image` sert de classe de base aux classes : `System.Drawing.Bitmap` et `System.Drawing.Imaging.Metafile`.

### Images définies par leurs pixels

La classe `Image` sert à charger à partir d'un fichier, à modifier, à afficher et à sauvegarder dans des fichiers, des images bitmap (i.e décrites exhaustivement par les états de leurs pixels). Les formats d'image supportés sont définis par les membres statiques de `System.Drawing.Imaging.ImageFormat`, parmi lesquels on trouve :

| Propriétés statiques de `ImageFormat` | Format associé |
|---|---|
| `Bmp` | Le format d'image bitmap. Précisons que ce format ne compresse pas l'image. |
| `Gif` | Le format GIF (*Graphic Interchange Format*). Précisons qu'une des caractéristiques de ce format est de réduire le nombre de couleurs de l'image pour la compresser (en général 256 couleurs). De plus, ce format permet de produire des animations. |
| `Jpeg` | Le format JPEG (*Joint Photo Expert Group*). Précisons que ce format compresse l'image avec une perte d'information. De plus, le taux de compression est paramétrable. |
| `Png` | Le format W3C PNG (*Portable Network Graphics*). Précisons que la particularité de ce format est de compresser l'image sans perte d'information. Il est particulièrement adapté aux copies d'écran. |
| `Tiff` | Le format TIFF (*Tag Image File Format*). |

La propriété `PixelFormat` de la classe `Image` détermine le nombre de *bits par pixel* (*bpp*) dans l'image et prend ses valeurs dans l'énumération `System.Drawing.Imaging.PixelFormat`. Dans le cas des formats de pixel où la valeur attribuée à chaque pixel détermine un index dans un tableau de couleur (appelé *palette* de l'image), vous devez utiliser la propriété `Palette` de la classe `Image`.

Pour afficher une image dans un formulaire ou dans un contrôle, il suffit d'utiliser une des versions surchargées de la méthode `DrawImage()` de la classe `Graphics` (présentée un peu plus haut).

Les transformations que vous pouvez appliquer à une image sont limitées aux « flip » (i.e retournements verticaux et horizontaux) et aux rotations d'angle droit. On verra dans la prochaine section comment traiter plus sérieusement une image.

Il est conseillé d'intégrer les images dans des fichiers de ressources intégrés à l'assemblage courant ou à un assemblage satellite.

### *Images définies par des opérations*

La classe `System.Drawing.Imaging.Metafile` dérive de la classe `Image`. Elle présente la possibilité de charger à partir d'un fichier, de construire ou de sauver dans un fichier, des images définies à partir d'opérations simples comme le tracé d'ellipse ou de traits etc.

La bibliothèque GDI supporte le format *EMF* (*Enhanced Meta File*) qui peut stocker des opérations. La bibliothèque GDI+ supporte le format *EMF+* qui peut servir à stocker toutes opérations de EMF, plus un certain nombre d'opérations propres à EMF+. Il y a donc une compatibilité ascendante entre les formats EMF et EMF+. Voici un exemple d'utilisation de la classe `Metafile` :

*Exemple 18-11 :*

```
using System.Drawing.Imaging ;
...
public class Form1 : System.Windows.Forms.Form {
   private Metafile m_Metafile ;
   private void OnClick1(object sender, System.EventArgs e) {
      using (Graphics g = CreateGraphics()) {
         System.IntPtr hDC = g.GetHdc();
         m_Metafile = new Metafile(hDC, EmfType.EmfPlusOnly);
         using (Graphics metafilegraphic =
                   Graphics.FromImage(m_Metafile))
         using (Brush brush = new SolidBrush(Color.Black)) {
            metafilegraphic.FillEllipse(brush, 10, 10, 50, 50);
            metafilegraphic.FillRectangle(brush, 5, 5, 10, 10);
         }
         // Il faut toujours libérer les hDC sinon
         // une exception est lancée.
         g.ReleaseHdc(hDC);
      }
   }
   public void OnClick2(object sender, System.EventArgs e) {
      if( m_Metafile != null)
         using (Graphics g = CreateGraphics()) {
            g.DrawImage(m_Metafile, 10, 10);
         }
   }
   ...
}
...
```

## *Optimisation des traitements d'images*

Le traitement d'image consiste à modifier les couleurs des pixels de l'image selon certaines opérations mathématiques. Chaque pixel est codé par trois valeurs entières, une pour le vert, une pour le rouge, une pour le bleu. L'intervalle de valeurs possibles pour ces valeurs entières dépend du nombre de *bits par pixel.* La meilleure qualité d'image possible est obtenue avec 24 bits par pixel, soit une valeur entre 0 et 255 pour chaque composante, soit un peu plus de 16 millions de couleurs. L'œil humain ne peut distinguer plus de 16 millions de couleurs.

Nous allons montrer le traitement d'image qui consiste à inverser les couleurs. Supposons que le nombre de bits par pixel est de 24. L'inversion des couleurs consiste à affecter le complément à 255 pour chacune des trois composantes, pour chacun des pixels de l'image. La Figure 18-13 montre l'application de ce traitement à une image. Pour l'anecdote sachez que cette image, souvent utilisée pour tester les traitements d'image, est la photographie de la playmate suédoise *Lena Soderberg*, extraite du magazine *Playboy* en 1972. Par la suite, elle fut invitée à certaines conférences sur le traitement d'image.

*Figure 18-13 : Lena et le traitement d'image*

Avec le *framework* .NET, il y a deux façons de procéder :

- Soit nous utilisons les méthodes `SetPixel()` et `GetPixel()` de la classe `Bitmap`. Voici l'extrait pertinent du code source :

*Exemple 18-12 :*

```
...
      using (Graphics g = CreateGraphics()) {
         Bitmap m_Bmp = new Bitmap("Lena.jpg") ;
         g.DrawImage(m_Bmp, new Point(5, 5)) ;
         // Attend 1 seconde...
         System.Threading.Thread.Sleep(1000) ;
         int width = m_Bmp.Width ;
         int height = m_Bmp.Height ;
         Color cSrc, cDest ;
         for (int y = 0 ; y < height ; y++)
            for (int x = 0 ; x < width ; x++) {
               cSrc = m_Bmp.GetPixel(x, y) ;
               cDest = Color.FromArgb(255 - cSrc.R, 255 - cSrc.G,
                                      255 - cSrc.B) ;
               m_Bmp.SetPixel(x, y, cDest) ;
            }
         g.DrawImage(m_Bmp, new Point(5, 5)) ;
      }
...
```

- Soit nous passons directement par des pointeurs pour accéder aux pixels de l'image. **Cette technique optimise le code par rapport à la technique précédente d'un facteur 20 à 100 selon le traitement à appliquer !** Cette technique est un peu plus délicate à implémenter, mais un tel facteur d'optimisation vaut bien ces efforts. Il faut notamment tenir compte des points suivants :
  - La méthode (ou la partie du code) qui réalise le traitement doit être qualifiée avec le mot-clé `unsafe` pour pouvoir manipuler des pointeurs (voir page 501).
  - Il faut verrouiller les accès à la zone de mémoire du bitmap en utilisant les méthodes `LockBits()`/`UnlockBits()` de la classe `Bitmap`.

  Voici l'extrait pertinent du code source :

*Exemple 18-13 :*

```
...
   public struct StructPixel {
      public byte R ; public byte G ; public byte B ;
   }
...
      using (Graphics g = CreateGraphics()) {
         Bitmap m_Bmp = new Bitmap("Lena.jpg") ;
         g.DrawImage(m_Bmp, new Point(5, 5)) ;
         // Attend 1 seconde...
         System.Threading.Thread.Sleep(1000) ;
         unsafe {
            int width = m_Bmp.Width ;
            int height = m_Bmp.Height ;
            BitmapData BmpData = m_Bmp.LockBits(
               new Rectangle(0, 0, width, height),
               ImageLockMode.ReadWrite,
               m_Bmp.PixelFormat) ;
            StructPixel* pCurrent = null ;
            StructPixel* pBmp = (StructPixel*)BmpData.Scan0 ;
            for (int y = 0 ; y < width ; y++) {
               pCurrent = pBmp + y * height ;
               for (int x = 0 ; x < height ; x++) {
                  pCurrent->R = (byte)(255 - pCurrent->R) ;
                  pCurrent->G = (byte)(255 - pCurrent->G) ;
                  pCurrent->B = (byte)(255 - pCurrent->B) ;
                  pCurrent++ ;
               }
            }
            m_Bmp.UnlockBits(BmpData) ;
         }
         g.DrawImage(m_Bmp, new Point(5, 5)) ;
      }
...
```

Dans ce cas précis, le facteur d'optimisation obtenu selon des tests sur une machine de référence, est d'environ 97.

## *Animation et double buffering*

Il est aisé de créer une animation en enchaînant l'affichage d'images à hauteur de plusieurs dizaines de fois par seconde. Pour réaliser ceci, on utilise en général une instance de la classe `System.Windows.Forms.Timer` qui se charge de déclencher à intervalle régulier l'appel à une méthode par le thread de la fenêtre. L'exemple suivant montre comment produire une animation représentant un carré qui tourne au milieu d'une fenêtre :

*Exemple 18-14 :*

```
using System.Drawing ;
using System.Windows.Forms ;
using System.Drawing.Drawing2D ;
public partial class AnimForm : Form {
   private float angle ;
   private Timer timer = new Timer() ;
   public AnimForm() {
      timer.Enabled = true;
      timer.Tick += OnTimer;
      timer.Interval = 20 ; // 20 milliseconds => 50 images par seconde.
      timer.Start();
   }
   private void OnTimer(object sender, System.EventArgs e) {
      angle ++ ;
      if (angle > 359)
         angle = 0 ;
      Refresh();
   }
   protected override void OnPaint(PaintEventArgs e) {
      Graphics g = e.Graphics ;
      Matrix matrix = new Matrix() ;
      matrix.Rotate(angle, MatrixOrder.Append) ;
      matrix.Translate(this.ClientSize.Width / 2,
                       this.ClientSize.Height/ 2, MatrixOrder.Append) ;
      g.Transform = matrix ;
      g.FillRectangle(Brushes.Azure, -100, -100, 200, 200) ;
   }
   [System.STAThread]
   public static void Main() {
      Application.Run(new AnimForm()) ;
   }
}
```

Si vous exécutez cet exemple, vous vous apercevrez que l'animation n'est pas parfaite. En effet, vous observerez des scintillements dans l'affichage. Ce défaut mineur mais perceptible est dû à la non synchronisation entre la fréquence d'affichage de votre écran et la fréquence de production des images. Concrètement, il arrive que votre carré soit affiché en plein milieu de sa construction.

Pour pallier à ce problème, on utilise la technique dite du *double buffering*. Cette technique consiste à maintenir en mémoire deux buffers graphiques. À chaque instant, un de ces buffers

contient la dernière image créée tandis que l'autre contient l'image en cours de construction. Dés qu'une image est produite, le rôle des buffers est inversé. Il est particulièrement aisé d'utiliser cette technique sur vos propres formulaires. Il suffit d'appeler la méthode `SetStyle()` avec les bons arguments dans le constructeur de votre formulaire après l'initialisation des composants. Ainsi pour ne plus avoir ce problème lors de la rotation de notre carré il suffit de réécrire notre exemple comme ceci :

*Exemple 18-15 :*

```
...
      timer.Start() ;
      SetStyle(
         ControlStyles.AllPaintingInWmPaint |
         ControlStyles.UserPaint |
         ControlStyles.OptimizedDoubleBuffer,true) ;
   }
   private void OnTimer(object sender, System.EventArgs e) {
...
```

Il se peut que ce mécanisme de double buffering ne soit pas adapté à vos animations. Dans ce cas, vous pouvez avoir recours aux classes `BufferedGraphicsContext` et `BufferedGraphics` afin gérer vous-même les buffers. Une instance de `BufferedGraphics` s'obtient à partir de la méthode `BufferedGraphicsContext.Allocate()`. Une telle instance gère un buffer en interne. Vous pouvez avoir accès à ce buffer au moyen de la propriété `BufferedGraphics.Graphics{get;}`. Une fois que vous avez dessiné sur ce buffer, il faut appeler la méthode `BufferedGraphics.Render()` afin d'afficher son contenu à l'écran. Voici notre exemple réécrit avec ces classes :

*Exemple 18-16 :*

```
using System.Drawing ;
using System.Windows.Forms ;
using System.Drawing.Drawing2D ;
public partial class AnimForm : Form {
   private float angle ;
   private Timer timer = new Timer() ;
   private BufferedGraphics bufferedGraphics;
   public AnimForm() {
      BufferedGraphicsContext context = BufferedGraphicsManager.Current;
      context.MaximumBuffer = new Size(this.Width + 1, this.Height + 1);
      bufferedGraphics = context.Allocate(this.CreateGraphics(),
              new Rectangle(0, 0, this.Width, this.Height));

      timer.Enabled = true ;
      timer.Tick += OnTimer ;
      timer.Interval = 20 ; // 50 déclenchements par seconde.
      timer.Start() ;
   }
   private void OnTimer(object sender, System.EventArgs e) {
      angle ++ ;
      if (angle > 359)
         angle = 0 ;
```

```
      Graphics g = bufferedGraphics.Graphics;
      g.Clear(Color.Black) ;
      Matrix matrix = new Matrix() ;
      matrix.Rotate(angle, MatrixOrder.Append) ;
      matrix.Translate(this.ClientSize.Width / 2,
                       this.ClientSize.Height/ 2, MatrixOrder.Append) ;
      g.Transform = matrix ;
      g.FillRectangle(Brushes.Azure, -100, -100, 200, 200) ;
      bufferedGraphics.Render( Graphics.FromHwnd(this.Handle) );
   }
   [System.STAThread]
   public static void Main() {
      Application.Run(new AnimForm()) ;
   }
}
```

# 19

# *ADO.NET 2.0*

## *Introduction aux bases de données*

### *Notion de SGBD*

Pratiquement tous les logiciels utilisent un *système de persistance* au sens large du terme. Par exemple la base des registres de *Windows* ou même un fichier `.ini` peuvent être vus comme des systèmes de persistance. Pour gérer un grand volume d'informations, les logiciels utilisent des *Systèmes de Gestion de Base de Données* (*SGBD*) développés en général par des entreprises tierces comme *Oracle* ou *Microsoft*. On peut citer les SGBD *SQL Server*, *Access*, *MySql*. Gérer des données ne se limite pas qu'à stocker des données. Les SGBD fournissent un grand nombre de fonctionnalités comme la recherche de données à partir de critères ou la protection des données. Certains développeurs développent leurs propres SGBD, souvent pour des raisons de performances voire par méconnaissance du modèle relationnel, mais cette pratique reste marginale et contestable.

### *Le modèle relationnel*

Les premiers SGBD ont fait leur apparition à la fin des années 60, dans le cadre des programmes spatiaux américains. Un progrès décisif a été réalisé dans les années 70 avec l'invention du *modèle relationnel*. Le modèle relationnel est basé sur un modèle mathématique qui permet de présenter les données d'une manière simple dans des tables.

La notion de table est assez proche de celle de *relation*. Une colonne d'une table s'appelle un attribut de la relation et est définie par un nom. Les éléments d'un attribut prennent leurs valeurs dans un *domaine* (en général un type). Il faut préciser pour chaque colonne s'il est obligatoire que toutes les lignes contiennent un élément valide (i.e dans le domaine). L'ensemble de la description des colonnes d'une table (i.e des attributs d'une relation) est appelé *schéma de la relation*. Les lignes d'une table sont aussi appelées *tuples* ou *enregistrements* en français ou *rows*

ou *records* en anglais. Une *clé primaire* (*primary key* en anglais) est l'ensemble des colonnes dont la connaissance de valeurs permet d'identifier une ligne unique de la table considérée. Souvent, la clé primaire n'est constituée que d'une colonne.

Le modèle relationnel permet d'éviter les redondances d'informations qui menacent l'intégrité de la table et qui consomment des ressources. Cette fonctionnalité importante est possible grâce aux *clés étrangères* (*foreign key* en anglais). Au lieu de disperser la même donnée dans plusieurs endroits de la base, on utilise des clés étrangères qui permettent de référencer la donnée. La donnée n'est donc pas dupliquée et reste accessible. Grâce à ce système, on peut stocker des structures complexes dans une base de données relationnelle, comme une arborescence de données. Un autre avantage du modèle relationnel est de pouvoir assurer l'intégrité des données de la base, grâce à la possibilité de décrire des *contraintes d'intégrité*. Par exemple si vous avez une table dont les lignes décrivent des voitures et une autre table dont les lignes décrivent des marques de voitures, vous pouvez garantir la contrainte suivante : à chaque voiture correspond une marque.

L'*algèbre relationnelle* contient six opérations de base, qui agissent sur des relations et produisent des relations. Ces opérations sont l'union, la différence, le produit cartésien, la projection, la restriction et la jointure. Au moyen de ces opérations, l'utilisateur d'une base de données relationnelle peut accéder aux données d'une base et les modifier.

## *Le langage SQL*

Le langage non procédural et standardisé nommé *SQL* (*Structured Query Langage*) a été développé pour accéder aux données d'une base relationnelle grâce à des requêtes. Ces requêtes modélisent des opérations de l'algèbre relationnelle. Le langage SQL permet de réaliser des requêtes très complexes en quelques mots. Cet ensemble d'opérations sur les données du langage SQL est nommé *DML* pour *Data Manipulation Langage*. On se sert aussi de requêtes SQL pour construire et modifier la structure d'une base de données (insertion de table, assignation de droits aux utilisateurs etc). Cette partie du langage est nommée *DDL* pour *Data Definition Langage*.

La plupart des SGBD à l'heure actuelle sont basés sur le modèle relationnel et supportent un langage dérivé de SQL. Par exemple, la gamme de SGBD *SQL Server* édité par *Microsoft* supporte le langage *T-SQL* (*Transact SQL*) qui en plus des possibilités de base de SQL permet de déclarer des variables, de contrôler des transactions, de gérer des exceptions, de gérer le format XML etc. La partie du langage relative à ces possibilités est nommée *DCL* pour *Data Control Langage*.

## *Nécessité d'une architecture distribuée*

Les applications avec interfaces utilisateurs qui accèdent à une base de données sont de deux types :

- Il y a les applications monolithiques, qui encapsulent dans un même exécutable l'interface utilisateur et le code pour accéder à la base.
- Les applications distribuées, dont seule la partie serveur est habilitée à accéder à la base. Le serveur peut être pris en charge par ASP.NET, utiliser les facilités de COM+ ou être écrit complètement. De nombreux types de middleware peuvent être utilisés entre les clients et le serveur (HTTP/SOAP, .NET Remoting etc). Les clients peuvent être légers (navigateur web) ou riches (exécutables *Windows Forms* par exemple).

Le choix d'une de ces deux architectures pour vos applications est fondamental. Le premier choix n'est judicieux que dans le cas de petites applications qui évolueront peu.

Le choix du type d'architecture doit se faire tôt dans le cycle de vie du projet.

# *Introduction à ADO.NET*

L'appellation *ADO.NET* englobe à la fois l'ensemble des classes du *framework* .NET utilisées pour manipuler les données contenues dans des SGBD relationnels, et la philosophie d'utilisation de ces classes.

Avant ADO.NET, la technologie *ADO* (*ActiveX Data Object*) constituait l'ensemble des classes qu'il fallait utiliser pour accéder aux bases de données dans les environnements *Microsoft*. Malgré son nom, ADO.NET est beaucoup plus qu'un successeur de ADO. Bien que ces deux technologies aient une finalité commune, de profondes différences les séparent, notamment parce qu'ADO.NET est beaucoup plus complet.

## *Mode connecté et mode déconnecté*

Les notions de *mode déconnecté* et de *mode connecté* décrivent la façon dont une application travaille avec une base de données. Il faut d'abord introduire la notion de *connexion* avec une base de données. Une connexion est une ressource qui est initialisée et utilisée par une application pour travailler avec une base de données. Une connexion avec une base de données peut prendre les états ouvert et fermé. Si une application détient une connexion ouverte avec une base de données, on dit qu'elle est connectée à la base. Une connexion est en général initialisée à partir d'une chaîne de caractères qui contient des informations sur le type de SGBD supportant la base de données et/ou sur la localisation physique de la base de données. Tout ceci est détaillé un peu plus loin.

Lorsqu'une application travaille avec ADO .NET, de nombreux cas de figures sont possibles :

- Une application travaille en mode connecté si elle charge des données de la base au fur et à mesure de ses besoins. L'application reste alors connectée à la base. L'application envoie ses modifications sur les données à la base au fur et à mesure qu'elles surviennent.
- Une application travaille en mode déconnecté dans les cas suivants :
    - Si l'application charge des données de la base, qu'elle se déconnecte de la base, qu'elle exploite et modifie les données, qu'elle se reconnecte à la base pour envoyer les modifications sur les données.
    - Si l'application charge des données de la base, qu'elle se déconnecte de la base et qu'elle exploite les données. Dans ce cas, on dit que l'application accède à la base en lecture seule.
    - Si l'application récupère des données à partir d'une autre source que la base de données (par exemple à partir d'une saisie manuelle d'un utilisateur ou d'un document XML) puis qu'elle se connecte à la base pour y stocker les nouvelles données.
    - Si l'application n'utilise pas la base de données. Préciser ce cas à un sens, car une application peut travailler avec des classes de ADO.NET sans pour autant utiliser une base de données. Par exemple, on verra que les classes de ADO.NET sont particulièrement adaptées pour la présentation de données au format XML.

La philosophie d'ADO était de travailler en mode connecté. Il est très difficile, voire impossible, de créer des serveurs performants en terme de montée en charge (*scalable*), lorsque l'on travaille en mode connecté. Notez qu'avec ADO, on peut travailler en mode déconnecté au prix de beaucoup d'efforts.

Précisons la signification du mot *scalable*. On dit d'une application qu'elle est *scalable*, si, lorsque vous ajoutez du *hardware* pour l'exécuter (processeur, RAM, PCs etc) vous obtenez un gain de performance commensurable à la quantité de *hardware* ajoutée. En pratique, on observe que l'ajout de *hardware* pour exécuter une application ne provoque pas forcément un gain de performance. En effet, de nombreux goulets d'étranglement subsistent toujours à différents niveaux (*locking* des données, *middleware*...). Seule l'architecture de l'application peut minimiser les effets néfastes de ces goulets d'étranglement.

Avec ADO.NET, on peut travailler soit en mode déconnecté soit en mode connecté. De nombreuses fonctionnalités intéressantes, que nous détaillons plus loin, sont disponibles pour chacun des deux modes.

La faiblesse du mode connecté est qu'il génère de très nombreux accès à la base de données et plus généralement, il génère de nombreux accès réseau si la base de données est séparée physiquement du reste de l'application. La faiblesse du mode déconnecté est qu'il amène à consommer beaucoup de mémoire, puisqu'une partie de la base est récupérée en mémoire vive pour chaque appel client. Cette faiblesse peut être prohibitive pour un serveur devant gérer un grand nombre de clients, chaque client obligeant le serveur à manipuler beaucoup de données en mémoire.

Bien qu'une pré-analyse soit toujours nécessaire, il est en général plus efficace de travailler en mode déconnecté avec ADO.NET.

## *Les fournisseurs de données (Providers)*

Un *fournisseur de données* (*provider* en anglais) est une couche logicielle permettant de communiquer avec un SGBD relationnel spécifique. Voici les quatre fournisseurs de données supportés par défaut par le *framework* .NET :

- Le SGBD *SQL Server* a son propre fournisseur de données. Les classes de ce fournisseur de données se trouvent dans l'espace de noms `System.Data.SqlClient`. Ce fournisseur de données fonctionne avec les versions 7.0 2000 et 2005 de *SQL Server*. Bien évidemment, les fonctionalités spécifiques à une version de *SQL Server* ne peuvent pas être exploitées à partir de ce fournisseur de données si vous travaillez avec une version antérieure de ce produit.
- Un autre fournisseur de données permet de communiquer avec les fournisseurs de données qui supportent l'API *OleDB*. OleDB est une API permettant d'accéder aux données d'un SGBD, avec la technologie COM. Les classes de ce fournisseur de données se trouvent dans l'espace de noms `System.Data.OleDbClient`. Notez qu'il faut utiliser ce fournisseur de données si vous travaillez avec des versions antérieures à 7.0 de *SQL Server*.
- Il existe un fournisseur de données .NET qui se place au dessus du protocole *ODBC* (*Open DataBase Connectivity*). Ce fournisseur de données géré permet d'exploiter les fournisseurs de données non gérés qui supportent l'API ODBC (mais pas la totalité, voir le site de *Microsoft* pour plus de détails). Les classes de ce fournisseur d'accès sont disponibles dans l'espace de noms `System.Data.Odbc`.

- Il existe un fournisseur de données .NET qui est spécialisé pour l'utilisation de bases de données *Oracle*. Les classes de ce fournisseur de données sont disponibles dans l'espace de noms `System.Data.OracleClient`.

À l'exception du fournisseur de données *Oracle* qui se trouve dans la DLL `System.Data.OracleClient.dll` les trois autres fournisseurs de données se trouvent dans la DLL `System.Data.dll`.

## *Schéma général de ADO.NET*

Voici un schéma général (mais simplifié) de l'architecture ADO.NET :

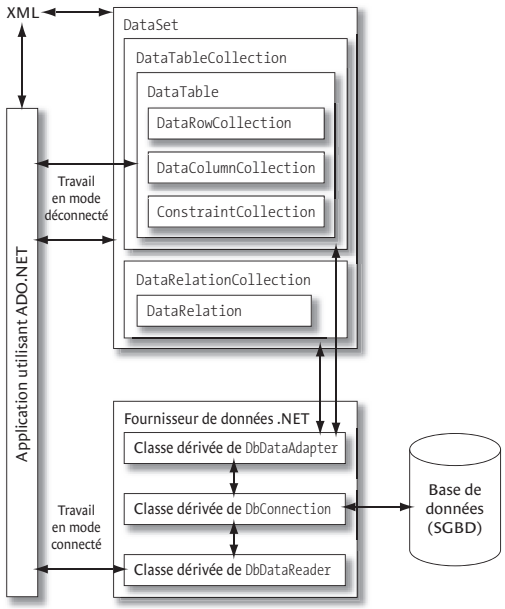


*Figure 19-1 : Schéma général de ADO.NET*

Ce schéma illustre plusieurs principes clés d'ADO.NET :

- Tous les accès au SGBD se font par l'intermédiaire d'une connexion dont l'implémentation fait partie du fournisseur de données.
- Les classes `DataSet` et `DataTable` permettent de travailler en mode déconnecté. Dans ce mode, tout chargement ou toute sauvegarde de données se fait par l'intermédiaire d'un objet *adaptateur*. Un `DataSet` est un cache de données relationnelles hébergé à l'exécution dans le domaine d'application courant. Vous pouvez remplir un `DataSet` avec des données provenant d'une base de données, vous déconnecter, consommer les données du `DataSet` et éventuellement les modifier, puis enfin vous reconnecter à la base pour éventuellement sauver les changements effectués sur les données.
- La classe `DataReader` permet de travailler en mode connecté.

- Comme nous le verrons dans le chapitre suivant consacré à XML plusieurs passerelles existent entre le monde du stockage des données au format XML et le monde du stockage relationnel des données.

## *La BD utilisée pour illustrer les exemples*

Dans tous les exemples de ce chapitre nous allons utiliser une base de données nommée ORGANISATION gérée par le SGBD *SQL Server*. Les exemples sont compatibles avec les versions 2000 et 2005 sauf indication contraire. Nous utiliserons bien entendu le fournisseur de données *SQL Client* spécifique à *SQL Server*. En dernière section, nous décrirons des fonctionnalités propres à ce fournisseur de données.

Notre base de données ORGANISATION contient les tables DEPARTEMENTS et EMPLOYES qui sont censées modéliser l'organisation interne d'une entreprise en supposant que chaque employé appartient à exactement un département. Les tables ainsi que les contraintes peuvent être créées avec le script *T-SQL* suivant :

*Exemple 19-1 :*

```
CREATE TABLE [dbo].[EMPLOYES] (
   [EmployeID] [int] IDENTITY (1, 1) ,
   [DepID] [char] (3)  NOT NULL ,
   [Nom] [nvarchar] (30)  NOT NULL ,
   [Prénom] [nvarchar] (30)  NOT NULL ,
   [Tél] [nvarchar] (20)  NULL
) ON [PRIMARY]
GO
CREATE TABLE [dbo].[DEPARTEMENTS] (
   [DepID] [char] (3) NOT NULL,
   [Departement] [nvarchar] (30)  NOT NULL
) ON [PRIMARY]
GO
ALTER TABLE EMPLOYES ADD CONSTRAINT ID_Primaire
PRIMARY KEY (EmployeID)
GO
ALTER TABLE DEPARTEMENTS ADD CONSTRAINT DepID_Primaire
PRIMARY KEY (DepID)
GO
ALTER TABLE EMPLOYES ADD CONSTRAINT Est_Employé_Par
FOREIGN KEY (DepID) REFERENCES DEPARTEMENTS(DepID)
GO
```

La clé primaire de la table EMPLOYES est constituée par la colonne EmployeID. La syntaxe IDENTITY (1, 1) signifie que la valeur entière de cette colonne est affectée automatiquement par *SQL Server* qui se sert d'un compteur interne, selon l'algorithme suivant : le compteur interne est positionné à 1 lorsqu'une ligne est insérée pour la première fois dans cette table ; à chaque insertion d'une ligne le compteur interne est incrémenté de 1. À chaque insertion d'une ligne dans la table EMPLOYES, la valeur EmployeID de la nouvelle ligne prend la valeur du compteur interne.

La clé primaire de la table `DEPARTEMENTS` est constituée par la colonne `DepID`. Elle doit être positionnée par l'utilisateur à chaque ajout d'une ligne dans la table. Voici le diagramme de notre base de données :

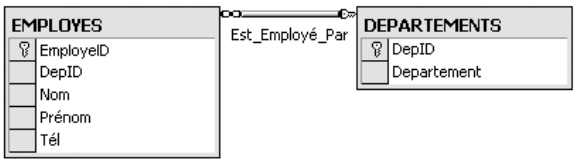


*Figure 19-2 : Diagramme de notre base de données*

Nous supposerons lors de l'utilisation de nos exemples, que l'état initial de la base de données est celui-ci :

| **Table DEPARTEMENTS** | |
|---|---|
| DepID | Departement |
| DEV | Développement |
| FIN | Financier |
| MKT | Marketing |

| **Table EMPLOYES** | | | | |
|---|---|---|---|---|
| EmployeID | DepID | Nom | Prénom | Tél |
| 1 | MKT | Lafleur | Léon | 0497112233 |
| 2 | DEV | Dupont | Anne | 0497112235 |
| 3 | DEV | Schmol | Jean | |
| 4 | FIN | Gripsou | Noël | 0497112237 |

Voici le script T-SQL permettant de remplir la base. Notez que l'on délègue au SGBD le calcul des valeurs de la colonne `EmployeID` :

*Exemple 19-2 :*

```
INSERT INTO DEPARTEMENTS VALUES ('DEV','Développement')
INSERT INTO DEPARTEMENTS VALUES ('FIN','Financier')
INSERT INTO DEPARTEMENTS VALUES ('MKT','Marketing')
GO
SET IDENTITY_INSERT EMPLOYES OFF
```

```
GO
INSERT INTO EMPLOYES VALUES ('MKT','Lafleur','Léon','0497112233')
INSERT INTO EMPLOYES VALUES ('DEV','Dupont','Anne','0497112235')
INSERT INTO EMPLOYES (DepID,Nom,Prénom) VALUES ('DEV','Schmol','Jean')
INSERT INTO EMPLOYES VALUES ('FIN','Gripsou','Noël','0497112237')
GO
```

La chaîne de connexion que nous utilisons dans nos exemples pour accéder à cette base est la suivante :

```
"server = localhost ; uid=sa ; pwd = ; database = ORGANISATION"
```

Pensez à configurer le mode d'authentification à *SQL Server and Windows* pour exécuter les exemples avec cette chaîne de connexion.

# *Connexion et fournisseurs de données*

## *Découpler une application d'un fournisseur de données*

Un grand nombre de paradigmes se retrouvent dans la plupart des SGBD relationnels : Ils sont accessibles au travers d'une connexion, on peut manipuler les données à l'aide de requêtes SQL etc. Bien entendu chaque couple SGBD/fournisseur de données associé a ses spécificités. Par exemple seul le fournisseur de données de SQL Server permet d'obtenir les statistiques d'une connexion.

Dans le nouvel espace de noms `System.Data.Common` ADO.NET2 présente plusieurs classes abstraites qui définissent un modèle pour l'ensemble des fonctionnalités partagées par les SGBD. Par exemple la classe `DbConnection` définit la notion de connexion vers une base de données tandis que la classe `DbCommand` définit la notion de commande vers une base de données. Notez qu'ADO.NET 1.x représentait cet ensemble de fonctionnalités partagées au moyen d'interfaces telles que `IDbConnection` ou `IDbCommand`. Ces interfaces sont toujours présentées par le *framework* et les nouvelles classes abstraites correspondantes les implémentent. Cependant, les nouvelles fonctionnalités partagées supportées par ADO.NET2 ne sont disponibles qu'au travers des classes abstraites aussi, il est conseillé de les préférer aux interfaces.

Lorsqu'une application exploite des données stockées dans un SGBD il est intéressant de chercher à rendre votre code le plus indépendant possible (le plus abstrait possible) d'un type de SGBD. L'intérêt de cette démarche est qu'il est courant de ne pas maîtriser le type de SGBD qui sera utilisé en production. Un client peut vouloir utiliser votre application avec un SGDB *Oracle* parce qu'il ne veut pas changer ses habitudes tandis qu'un autre client préfèrera un SGBD gratuit tel que *MySQL*. Cette indépendance du type de SGBD est aisément obtenue si vous n'exploitez que les fonctionnalités communes aux SGBD. En effet, il suffit alors de ne manipuler les données qu'au travers des classes abstraites de l'espace de noms `System.Data.Common` dont nous venons de parler. Néanmoins, un problème se pose : à un endroit de votre code, il faut créer les objets ADO.NET que vous manipulez. Cette opération implique de préciser explicitement la classe à utiliser. Par exemple :

*Exemple 19-3 :*

```
using System.Data.Common ;
using System.Data.SqlClient ;
```

```
class Program {
   static void Main() {
      DbConnection cnx ;
      DbCommand    cmd ;
      if( /*test si on travaille avec SQL Server*/ true ){
         cnx = new SqlConnection();// Ici on couple notre application...
         cmd = new SqlCommand() ;   // ...avec le provider SqlClient.
      }
      // Ici on exploite cnx et cmd indépendamment du SGBD sous-jacent.
   }
}
```

Pour limiter les effets de ce problème chaque fournisseur de données ADO.NET2 présente une classe dérivée de la classe abstraite `System.Data.Common.DbProviderFactory`. Cette classe présente les méthodes suivantes :

```
DbConnection CreateConnection() ;
DbCommand CreateCommand() ;
DbCommandBuilder CreateCommandBuilder() ;
DbConnection CreateConnection() ;
DbConnectionStringBuilder CreateConnectionStringBuilder() ;
DbDataAdapter CreateDataAdapter() ;
DbDataSourceEnumerator CreateDataSourceEnumerator() ;
DbParameter CreateParameter() ;
DbPermission CreatePermission() ;
```

À première vue, il faudrait réécrire l'exemple comme ceci :

*Exemple 19-4 :*

```
using System.Data.Common ;
using System.Data.SqlClient ;
class Program {
   static void Main() {
      DbProviderFactory fabrique ;
      // Tester si l'on travaille avec un SGBD type SQL Server.
      if (true)
          fabrique= new SqlClientFactory();
      // À partir d'ici le code est indépendent du provider sous-jacent.
      DbConnection cnx = fabrique.CreateConnection() ;
      DbCommand    cmd = fabrique.CreateCommand() ;
   }
}
```

Cependant cet exemple ne compile pas car la classe `SqlClientFactory` n'a pas de constructeur public. La seule façon de construire une instance d'une classe dérivée de `DbProviderFactory` est d'utiliser la méthode statique `CreateFactory()` de la classe `System.Data.Common.DbProviderFactories`. Cette dernière classe est en fait une fabrique de fabrique. La méthode `CreateFactory()` retourne une fabrique d'objets ADO.NET pour un fournisseur de données si vous lui passez en argument l'espace de nom du fournisseur de données sous forme d'une chaîne de caractères. Notre exemple se réécrit donc comme ceci :

*Exemple 19-5 :*

```
using System.Data.Common ;
// Nous n'avons plus besoin du namespace System.Data.SqlClient !!
class Program {
   static void Main() {
      DbProviderFactory fabrique =
              DbProviderFactories.GetFactory("System.Data.SqlClient");
      // À partir d'ici le code est indépendent du provider sous-jacent.
      DbConnection cnx = fabrique.CreateConnection() ;
      DbCommand    cmd = fabrique.CreateCommand() ;
   }
}
```

Pour que notre exemple soit à 100% indépendant d'un fournisseur de données il ne nous reste plus qu'à obtenir la chaîne de caractères contenant l'espace de noms à partir d'un fichier de configuration. Ceci est prévu par l'attribut XML `providerName` d'un élément XML `add` de l'élément XML `connectionStrings` du fichier de configuration de votre application. Par exemple le fichier de configuration de votre application peut ressembler à ceci :

*Exemple 19-6 :* *app.exe.config*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <connectionStrings>
      <add name="Ma DB" providerName="System.Data.SqlClient"
           connectionString=
      "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION"/>
   </connectionStrings>
</configuration>
```

Le code de votre application ressemblera alors à ceci. Il est bien 100% indépendant de tous les fournisseurs de données :

*Exemple 19-7 :* *app.cs*

```
using System.Data.Common ;
using System.Configuration ;
class Program {
   static void Main() {
      ConnectionStringSettings cfg =
         ConfigurationManager.ConnectionStrings["Ma DB"];
      DbProviderFactory fabrique =
         DbProviderFactories.GetFactory(cfg.ProviderName) ;
      DbConnection cnx ;
      DbCommand    cmd ;
      cnx = fabrique.CreateConnection() ;
      cmd = fabrique.CreateCommand() ;
      cnx.ConnectionString = cfg.ConnectionString;
      cmd.Connection = cnx ;
   }
}
```

L'architecture présentée par cet ensemble de classes est plus connue sous la forme du *design pattern* nommé *fabrique abstraite* (*abstract factory* en anglais). Cette architecture ne vous empêche pas de ponctuellement tenter d'utiliser les services propres à un fournisseur de données particulier. Par exemple le programme suivant montre comment obtenir les statistiques d'une connexion à condition que l'on utilise le fournisseur de données de *SQL Server* (seul ce fournisseur de données expose cette possibilité) :

*Exemple 19-8 :* *app.cs*

```
using System.Data.Common ;
using System.Configuration ;
using System.Collections ;  // Pour IDictionary.
using System.Data.SqlClient ;
class Program {
   static void Main() {
      ConnectionStringSettings cfg =
         ConfigurationManager.ConnectionStrings["Ma DB"] ;
      DbProviderFactory fabrique =
         DbProviderFactories.GetFactory(cfg.ProviderName) ;
      DbConnection cnx = fabrique.CreateConnection() ;
      IDictionary cnxStats = EventuallyGetStats(cnx) ;
   }
   public static IDictionary EventuallyGetStats(DbConnection cnx) {
      if ( cnx is SqlConnection )
         return ( cnx as SqlConnection ).RetrieveStatistics();
      return null ;
   }
}
```

La liste des fournisseurs de données est extensible. Aussi, chaque fournisseur de données disponible sur une machine doit se déclarer dans le fichier `machine.config` comme ceci :

*Exemple 19-9 :* *app.exe.config*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  ...
  <system.data>
    <DbProviderFactories>
      ...
      <add name="OleDb Data Provider"
           invariant="System.Data.OleDb"
           support="BF"
           description=".Net Framework Data Provider for OleDb"
           type="System.Data.OleDb.OleDbFactory, System.Data,
                 Version=2.0.3600.0, Culture=neutral,
                 PublicKeyToken=b77a5c561934e089" />
    ...
    </DbProviderFactories>
  </system.data>
</configuration >
```

Notez qu'un fournisseur de données spécifique à une application web peut être déclaré dans le fichier `web.config` correspondant.

La chaîne de caractères spécifiée par l'attribut `invariant` est la même que celle spécifiée en argument de la méthode `DbProviderFactory.GetFactory()`. L'attribut `support` définit un masque de 8 bits relatif à l'indicateur binaire `System.Data.Common.DbProviderSupportedClasses` spécifiant les possibilités présentées par le fournisseur de données. Enfin l'attribut `type` indique le nom de la classe « fabrique » dérivant de `DbProviderFactory` ainsi que l'assemblage qui la contient. Cet assemblage doit être soit dans le répertoire de l'application soit dans le répertoire GAC.

## *Chaînes de connexion*

Une *chaîne de connexion* est une chaîne de caractères qui contient les informations nécessaires pour localiser et pour se connecter à une base de données. Une chaîne de connexion est constituée de plusieurs couples clé/valeur, séparés par des points virgules. Par exemple :

```
string s1 ="Data Source=Server;Database=MyDB;User ID=FOOID;Password=FOOPWD;" ;
string s2 ="Data Source=Server ; " +
           "Initial Catalog=pubs;Integrated Security=SSPI;" ;
string s3 ="SERVER = localhost ; UID=sa ; PWD = ; database = ORGANISATION" ;
```

Certaines données sont nécessairement représentées dans une chaîne de connexion. On peut citer le nom du serveur hébergeant la base de données ou le login de l'utilisateur DB sous lequel on souhaite se connecter. Les clés associées à ces données diffèrent selon le fournisseur de données. Par exemple, le fournisseur de données OleDb se sert des clés `"Data Source"` et `"User ID"` tandis que le fournisseur de données Odbc se sert des clés `"Server"` et `"UID"`. Le fournisseur de données SqlClient supporte ces deux syntaxes. Certains fournisseurs de données présentent la possibilité de construire une chaîne de connexion au moyen d'une classe dérivée de `System.Data.Common.DbConnectionStringBuilder`. Cette possibilité est illustrée par l'exemple suivant. Cet exemple exploite aussi le fait que le fournisseur de données SqlClient permet d'énumérer les sources de données auxquelles il peut accéder :

*Exemple 19-10 :* *app.cs*

```
using System.Data ;
using System.Data.Common ;
using System.Configuration ;
class Program {
   static void Main() {
      Configuration cfg = ConfigurationManager.OpenExeConfiguration(
         ConfigurationUserLevel.None) ;
      ConnectionStringSettings cfg_cs =
         cfg.ConnectionStrings.ConnectionStrings["Ma DB"] ;
      DbProviderFactory fabrique =
         DbProviderFactories.GetFactory(cfg_cs.ProviderName) ;
      DbDataSourceEnumerator e=fabrique.CreateDataSourceEnumerator();
      DataTable tbl = e.GetDataSources();

      if (tbl.Rows.Count > 0) {
         // Laisser l'utilisateur choisir un serveur.
```

```
            int choixUtilisateur = 0 ;
            DataRow row = tbl.Rows[choixUtilisateur] ;
            string dataSource = row[ "ServerName" ].ToString() ;
            string instanceName = row[ "InstanceName" ].ToString() ;
            string isClustered = row[ "IsClustered" ].ToString() ;
            string version = row[ "Version" ].ToString();

            // Met à jour la chaîne de cnx dans le fichier de config
            // indépendamment du fournisseur de données sous-jacent.
            DbConnectionStringBuilder csBuilder =
               fabrique.CreateConnectionStringBuilder();
            csBuilder.ConnectionString = cfg_cs.ConnectionString;
            if( csBuilder.ContainsKey("Server"))
               csBuilder.Remove("Server");
            csBuilder.Add("Server", dataSource) ;
            cfg_cs.ConnectionString = csBuilder.ConnectionString ;
            cfg.Save() ;
         } // end choix de la ligne.
      } // end méthode Main()
   } // end class Program.
```

Cet exemple est particulièrement pertinent si vous utilisez la clé `"Data Source"` dans votre chaîne de connexion dans votre fichier de configuration. En effet, la valeur associée sera correctement mise à jour bien que nous utilisions la chaîne de caractères `"Server"` pour la référencer dans notre programme.

Si vous ne souhaitez pas utiliser cette possibilité de construction dynamique de chaînes de connexion nous vous conseillons de vous référer au site `http://www.connectionstrings.com/`. Il réunit un grand nombre d'exemples de chaînes de connexion présentés en fonction des sources de données disponibles sur le marché des SGBD.

Lorsque vous utilisez le fournisseur de données ODBC il est conseillé d'exploiter la notion de *DSN* (*Data Source Name*) qui évite aux développeurs de manipuler les données contenues dans la chaîne de connexion. Un DSN est une information stockée dans le système d'exploitation de la machine qui exécute l'application. Cette information associe une base de données à une chaîne de caractères. Par exemple une chaîne de connexion utilisant un DSN peut ressembler à ceci :

```
string s1 = "DSN=MonAppliDSN" ;
```

En plus de sa simplicité d'utilisation, un DSN présente l'avantage de constituer une indirection vers une base de données paramétrable après compilation de l'application. Cette indirection se fait indépendamment du SGBD sous-jacent et indépendamment de l'emplacement physique de la base, puisque ces données sont des paramètres du DSN. En général on configure les DSN présents dans un système d'exploitation *Microsoft* avec le menu *Panneau de configuration* ▶ *Outil d'administration* ▶ *Sources de données (ODBC)*.

## *Stocker les chaînes de connexion*

Les chaînes de connexion ont la possibilité d'être stockées dans un fichier de configuration. Cette notion de fichier de configuration fait l'objet de la section page 58.

L'Exemple 19-6 illustre un fichier de configuration qui définit une chaine de connexion dans son élément `<connexionStrings>`. L'Exemple 19-7 montre comment du code C# d'une application peut avoir accès aux chaînes de connexion définies dans un fichier de configuration.

Les chaînes de connexion sont parfois considérées comme des données confidentielles du fait qu'elles contiennent souvent un mot de passe. Le *framework* .NET 2.0 vous permet de sauver une version encryptée d'une chaine de connexion dans un fichier de configuration. Ceci est expliqué en page .

## *Pool de connexions*

Ceux qui ont déjà écrit des serveurs accédant à des bases de données savent que l'utilisation d'un *pool de connexions* peut augmenter d'une manière significative les performances. Un pool de connexions est une collection de connexions équivalentes. C'est-à-dire que les connexions sont connectées à la même base et que chaque demande d'un client se sert d'une de ces connexions pour accéder aux données. Les connexions sont des objets complexes, coûteux à initialiser et à détruire. L'idée sous-jacente à la notion de pool de connexions est de recycler les connexions déjà ouvertes de façon à diminuer globalement les coûts de créations, d'initialisation et de destruction des connexions.

La bonne nouvelle est que ADO.NET fait en sorte que ce mécanisme de pooling de connexions soit complètement transparent pour l'utilisateur. Un pool de connexions est automatiquement créé en interne pour chaque chaîne de connexion utilisée. Par exemple :

```
...
string sCnx = "server=localhost ;uid=sa ;pwd =;database = ORGANISATION";
SqlConnection cnx1 = new SqlConnection(sCnx) ;
cnx1.Open() ;
// Ici le pool de connexion associé à la chaine sCnx ne contient pas de
// connexion mais le thread courant en utilise une.
cnx1.Close() ;
// La connexion n'est pas détruite mais est placée dans le pool associé
// à la chaine sCnx.
...
```

La gestion en interne du pool de connexions dépend du fournisseur de données sous-jacent. Aussi nous vous conseillons de vous référez à la documentation officielle du fournisseur si vous souhaitez paramétrer cette gestion. Sachez que le fournisseur de données SqlClient présente des possibilités pour permettre programmatiquement un certain contrôle sur la façon dont le *pooling* de connexions d'effectue. Plus d'informations à ce sujet sont disponibles dans l'article **Connection Pooling for the .NET Framework Data Provider for SQL Server** des **MSDN**.

## *Obtenir les métadonnées d'une base de données*

ADO.NET 2.0 présente un *framework* permettant de naviguer programmatiquement dans le schéma d'un SGBD. Par schéma, nous entendons l'ensemble des métadonnées du SGBD telles que les ensembles des tables, des colonnes, des vues, des utilisateurs, des clés étrangères, des index etc. Typiquement, un tel *framework* permet de développer des outils qui présentent la structure d'un SGBD. *Visual Studio 2005* contient un tel outil illustré par le panneau gauche de la Figure 19-3 (page 732). On y voit la structure du SGBD exposée sous une forme hiérarchique.

Toute la problématique d'un tel *framework* est qu'il est générique dans le sens où il est commun à tous les types de SGBD et donc à tous les fournisseurs de données. Or, les différents types de SGBD n'ont pas exactement la même structure. Bien entendu, on y retrouve dans chacun les concepts classiques tels que les notions de tables ou d'index. Mais les détails sont en général différents. Par exemple, il existe trois types de filtre (aussi nommé restriction) qui peuvent s'appliquer à une requête sur l'ensemble des tables d'un SGBD type *Oracle* (`OWNER`, `TABLE_NAME`, et `TYPE`) tandis qu'il en existe quatre pour les SGBD de type *SQL Server* (`table_catalog`, `table_schema`, `table_name` et `table_type`).

La réponse à cette problématique est forcément un affaiblissement du typage. Les requêtes sur la structure d'un SGBD se font d'une manière non typées à l'aide d'une seule méthode, la méthode `DataTable GetSchema()` définie dans la classe de base `DbConnection`. L'ensemble des éléments structuraux du SGBD satisfaisant à une requête est contenu dans une `DataTable`. Si vous utilisez cette méthode sans paramètre vous obtiendrez ce que l'on nomme l'ensemble des collections de métadonnées. Pour un SGBD de type *SQL Server* cet ensemble est : `MetaDataCollections`, `DataSourceInformation`, `DataTypes`, `Restrictions`, `ReservedWords`, `Users`, `Databases`, `Tables`, `Columns`, `Views`, `ViewColumns`, `ProcedureParameters`, `Procedures`, `ForeignKeys`, `IndexColumns`, `Indexes`, et `UserDefinedTypes`. Si vous utilisez cette méthode avec un paramètre de type chaîne de caractères égale à une de ces collections, vous obtiendrez l'ensemble des éléments de cette collection. L'exemple suivant illustre ceci en énumérant l'ensemble des tables d'une SGBD :

*Exemple 19-11 :*

```
using System.Data.Common ;
using System.Data.SqlClient ;
using System.Data ;
class Program {
   static void Main() {
      string sCnx =
        "server = localhost ; uid=sa ; pwd= ; database = ORGANISATION" ;
      using (DbConnection cnx = new SqlConnection(sCnx)) {
         cnx.Open() ;
         DataTable tbl = cnx.GetSchema("tables");
         foreach (DataRow row in tbl.Rows) {
            foreach (DataColumn col in tbl.Columns)
               System.Console.WriteLine(col.ToString() + " = " +
                                 row[col].ToString()) ;
            System.Console.WriteLine() ;
         }
      } // end using cnx.
   }
}
```

Voici un extrait de l'affichage de cet exemple :

```
...
TABLE_CATALOG = ORGANISATION
TABLE_SCHEMA = dbo
TABLE_NAME = EMPLOYES
TABLE_TYPE = BASE TABLE
```

```
TABLE_CATALOG = ORGANISATION
TABLE_SCHEMA = dbo
TABLE_NAME = DEPARTEMENTS
TABLE_TYPE = BASE TABLE
...
```

Enfin, une troisième surcharge de la méthode `GetSchema()` existe. Elle vous permet de filtrer le résultat de la requête.

Plus d'informations à ce sujet sont disponibles dans l'article **Schemas in ADO.NET 2.0** des **MSDN** rédigé par *Bob Beauchemin*. La notion de restriction y est détaillée ainsi que les impacts de ce *framework* au niveau de la conception d'un fournisseur de données.

# *Travailler en mode connecté avec des DataReader*

## *Utiliser une implémentation de DataReader pour obtenir des données*

Voici un petit programme en mode console, utilisant la classe `SqlReader` pour récupérer le contenu de la table `EMPLOYES`. Les lignes sont obtenues à partir de la base, une à une par l'appel à la méthode `Read()`. La connexion n'est fermée que lorsque les données ont été traitées :

*Exemple 19-12 :*

```
using System ;
using System.Data.Common ;
using System.Data.SqlClient ;
class Program {
   static void Main() {
      // Chaine de connexion.
      string sCnx =
        "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION" ;
      // Requête SQL pour récupérer des données.
      string sCmd = "SELECT * FROM EMPLOYES" ;
      // Création d'un objet connexion.
      using( DbConnection cnx = new SqlConnection(sCnx) ) {
         // Création d'une commande.
         using(DbCommand cmd=new SqlCommand(sCmd,cnx as SqlConnection)){
            // Ouverture de la connexion.
            cnx.Open();
            // Exécution de la commande.
            using( DbDataReader rdr = cmd.ExecuteReader() ) {
               // Affichage des colonnes 2 et 3 de la table EMPLOYES.
               while( rdr.Read() )
                  Console.WriteLine(rdr.GetString(2) + " " +
                                    rdr.GetString(3));
            } // end using rdr.
         } // end using cmd.
      } // end using cnx.
```

```
   }
}
```

Cet exemple affiche sur la console :

```
Lafleur Léon
Dupont Anne
Schmol Jean
Gripsou  Noël
```

La plupart des classes de ADO.NET implémentent l'interface `IDispose` et doivent donc soit être instanciées au sein d'une clause `using` soit être disposées manuellement.

La classe `DbProviderFactory` ne présente pas la méthode `CreateDbDataReader()` bien que chaque fournisseur de données ait une classe dérivée de `DbDataReader`. La raison est que les nouvelles instances de ces classes sont nécessairement obtenues en retour de l'appel à la méthode `ExecuteReader()` sur une commande.

## *Obtenir une valeur à partir d'une base de données*

Il arrive qu'une application ait besoin d'une simple valeur calculée à partir des informations contenues dans une base de données. Par exemple on peut souhaiter compter le nombre de lignes d'une table (`COUNT`), calculer la somme (`SUM`), la moyenne (`AVG`), le minimum (`MIN`), le maximum (`MAX`), la *variance* (`VAR`, `VARP`) ou l'*écart type* (`STDEV`, `STDEVP` pour *standard deviation* en anglais) des valeurs d'une ou de plusieurs colonnes, d'une ou de plusieurs tables. Ce serait un gros gaspillage de ressources que de récupérer la table entière ou même la colonne entière, afin d'effectuer le calcul au niveau de votre code. Aussi, le langage SQL peut réaliser ces calculs directement au niveau du SGBD grâce aux mots-clés cités, utilisés dans une requête. Voici un exemple d'utilisation du mot-clé `COUNT`, qui permet de récupérer le nombre de lignes de la table `EMPLOYES` :

*Exemple 19-13 :*

```
using System ;
using System.Data.Common ;
using System.Data.SqlClient ;
class Program {
   static void Main() {
      // Requête SQL pour récupérer le scalaire.
      string sCmd = "SELECT COUNT(*) FROM EMPLOYES";
      int nEmployes ;
      using( DbConnection cnx = new SqlConnection(
       "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION")){
         cnx.Open() ;
         using( DbCommand cmd =
                   new SqlCommand(sCmd,cnx as SqlConnection) ) {
            nEmployes = (int) cmd.ExecuteScalar();
         } // end using cmd.
      } // end using cnx.
      Console.WriteLine("Nombre d'employés:{0}", nEmployes) ;
   }
}
```

## *Utiliser des requêtes SQL pour obtenir et modifier des données*

Les commandes que l'on peut exécuter sur une base de données sont de trois types.

- Une commande/requête SQL.
- Une *procédure stockée* qui associe un nom et des paramètres à une requête SQL stockée directement par le SGBD. Comprenez bien qu'architecturalement parlant, utiliser des procédures stockées permet de déplacer de la complexité de votre code source vers le SGBD. La proximité d'un traitement dans une procédure stockée avec les données elles-mêmes est un avantage. Cependant, l'utilisation de procédures stockées complexifie en général la maintenance globale d'une application, notamment parce que plusieurs langages de programmation sont alors utilisés (T-SQL, C# etc).
- Une commande ne prenant en compte que le nom d'une ou de plusieurs tables. Dans ce cas, le contenu entier des tables est récupéré par la commande.

Ces types de commandes peuvent être définis par la propriété `CommandType` de l'interface `IDbCommand`. La première solution est choisie par défaut. Il existe principalement quatre types de commandes/requêtes SQL :

- `SELECT` : permet d'obtenir des données. Ce type de requête est généralement utilisé pour remplir un `DataSet` à partir d'une commande et d'un `DataAdapter`, comme vu précédemment.
- `INSERT` : permet d'insérer une ligne dans une table, par exemple :

```
INSERT INTO DEPARTEMENTS VALUES ('COM','Communication')
```

- `UPDATE` : permet de modifier les valeurs d'une ou de plusieurs lignes dans une table, par exemple :

```
UPDATE DEPARTEMENTS SET Departement='Comm' WHERE DepID = 'COM'
```

- `DELETE` : permet de supprimer une ou plusieurs lignes dans une table, par exemple :

```
DELETE FROM DEPARTEMENTS WHERE DepID = 'COM'
```

Pour exécuter une telle commande il suffit d'utiliser la méthode `ExecuteNonQuery()` de l'interface `IDbCommand` (notez que *non query* signifie que l'on exécute pas une requête qui va nous retourner des informations) :

*Exemple 19-14 :*

```
using System.Data.Common ;
using System.Data.SqlClient ;
class Program {
   static void Main() {
      // Requête SQL pour insérer un enregistrement.
      string sCmd1 =
       "INSERT INTO DEPARTEMENTS VALUES ('COM','Communication')" ;

      // Requête SQL pour modifier un enregistrement.
      string sCmd2 =
```

```
        "UPDATE DEPARTEMENTS SET Departement='Comm' WHERE DepID = 'COM'" ;

      // Requête SQL pour supprimer un enregistrement.
      string sCmd3 = "DELETE FROM DEPARTEMENTS WHERE DepID = 'COM'" ;

      using ( DbConnection cnx = new SqlConnection(
       "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION")){
         cnx.Open() ;
         DbCommand cmd = new SqlCommand(sCmd1, (SqlConnection)cnx) ;
         int nLignesAffectees = (int)cmd.ExecuteNonQuery() ;
         cmd.CommandText = sCmd2 ;
         nLignesAffectees = (int)cmd.ExecuteNonQuery() ;
         cmd.CommandText = sCmd3 ;
         nLignesAffectees = (int)cmd.ExecuteNonQuery() ;
      } // end using cnx.
   }
}
```

Un peu plus loin dans ce chapitre nous insèrerons un employé dans la table `EMPLOYES`. La commande SQL sera celle-ci :

```
INSERT INTO EMPLOYES VALUES ('COM','Smith','Adam','0497112239')
```

Nous n'initialisons pas le champ `EmployeID` de la ligne insérée dans la table `EMPLOYES`, puisque la valeur de ce champ est calculée par le SGBD. La technique utilisée pour récupérer cette valeur après une insertion dépend du fournisseur de données. Cette technique est décrite dans l'article **Retrieving Identity or Autonumber Values** des **MSDN**.

# *Travailler en mode déconnecté avec des DataSet*

Les exemples de cette section utilisent la base de données présentée page 710.

## *Remplir un cache avec des données d'une base*

Voici un petit programme en mode console, utilisant les classes `SqlDataAdapter` et `DataSet` pour récupérer le contenu de la table `EMPLOYES`. On voit que le `DataAdapter` représente un ensemble de commandes couplées avec une connexion qui est utilisé pour remplir le `DataSet`. Les données sont traitées alors que l'application n'est plus connectée à la base. Le `DataSet` fait alors office de cache déconnecté de données. Le traitement des données consiste en l'affichage des couples nom/prénom sur la console :

*Exemple 19-15 :*

```
using System ;
using System.Data ;
using System.Data.Common ;
using System.Data.SqlClient ;
class Program {
   static void FillWithData(DataSet dSet) {
      // Création d'un DataAdapter pour accéder à la base de données.
```

```
      // On lui fournit la requête SQL et la chaîne de connexion.
      using ( DbDataAdapter dAdapter = new SqlDataAdapter(
       "SELECT * FROM EMPLOYES",
       "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION")){

         // Cet appel à la méthode Fill() provoque :
         //  - L'ouverture d'une connexion avec la base de données.
         //  - L'exécution de la requête SQL.
         //  - La création d'une table EMPLOYES dans le DataSet.
         //  - Le remplissage de la table EMPLOYES du DataSet
         //    avec les données récupérées par la requête SQL.
         dAdapter.Fill(dSet);
      } // end using dAdapter, provoque la déconnection.
   }
   static void Main() {
      DataSet dSet = new DataSet() ;
      FillWithData(dSet) ;
      // Affichage des colonnes Nom et Prénom de la table EMPLOYES.
      // L'index de la table est forcément 0 puisque nous n'avons
      // récupéré qu'une seule table.
      DataTable dTable = dSet.Tables[0];
      foreach (DataRow dRow in dTable.Rows)
         Console.WriteLine(dRow["Nom"] + " " + dRow["Prénom"]);
   }
}
```

Cet exemple affiche sur la console :

```
Lafleur Léon
Dupont Anne
Schmol Jean
Gripsou Noël
```

## *Travailler avec des relations entre les tables d'un DataSet*

Une instance de la classe `DataSet` peut contenir une ou plusieurs tables. Vous avez la possibilité d'ajouter des relations entre les tables contenues dans un même `DataSet`. Une relation se fait entre une colonne d'une table et une colonne d'une autre table. Une telle relation est utilisée en général pour modéliser une relation *« one to many »* entre les lignes d'une table parent (i.e la table dont la colonne dans la relation constitue sa clé primaire) et les lignes d'une table fille (i.e la table dont la colonne dans la relation constitue une clé étrangère). Dans l'exemple suivant, on met en relation la colonne `DepID` de la table `DEPARTEMENTS` et la colonne `DepID` de la table `EMPLOYES`. En effet, il y a bien une relation *« one to many »* entre un département et les employés du département puisque chaque employé a un seul département et un département peut contenir zéro, un ou plusieurs employés. La création d'une telle relation dans le `DataSet` permet de parcourir logiquement les données :

*Exemple 19-16 :*

```
using System ;
using System.Data ;
using System.Data.SqlClient ;
class Program {
   static void Main() {
      DataSet dSet = new DataSet() ;
      using ( SqlConnection cnx = new SqlConnection(
       "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION")){
         using (SqlDataAdapter dAdapter = new SqlDataAdapter()) {
            dAdapter.SelectCommand = new SqlCommand(
               "SELECT * FROM EMPLOYES", cnx) ;
            dAdapter.Fill(dSet, "EMPLOYES") ;
            dAdapter.SelectCommand = new SqlCommand(
               "SELECT * FROM DEPARTEMENTS", cnx) ;
            dAdapter.Fill(dSet, "DEPARTEMENTS") ;
         } // end using cnx.
      } // end using dAdapter.
      // Crée une relation dans le DataSet.
      // La relation se fait entre la colonne DepID de la table
      // DEPARTEMENTS et la colonne DepID de la table EMPLOYES.
      DataColumn dCol1 = dSet.Tables["DEPARTEMENTS"].Columns["DepID"] ;
      DataColumn dCol2 = dSet.Tables["EMPLOYES"].Columns["DepID"] ;
      DataRelation dRelation = dSet.Relations.Add(
               "Employés du département", dCol1, dCol2);

      // Navigue logiquement dans la relation 'one to many'
      // d'un département vers ses employés.
      foreach (DataRow dRow1 in dSet.Tables["DEPARTEMENTS"].Rows){
         Console.WriteLine("Département : {0}", dRow1["Departement"]) ;
         foreach (DataRow dRow2 in dRow1.GetChildRows(dRelation) )
            Console.WriteLine("  {0} {1}",dRow2["Nom"],dRow2["Prénom"]) ;
      }
   }
}
```

Ce petit programme affiche :

```
Département : Développement
   Dupont Anne
   Schmol Jean
Département : Financier
   Gripsou Noël
Département : Marketing
   Lafleur Léon
```

## *Sauver les données modifiées d'un cache*

Lorsque vous modifiez les données dans une `DataTable` (contenue dans un `DataSet` ou non) les données originales sont conservées. Ainsi, en interne, une `DataTable` maintient les don-

nées courantes et les données originales. En tant que client d'une DataTable vous n'avez accès qu'aux données courantes. Cependant chaque ligne d'une table (i.e chaque instance de la classe DataRow) présente la propriété DataRowState RowState{get;} qui indique si ses données ont subi des changements :

| Valeur de l'énumération DataRowState | Description |
|---|---|
| Unchanged | La ligne n'a pas été modifiée. |
| Added | La ligne a été ajoutée à la table. |
| Deleted | La ligne a été supprimée de la table. |
| Modified | Les données contenues dans la ligne ont été modifiées. |
| Detached | La ligne vient d'être créée mais ne fait pas encore partie d'une table. |

Il serait plus juste de rajouter pour chacune de ces descriptions, « *depuis le dernier appel à la méthode AcceptChanges() ou RejectChanges() sur la DataTable contenant la ligne* ». En effet, si vous appelez l'une ou l'autre de ces méthodes au nom éloquent, l'état de chaque ligne est positionné à Unchanged. De plus, si l'on appelle AcceptChanges(), les lignes qui étaient dans l'état Deleted sont effectivement détruites. Si l'on appelle RejectChanges(), les lignes qui étaient dans l'état Added sont détruites et les données qui ont été modifiées sont remises à leurs états initiaux.

L'appel à la méthode AcceptChanges() sur un DataSet ou une DataTable ne signifie aucunement qu'il y a un accès à la base de données. N'oubliez pas qu'un DataSet ou une DataTable est un cache de données déconnecté. Cependant, si un cache contient des données extraites d'une base, avant d'accepter les changements, il faut en général mettre à jour les données dans la base. Nous allons voir que les opérations de mise à jour de la base de données et d'acceptation des changements dans un cache sont deux opérations distinctes, qui nécessitent chacune du code.

Pour mettre à jour dans une base les données modifiées dans un cache il faut se reconnecter à la base puis effectuer une requête SQL pour chaque modification. Pour vous évitez d'avoir à coder cette opération fastidieuse le fournisseur de donnée sous-jacent a la possibilité de présenter une classe qui est capable de fabriquer ces requêtes SQL en les déduisant d'une requête SELECT. Cette classe doit dériver de System.Data.Common.DbCommandBuilder. Tout ceci est illustré par l'exemple suivant :

*Exemple 19-17 :*

```
using System.Data ;
using System.Data.SqlClient ;
class Program {
   static string sCnx =
        "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION" ;
   static string sCmd = "SELECT * FROM DEPARTEMENTS" ;
   static void Main() {
      DataTable dTable = new DataTable() ;
```

```
      FillTableFromDB(dTable);
      ChangeDataInTable(dTable);
      SynchronizeChangesWithDB(dTable);
   }
   static void ChangeDataInTable(DataTable dTable) {
      // Ajoute un nouveau département.
      dTable.Rows.Add("COM","Communication");
   }
   static void FillTableFromDB(DataTable dTable) {
      using ( SqlDataAdapter dAdapter = new SqlDataAdapter(sCmd, sCnx)){
         dAdapter.Fill(dTable) ;
      } // end using dAdapter.
   }
   static void SynchronizeChangesWithDB(DataTable dTable) {
      using ( SqlDataAdapter dAdapter = new SqlDataAdapter(sCmd, sCnx)){
         // Construit les commandes de mise à jour dans dAdapter...
         // ...à partir de la commande Select.
         SqlCommandBuilder cmdBuilder = new SqlCommandBuilder(dAdapter);
         try {
            // Met à jour les données dans la base.
            dAdapter.Update(dTable);
         }
         catch {
            // Ici, traiter l'erreur de maj.
            return ;
         }
         // Accepte les changements dans le cache.
         dTable.AcceptChanges();
      } // end using dAdapter.
   }
}
```

Sachez que ce principe de conservation en interne des données originales et des données courantes afin de faciliter la mise à jour se retrouve dans un *design pattern* de *Martin Fowler* nommé *Unit Of Work*.

Tel que ce programme est écrit, l'appel à la méthode `Update()` provoque un accès à la base pour chaque requête SQL. Cette technique est peu efficace si vous avez à sauver un grand nombre de changements aussi, depuis ADO.NET2, les classes dérivées de `DbDataAdapter` ont la possibilité de mettre à jour en un seul accès à la base tout un lot de changements. Pour cela, il suffit de positionner la propriété `int DbDataAdapter.UpdateBatchSize{get;set;}` qui spécifie le nombre de requêtes SQL contenues dans un lot.

Enfin, notez que l'opération de mise à jour est susceptible de lancer une exception. La raison est qu'ADO.NET utilise une gestion optimiste des accès concurrents aux données d'une base.

## *Mode déconnecté et gestion optimiste des accès concurrents*

Lorsque plusieurs utilisateurs travaillent simultanément en mode déconnecté avec les mêmes bases de données, il y a possibilité de conflit lors de la modification des données. Par exemple

supposons que l'utilisateur A récupère le numéro de téléphone de Anne Dupont à 8h00. Supposons que l'utilisateur B modifie le numéro de téléphone de Anne Dupont à 8h05. Enfin, supposons que l'utilisateur A décide de changer aussi le numéro de téléphone de Anne Dupont à 8h10, à partir du numéro qu'il a récupéré 10 minutes plus tôt. Il y a clairement un conflit de mise à jour du numéro de téléphone de Anne Dupont. Quel est finalement le numéro de Anne Dupont : celui que l'utilisateur B a sauvé à 8h05 ou celui que l'utilisateur A a sauvé à 8h10 ?

ADO.NET gère ces conflits d'une manière *optimiste*. Cela signifie qu'aucune mesure n'est mise en œuvre pour éviter ces conflits, mais si un conflit survient, il est détecté et par défaut la donnée n'est pas modifiée. Dans notre exemple, le numéro de téléphone garderait donc sa valeur affectée à 8h05 et l'utilisateur A serait averti du conflit à 8h10. La gestion est optimiste dans le sens où on espère que les conflits ne surviennent que très rarement. Lorsqu'un conflit est détecté, une exception de type `System.Data.DBConcurrenyException` est levée lors de l'appel à la méthode `Update()` sur l'adaptateur.

Théoriquement il existe une technique pour prévenir les conflits. Cette technique est appelée gestion *pessimiste* des conflits. Cette technique utilise des systèmes de verrous sur les lignes (voire sur les tables) afin de synchroniser les accès. La gestion est pessimiste dans le sens où l'on prévoit que les conflits arrivent suffisamment souvent pour nuire au bon déroulement de l'application. Ce système de verrous implique des temps d'attentes pour les clients qui souhaitent accéder aux données, et donc une baisse significative des performances générales de l'application.

Il est conseillé aux développeurs utilisant ADO.NET de faire en sorte que les architectures de leurs bases de données et de leurs applications soient adaptées à une gestion optimiste. Cette contrainte est assez faible dans la mesure où il suffit de minimiser les cas où plusieurs utilisateurs accèdent en écriture à la même donnée. Dans la plupart des systèmes distribués réels, cette contrainte est satisfaite *de facto*. En effet, en général le volume des données à traiter est immense par rapport aux nombres d'utilisateurs simultanés. Concrètement, si des dizaines d'utilisateurs utilisent simultanément des dizaines de millions de lignes, les conflits ne surviennent que très rarement.

## *Ajouter des contraintes à une table d'un DataSet*

Vous pouvez associer des *contraintes* à une table contenue dans un `DataSet`. Ces contraintes servent de « garde-fou ». Concrètement, la présence de telles contraintes sur une table provoque l'envoi d'une exception lorsqu'une modification ne respecte pas une des contraintes. Pour que l'exception soit effectivement lancée, il faut que la propriété `EnforceConstraints` du `DataSet` contenant la `DataTable` soit positionnée à `true`.

Chaque contrainte est représentée par une instance d'une classe dérivée de `System.Data.Constraint`. Deux classes dérivent de la classe `Constraint` :

- `System.Data.UniqueConstraint`
  Une contrainte de ce type impose que la table que l'on obtiendrait en projetant la table courante sur certaines de ses colonnes n'ait que des lignes différentes deux à deux. Les colonnes sont précisées dans la contrainte. Ce type de contraintes est particulièrement utile pour vérifier que l'on n'utilise pas une valeur déjà utilisée pour clé primaire, lorsque l'on ajoute une nouvelle ligne dans une table.
- `System.Data.ForeignKeyConstraint`
  Cette contrainte s'utilise sur une table parent lorsqu'une relation « one to many » sur des clés primaires/clés étrangères existe entre une table parent et une table fille. Une telle

contrainte permet de préciser le comportement à adopter lorsqu'une ligne de la table parent voit sa clé primaire modifiée ou lorsqu'une ligne d'une table parent est détruite. Pour cela, les propriétés `DeleteRule` et `UpdateRule` d'une telle contrainte prennent leurs valeurs dans l'énumération `System.Data.Rule`.

- `Cascade` : les lignes concernées de la table fille sont soit mises à jour avec la nouvelle valeur de la clé primaire de la table parent, soit supprimées si la ligne de la table parent est supprimée. C'est le comportement adopté par défaut.
- `None` : rien n'est fait sur les lignes concernées de la table fille.
- `SetDefault` : chaque valeur d'une ligne concernée de la table fille prend la valeur par défaut précisée dans sa colonne (propriété `DefaultValue` de la classe `Column`).
- `SetNull` : les valeurs correspondantes des lignes concernées de la table fille sont positionnées à nulle.

## *Vue des données avec la classe DataView*

La classe `System.Data.DataView` vous permet de sélectionner partiellement le contenu d'une `DataTable` selon différents critères :

- Vous pouvez décider de ne visualiser que les lignes qui sont dans un certain état en positionnant la propriété `DataRowViewState RowStateFilter{get;set;}`. Par exemple la valeur `DataRowViewState.ModifiedCurrent` vous permet de visualiser le contenu courant de toutes les lignes qui ont été modifiées tandis que la valeur `DataRowViewState.Deleted` vous permet de visualiser le contenu original de toutes les lignes qui ont été supprimées.
- Vous pouvez décider d'appliquer un filtre sur le contenu de certaines colonnes en positionnant la propriété `string RowFilter{get;set;}`. Le formatage de la chaîne de caractères représentant le filtre est similaire au formatage d'une clause `WHERE` en SQL. Plus de détails sur ce formatage sont disponibles dans l'entrée **La propriété DataColumn.Expression** des **MSDN**.
- Vous pouvez décider de trier les lignes filtrées selon l'ordre croissant du contenu des colonnes représentant la clé primaire en positionnant la propriété `bool ApplyDefaultSort{get;set;}`.
- Vous pouvez décider de trier les lignes filtrées selon le contenu de une ou plusieurs colonnes en utilisant la propriété `string Sort{get;set;}`. Le tri se fait par ordre croissant par défaut. Vous devez séparer le nom des colonnes avec une virgule et utiliser les expressions `ASC` ou `DESC` pour spécifier un tri par ordre croissant ou décroissant.

Le contenu d'une `DataView` est dynamique. Autrement dit, il reflète en temps réel les changements apportés sur les données de la `DataTable` sous-jacente. Cette particularité est illustrée par le programme suivant :

*Exemple 19-18 :*

```
using System ;
using System.Data ;
using System.Data.SqlClient ;
class Program {
   static string sCnx =
```

```
      "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION" ;
   static string sCmd = "SELECT * FROM EMPLOYES" ;
   static void Main() {
      DataTable dTable = new DataTable() ;
      using (SqlDataAdapter dAdapter = new SqlDataAdapter(sCmd, sCnx)) {
         dAdapter.Fill(dTable) ;
      } // end using dAdapter.
      DataView dView = new DataView(dTable);
      // Filtre les employés du département développement.
      dView.RowFilter = "DepID='DEV'" ;

      Console.WriteLine("--> Data Table:") ;
      foreach (DataRow dRow in dTable.Rows)
         Console.WriteLine(dRow["DepID"] + " " + dRow["Nom"]) ;

      Console.WriteLine("--> Data View avant mutation de Gripsou:") ;
      for (int i = 0 ; i < dView.Count ; i++)
         Console.WriteLine(dView[i]["DepID"] + " " + dView[i]["Nom"]) ;

      // Mutation de Gripsou au département développement.
      foreach (DataRow dRow in dTable.Rows)
         if (((string)dRow["Nom"]) == "Gripsou")
            dRow["DepID"] = "DEV" ;

      Console.WriteLine("--> Data View aprés mutation de Gripsou:") ;
      for (int i = 0 ; i < dView.Count ; i++)
         Console.WriteLine(dView[i]["DepID"] + " " + dView[i]["Nom"]) ;
   }
}
```

Ce programme affiche ceci :

```
--> Data Table:
MKT Lafleur
DEV Dupont
DEV Schmol
FIN Gripsou
--> Data View avant mutation de Gripsou:
DEV Dupont
DEV Schmol
--> Data View aprés mutation de Gripsou:
DEV Dupont
DEV Schmol
DEV Gripsou
```

À la différence du système de vues fourni par les différents SGBD, la classe `DataView` ne vous permet pas de créer de vue sur une jointure de tables. Elle ne permet pas non plus d'exclure ou d'ajouter des colonnes.

La méthode `DataTable DataView.ToTable()` vous permet de fabriquer une nouvelle `DataTable` avec une copie du contenu présenté par la `DataView`.

Vous pouvez ajouter des lignes à une `DataView` si la propriété `bool AllowNew{get;set;}` est positionnée. Vous pouvez aussi modifier le contenu des lignes existantes si la propriété `bool AllowEdit{get;set;}` est positionnée. Ces modifications ne seront reportées dans la `DataTable` sous-jacente que si la méthode `EndEdit()` de la `DataRowView` concernée est appelée. Plus de détails à ce sujet sont disponibles dans l'article **Modifying Data Using a DataView** des **MSDN**.

Si vous exploitez la possibilité de tri sur une ou plusieurs colonnes vous pouvez utilisez les méthodes `Find()` et `FindRows()` pour récupérer une ou plusieurs lignes dont le vecteur des valeurs des colonnes utilisées pour le tri est égal à un vecteur de valeurs spécifié.

La méthode `DataView DataRowView.CreateChildView(DataRelation)` permet d'obtenir une `DataView` sur les lignes de la `DataTable` enfant lorsque vous disposez d'une relation entre tables dans un `DataSet`. Bien évidemment il faut pour cela que la `DataTable` sous-jacente à la `DataView` contenant la ligne source soit la table parent de la relation.

La classe `System.Data.DataViewManager` permet de gérer plusieurs vues sur plusieurs tables d'un même `DataSet`.

# *DataSet typés*

Maintenant que nous avons présenté la notion de `DataSet` il est temps de montrer que l'on utilise en fait que rarement la classe `DataSet`. En effet, on préfère utiliser ce que l'on nomme des *`DataSet typés`*. L'outil `xsd.exe` fourni avec le *framework* permet de construire une classe C# ou VB.NET dérivée de la classe `DataSet`. Cette construction se fait à partir d'un schéma XSD ou à partir des tables d'une base de données. Cette classe générée présente des sous classes qui permettent d'accéder aux données du cache d'une manière fortement typée. Pour bien illustrer cette notion de typage fort, voici l'Exemple 19-16 réécrit en exploitant la classe de `Dataset` typés `ConsoleApplication1.ORGANISATIONDataSet` :

*Exemple 19-19 :*

```
using System ;
using System.Data ;
using System.Data.Common ;
using System.Data.SqlClient ;
using ConsoleApplication1 ; // Espace de noms contenant la classe
                            // ORGANISATIONDataSet de DataSet typé.
class Program {
   static void Main() {
      ORGANISATIONDataSet dSet = new ORGANISATIONDataSet();
      using (SqlConnection cnx = new SqlConnection(
       "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION")){
         using (SqlDataAdapter dAdapter = new SqlDataAdapter()) {
            dAdapter.SelectCommand = new SqlCommand(
               "SELECT * FROM EMPLOYES", cnx) ;
            dAdapter.Fill(dSet, "EMPLOYES") ;
            dAdapter.SelectCommand = new SqlCommand(
               "SELECT * FROM DEPARTEMENTS", cnx) ;
```

```
            dAdapter.Fill(dSet, "DEPARTEMENTS") ;
         } // end using dAdapter.
      } // end using cnx.
      foreach (ORGANISATIONDataSet.DEPARTEMENTSRow dRow in
                dSet.DEPARTEMENTS) {
         Console.WriteLine("Département : " + dRow.Departement) ;
         foreach (ORGANISATIONDataSet.EMPLOYESRow eRow in
                   dRow.GetEMPLOYESRows())
            Console.WriteLine("   " + eRow.Nom
                              + " " + eRow.DEPARTEMENTSRow.Departement) ;
      }
   }
}
```

Cet exemple affiche ceci :

```
Département : Développement
   Dupont Développement
   Schmol Développement
Département : Financier
   Gripsou Financier
Département : Marketing
   Lafleur Marketing
```

On observe qu'une instance de la classe `ORGANISATIONDataSet` se manipule comme un `DataSet`. Ceci est normal puisque c'est un `DataSet` car la classe `ORGANISATIONDataSet` dérive de la classe `DataSet`. On voit aussi que la classe `ORGANISATIONDataSet` contient les classes encapsulées `EMPLOYESRow` et `DEPARTEMENTSRow` qui sont très pratiques. Non seulement elles présentent une propriété correctement typée pour chaque colonne de la table sous jacente, mais en plus, elles ont des membres permettant de naviguer dans la relation `Est_Employé_Par`. Enfin, on voit que la classe `ORGANISATIONDataSet` présente les propriétés `EMPLOYES` et `DEPARTEMENTS` qui permettent d'accéder directement aux tables.

## *Création d'une classe de DataSet typés*

Comme nous l'avons expliqué, la classe `ORGANISATIONDataSet` ainsi que ses classes encapsulées ont été générées par l'outil `xsd.exe`. Pour éviter de se servir de cet outil en ligne de commande *Visual Studio* présente des menus pour créer des classes de `DataSet` typés :

- Soit vous faite : *Menu Data* ▶ *Add New Data Source* ▶ *Database* ▶ puis vous sélectionnez une base de données ainsi que ses tables qui vont être prises en compte dans votre classe de `DataSet` typé.
- Soit vous faite : *Click droit sur votre projet* ▶ *Add* ▶ *New Item...* ▶ *DataSet* ▶ puis vous sélectionnez une base de données ainsi que ses tables qui vont être prises en compte dans votre classe de `DataSet` typé au moyen de la fenêtre *Server Explorer*.

Dans les deux cas, votre projet contient un nouveau fichier d'extension `.xsd` (en l'occurrence `ORGANISATIONDataSet.xsd`). Un fichier de code source (en l'occurrence `ORGANISATIONDataSet.Designer.cs`) est associé à ce dernier. C'est ce fichier qui contient le code généré de la classe `ORGANISATIONDataSet` et de ses classes encapsulées :

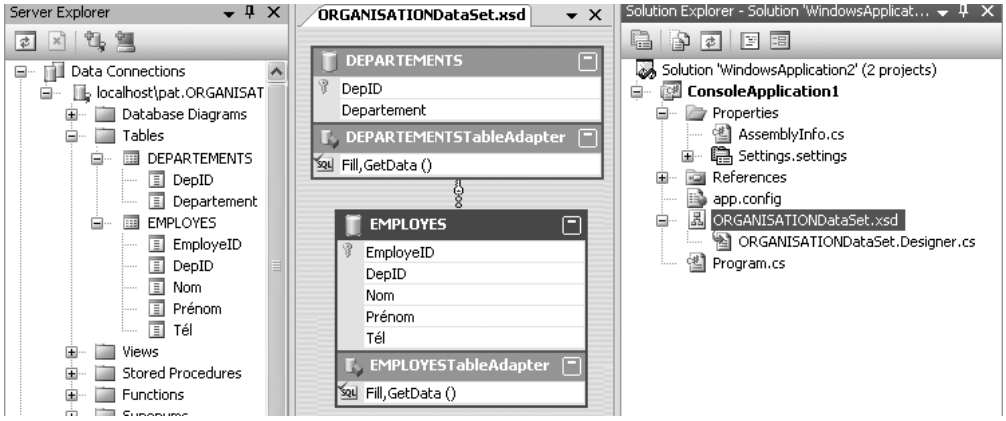


*Figure 19-3 : Visual Studio et les DataSet typés*

Si vous souhaitez ajouter des fonctionnalités à une de ces classes générées, il faut cliquer droit sur le fichier `ORGANISATIONDataSet.xsd` et sélectionner *View Code*. Cette action aura pour effet de créer un nouveau fichier associé nommé `ORGANISATIONDataSet.cs`. Ce fichier contient une déclaration partielle de la classe `ORGANISATIONDataSet`. Vous pouvez alors vous servir de cette déclaration partielle pour ajouter de nouveaux membres à la classe `ORGANISATIONDataSet` ou à ses classes encapsulées. En effet, toutes ces classes générées sont déclarées d'une manière partielle. Ainsi, cette technique vous permet de séparer le code généré et votre propre code dans deux fichiers distincts. Ceci a pour but d'éviter des problèmes de perte de code à cause des mises à jour.

## *TableAdapter et requêtes SQL typées*

Lors de la génération d'un `DataSet` typé, *Visual Studio* génère aussi des classes de `DataAdapter` typés dans un espace de noms dédié (en l'occurrence `ConsoleApplication1.ORGANISATIONDataSetTableAdapters`). On nomme ces classes des `TableAdapter`. Rappelons qu'une instance de `DataAdapter` est un ensemble de commandes couplé avec une connexion permettant de remplir les tables d'un `DataSet`. Réécrivons notre Exemple 19-19 à l'aide des classes `EMPLOYESTableAdapter` et `DEPARTEMENTSTableAdapter` qui ont été générées :

*Exemple 19-20 :*

```
...
// Espace de noms contenant les TableAdapters.
using ConsoleApplication1.ORGANISATIONDataSetTableAdapters;
class Program {
   static void Main() {
      ORGANISATIONDataSet dSet = new ORGANISATIONDataSet() ;
      using (SqlConnection cnx = new SqlConnection(
       "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION")){
         using (EMPLOYESTableAdapter eAdapter = new
                                        EMPLOYESTableAdapter()) {
```

```
            eAdapter.Connection = cnx;
            eAdapter.Fill(dSet.EMPLOYES);
         } // end using eAdapter.
         using (DEPARTEMENTSTableAdapter dAdapter = new
                                    DEPARTEMENTSTableAdapter()) {
            dAdapter.Connection = cnx;
            dAdapter.Fill(dSet.DEPARTEMENTS);
         } // end using dAdapter.
      } // end using cnx.
   ...
```

On remarque l'utilisation des méthodes `Fill(EMPLOYESDataTable)` et `Fill(DEPARTEMENTSDataTable)`. Ces méthodes nous évitent d'avoir à écrire les requêtes SQL de sélection globale des lignes d'une table. Nous aurions pu aussi utiliser les méthodes `EMPLOYESDataTable GetData()` et `DEPARTEMENTSDataTable GetData()` qui jouent le même rôle mis à part qu'elles vous évitent d'avoir à créer la table.

En fait, l'intérêt majeur des `TableAdapter` réside dans le fait qu'ils nous évitent d'avoir à écrire toutes sortes de requêtes SQL (i.e des requêtes de type `SELECT`, `INSERT`, `UPDATE` et `DELETE`). L'idée est de remplacer chaque requête SQL par une méthode d'un `TableAdapter`. L'avantage est que les paramètres entrant et sortant de ces méthodes sont typés. Un *wizard* de création de telle méthode existe. Voici comment s'en servir pour créer une méthode `EMPLOYESDataTable GetEmployesByDepID(string DepID)` qui retourne une table contenant les employés d'un département :

Click droit de `ORGANISATIONDataSet.xsd` ▶ *View designer* ▶ *Click droit sur la zone* `EMPLOYESTableAdapter` ▶ *Add Query* ▶ *Use SQL Statements* ▶ *Next* ▶ *SELECT which returns Rows* (à ce stade vous pouvez aussi choisir de créer une requête de type *INSERT*, *UPDATE*, *DELETE*) ▶ *Next* ▶ *Query Builder* ▶ *Créer la requête :* `SELECT EmployeID, DepID, Nom, Prénom, Tél FROM EMPLOYES WHERE DepID = @DepID` ▶ *OK* ▶ *Next* ▶ *Return a DataTable avec pour nom de méthode* `GetEmployesByDepID()` ▶ *Finish*

Voici un exemple d'un programme qui utilise cette méthode :

*Exemple 19-21 :*

```
using System.Data.SqlClient ;
using ConsoleApplication1 ;
using ConsoleApplication1.ORGANISATIONDataSetTableAdapters ;
class Program {
   static void Main() {
      ORGANISATIONDataSet.EMPLOYESDataTable table ;
      using (SqlConnection cnx = new SqlConnection(
       "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION")){
         using (EMPLOYESTableAdapter eAdapter = new
                                    EMPLOYESTableAdapter()) {
            table = eAdapter.GetEmployesByDepID("DEV");
         } // end using eAdapter.
      } // end using cnx.
      foreach (ORGANISATIONDataSet.EMPLOYESRow eRow in table)
         System.Console.WriteLine("   " + eRow.Nom) ;
```

```
   }
}
```

# *Pont entre le mode connecté et le mode déconnecté*

ADO.NET2 présente un pont entre le mode connecté et le mode non connecté. Concrètement, vous pouvez remplir un `DataSet` ou une `DataTable` à partir d'un `DataReader`. À l'opposé vous pouvez parcourir les données contenues dans un `DataSet` ou dans une `DataTable` à l'aide d'un `DataReader`. Ces deux possibilités sont illustrées par l'exemple suivant qui remplit une `DataTable` avec les données de la table `EMPLOYES` par l'intermédiaire d'un `DataReader` connecté à la base. Dans un second temps les données de la `DataTable` sont lues à l'aide d'un autre `DataReader` non connecté à la base. Tout se passe alors comme si la `DataTable` était une base de données à laquelle était connecté ce deuxième `DataReader` :

*Exemple 19-22 :*

```
using System ;
using System.Data ;
using System.Data.Common ;
using System.Data.SqlClient ;
class Program {
   static void Main() {
      DataTable dTable = new DataTable() ;
      using ( DbConnection cnx = new SqlConnection (
       "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION")){
         using ( DbCommand cmd = new SqlCommand (
             "SELECT * FROM EMPLOYES", (SqlConnection) cnx) ) {
            cnx.Open() ;
            using (DbDataReader rdrCmd = cmd.ExecuteReader()) {
               // DataReader connecté -> DataTable
               dTable.Load(rdrCmd, LoadOption.OverwriteChanges);
            } // end using rdrCmd.
         } // end using cmd.
      } // end using cnx.
      // DataTable -> DataReader non connecté.
      using ( DbDataReader rdrTbl = dTable.CreateDataReader() ) {
         while (rdrTbl.Read())
            Console.WriteLine(rdrTbl.GetString(2) + " " +
                              rdrTbl.GetString(3)) ;
      } // end using rdrTbl.
   }
}
```

La méthode `Load()` de la classe `DataTable` (qui est aussi présentée par la classe `DataSet`) prend en second argument une valeur de l'énumération `LoadOption`. Cette information indique comment résoudre les conflits de mise à jour entre les données originales et courantes contenues dans le `DataSet` et les données en provenance du `DataReader`. L'énumération `LoadOption` présente trois valeurs :

- `PreserveChanges` : (valeur par défaut) Ecrase seulement les données originales du `DataSet` avec les données en provenance du `DataReader`.
- `OverwriteChanges` : Ecrase les données originales et courantes du `DataSet`. avec les données en provenance du `DataReader`.
- `Upsert` : Ecrase seulement les données courantes du `DataSet` avec les données en provenance du `DataReader`.

# *Ponts entre l'objet et le relationnel*

Le modèle relationnel a atteint ses limites avec l'avènement des langages objets. En effet, le modèle relationnel n'est pas adapté à la sauvegarde des états des objets des programmes écrits en langages objets, comme C#, C++ ou Java.

## *La problématique structurelle*

Lorsque l'on planifie d'utiliser un modèle de persistance relationnel à partir d'un langage objet, le premier problème que l'on se pose est celui-ci : il n'existe pas de façon simple pour stocker l'état d'un objet dont la classe est dérivée d'une autre classe :

- Soit on crée une seule table pour toute une hiérarchie de classes (*design pattern* de *Martin Fowler* nommé ***Single Table Inheritance***). Bien que pour une petite hiérarchie de classes cette solution soit efficace, elle devient rapidement impraticable lorsque le nombre de classes et de champs augmente.
- Soit on crée une table par classe non abstraite (*design pattern* de *Martin Fowler* nommé ***Concrete Table Inheritance***). Cette solution a le gros désavantage d'être difficilement maintenable. Si une classe de base change, il faut changer également la structure de toutes les tables représentant les classes dérivées.
- Soit on crée une table pour chaque classe en utilisant un système de clé étrangère (*design patterns* de *Martin Fowler* nommé ***Class Table Inheritance***). Une table ne contient que les champs définis dans la classe correspondante. Cette solution résout le problème de la maintenance, mais introduit de la complexité dans l'accès aux données.

Un autre problème d'ordre structurel est dû à la façon de stocker les relations « *one to many* » et « *many to many* » entre objets. Dans les langages objets, les collections sont physiquement gérées soit par un système de positionnement d'objet (types valeur) soit par un système de référencement d'objet (types référence). Aucun de ces deux systèmes n'est applicable dans une base de données relationnelle. Pour résoudre les cas de relations « *one to many* », on utilise des clés étrangères. Pour résoudre les cas de relations « *many to many* » on utilise une table d'association. Il faut bien comprendre que toute la difficulté du stockage des relations vient de la gestion du passage du système « clé étrangère/table d'association » au système « référence/positionnement » et inversement.

## *La problématique comportementale*

Un autre problème rencontré, lorsque l'on utilise un modèle de persistance relationnel à partir d'un langage objet, est comportemental. Le problème comportemental est relatif à la façon dont les objets sont chargés et sauvegardés. En général, vous chargez plusieurs états dans des objets, vous faites des modifications puis vous sauvez les nouveaux états de ces objets. Il est souvent

efficace de ne charger que partiellement l'état d'un objet bien que cela nécessite plus d'analyse et de conception (*design pattern* de *Martin Fowler* nommé ***Lazy Load***). De plus, durant le traitement des données par l'application, il est assez difficile de garder la trace de chaque changement. Pour cette tâche précise, la technologie ADO.NET propose une solution efficace qui fait l'objet de la section page 725.

## *Trois grands types de solutions*

Il existe trois approches différentes pour traiter ces problématiques : Les bases de données objet, la génération de code et la réflexion sur une table d'association :

- Une nouvelle génération de bases de données, permettant de stocker simplement les états des objets, a vu le jour il y a quelques années. L'idée est simplement de stocker l'état d'un objet soit en le sérialisant, soit en établissant une correspondance avec les données relationnelles pour bénéficier de l'efficacité du modèle de requête relationnel. L'utilisation de ces bases de données dans l'industrie est jusqu'ici restée marginale. Un tournant vient d'être pris par *Microsoft* dans *SQL Serveur 2005*. En effet, à l'exécution, le processus de ce SGBD héberge le CLR de façon à permettre une certaine intégration entre les types gérés .NET et les données des bases. Par exemple, sous certaines conditions, un type .NET peut typer directement les données d'une colonne d'une table (cette fonctionnalité est nommée *User Defined Type* ou *UDT*). En outre, les procédures stockées peuvent être rédigées en code géré .NET.
- Si vous souhaitez rester dans une approche plus traditionnelle, vous pouvez minimiser les conséquences des problèmes de maintenances et de complexités inhérentes au paradigme objet/relationnel avec la génération automatique de code (*design pattern* de *Martin Fowler* nommé ***Metadata Mapping*** **avec génération de code**). Dans cette méthode, le code SQL nécessaire pour construire et exploiter une base de données et la couche logicielle qui attaque la base de données sont construits à partir d'une même description. Il y a moins de problèmes de maintenance, et une grande partie de la complexité peut être encapsulée dans le code généré.
- Une autre façon que la génération de code pour établir un lien entre les objets et les données relationnelles est de prévoir une table de correspondance (i.e de *mapping*) entre les champs des classes et les colonnes de tables. À l'exécution, on se sert d'un moteur de *mapping* pour parcourir une telle table et établir la correspondance entre les états des objets et les données relationnelles. C'est le *design pattern* de Fowler nommé ***Metadata Mapping*** **réflexif**.

Un débat sur les pros et cons des deux variantes du *design pattern* Metadata Mapping est disponible à l'URL `http://www.theserverside.net/news/thread.tss?thread_id=29071`.

Enfin, précisons qu'une des priorité des concepteurs de la version 3.0 de C#, est d'unifier au niveau langage le monde des données (XML et SQL) et le monde des objets. *A priori*, leurs travaux se basent sur le langage expérimental Cw (prononcez C omega) présenté à l'URL `http://research.microsoft.com/Comega/`.

## *Les outils de mapping objet/relationnel*

Il existe plus d'une quarantaine d'outils de mapping objet/relationnel (*mapping* OR) ciblant la plateforme .NET. La plupart se basent sur une ou l'autre version du *design pattern* Metadata Mapping. Certains sont gratuits et open-source alors que d'autres sont payants. Parmi les outils les plus populaires nous pouvons citer :

- NHibernate : http://wiki.nhibernate.org/display/NH/Home
- Data Tier Modeler (DTM) : http://www.evaluant.com/en/solutions/dtm/default.aspx
- OlyMars (pour « Olympique de Marseille » ☺) http://www.microsoft.com/france/msdn/technologies/outils/olymars/default.asp
- Object Broker http://sourceforge.net/projects/objectbroker/

Une liste plus complète est disponible sur le site http://sharptoolbox.com dans la catégorie *Object-Relational mapping*.

*Microsoft* souhaitait fournir un outil de mapping OR avec .NET 2.0 nommé *Object Space*. La livraison de cet outil a été repoussée, à priori parce que le projet a gagné en envergure et sera notamment intégré au futur moteur de gestion de données nommé *WinFS*.

Vous pouvez consulter cet article de *Fabrice Marguerie* qui présente un certain nombre de critères à prendre en compte lors du choix d'un outil de mapping OR http://madgeek.com/Articles/ORMapping/FR/mapping.htm.

# *Fonctionalités spécifiques au fournisseur de données de SQL Server*

## *Requêtes asynchrones*

Le fournisseur de données SqlClient de ADO.NET2 permet d'effectuer des commandes d'une manière asynchrone. La nouveauté par rapport à .NET v1.x est que pendant l'exécution asynchrone d'une commande, aucun thread n'attend le résultat de la base. Cette façon de procéder est différente et plus performante que l'utilisation d'un thread du pool à l'aide d'un délégué asynchrone. En interne cette nouvelle possibilité utilise un mécanisme win32 d'entrée/sortie asynchrone fournit depuis *Windows 2000*. Elle est disponible sur toutes les versions de *SQL Server* supportées par le fournisseur de données SqlClient à savoir 7/2000/2005.

Au niveau *framework*, cette possibilité est exploitable par six nouvelles méthodes de la classe SqlCommand :

```
BeginExecuteNonQuery()  /  EndExecuteNonQuery()
BeginExecuteReader()    /  EndExecuteReader()
BeginExecuteXmlReader() /  EndExecuteXmlReader()
```

Bien qu'en interne le fonctionnement soit différent, ces méthodes s'utilisent pareillement qu'un délégué asynchrone, à savoir, en utilisant un objet accessible par l'interface IAsyncResult pour matérialiser l'appel asynchrone. L'exemple suivant obtient deux DataReader sur notre base ORGANISATION d'une manière asynchrone. Notez qu'une connexion ne peut supporter simultanément plusieurs appels asynchrones aussi nous sommes obligés d'utiliser deux connexions équivalentes. En outre, notez que vous devez spécifier async=true dans la chaîne de connexion des connexions susceptibles d'être utilisées pour effectuer des appels asynchrones. En effet, le support de cette possibilité à un léger impact négatif sur l'exécution synchrone des commandes d'où l'intérêt du drapeau async.

*Exemple 19-23 :*

```
using System.Data ;
using System.Data.SqlClient ;
class Program {
   static string sCnx = "server = localhost ; uid=sa ; pwd = ; " +
                        "database = ORGANISATION ; async=true" ;
   static string sCmd1 = "SELECT * FROM EMPLOYES" ;
   static string sCmd2 = "SELECT * FROM DEPARTEMENTS" ;
   static void Main() {
      using (SqlConnection cnx1 = new SqlConnection(sCnx))
      using (SqlConnection cnx2 = new SqlConnection(sCnx)) {
         cnx1.Open() ;
         SqlCommand cmd1 = new SqlCommand(sCmd1, cnx1) ;
         System.IAsyncResult ar1 = cmd1.BeginExecuteReader() ;
         cnx2.Open() ;
         SqlCommand cmd2 = new SqlCommand(sCmd2, cnx2) ;
         System.IAsyncResult ar2 = cmd2.BeginExecuteReader() ;
         // Ici vous pouvez faire du travail avec le thread courant.
         SqlDataReader rdr1 = cmd1.EndExecuteReader(ar1);
         SqlDataReader rdr2 = cmd2.EndExecuteReader(ar2);
         // Ici utilisation de rdr1 et rdr2
      } // end using cnx1 et cnx2.
   }
}
```

Notez enfin que des vérifications pouvant entraîner une exception lors de l'appel d'une méthode `BeginXxx()` sont effectuées sur les paramètres entrants. En outre l'exécution peut aussi connaître des échecs qui provoqueront l'envoi d'un exception lors de l'appel d'une méthode `EndXxx()`.

## *Copie de données en masse (bulk copy)*

L'outil `bcp.exe` (*Bulk Copy Program, programme de copie en masse* en français) livré avec *SQL Server* permet de copier ou d'obtenir un gros volume de données à partir d'un serveur de données *SQL Server*. Le fournisseur `SqlClient` de ADO.NET2 permet d'utiliser les services de cet outil grâce à la nouvelle classe `System.Data.SqlClient.SqlBulkCopy`. Cette classe permet la copie en masse de données à partir d'une `DataTable`, d'un `DataSet` ou d'un `DataReader` vers une base *SQL Server*. L'exemple suivant montre comment copier en masse le contenu d'une table `DEPARTEMENT` d'une base `ORGANISATION` vers une table `DEPARTEMENT` d'une autre base `ORGANISATION2` :

*Exemple 19-24 :*

```
using System.Data ;
using System.Data.SqlClient ;
class Program {
   static string sCnx =
     "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION" ;
   static string sCnx2 =
     "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION2" ;
   static void Main() {
```

```
      using (SqlConnection cnx = new SqlConnection(sCnx)) {
         SqlCommand cmd = new SqlCommand("SELECT * FROM DEPARTEMENTS ",
                                         cnx) ;
         cnx.Open() ;
         SqlDataReader rdr = cmd.ExecuteReader() ;
         // Copie massive des N_ROWS lignes.
         using (SqlBulkCopy bulkCopier = new SqlBulkCopy(sCnx2)) {
            bulkCopier.DestinationTableName = "DEPARTEMENTS";
            bulkCopier.WriteToServer(rdr);
         } // end using bulkCopier ;
      } // end using cnx.
   }
}
```

La classe `SqlBulkCopy` présente aussi la possibilité d'être notifiée sur l'état courant de la copie, la possibilité de régler un *time out* sur l'opération de copie et la possibilité de définir une correspondance entre les colonnes sources et destinations au cas où les schémas des tables impliquées ne seraient pas strictement identiques.

## *Obtenir les statistiques d'une connexion*

L'Exemple 19-8 en page 715 montre comment obtenir les statistiques concernant l'activité d'une connexion vers un SGBD de type *SQL Server*. L'objet retourné est un dictionnaire dont les clés sont les noms des statistiques. Ce dictionnaire contient une vingtaine de statistiques telles que le nombre d'octets envoyés ou reçus. En plus de la méthode `RetrieveStatistics()`, la classe `SqlConnection` présente la méthode `ResetStatistics()` qui permet de remettre à zéro les statistiques d'une connexion.

## *SQL Server 2005 Express Edition*

Bien que ce produit ne fasse pas partie intégrante ni du *framework* .NET 2.0 ni de la plateforme .NET 2.0, nous le mentionnons car il va faire certainement rencontré un grand succès auprès des développeurs .NET 2.0. *SQL Server 2005 Express Edition* est le successeur des SGBD gratuits et librement distribuables que sont *Microsoft SQL Server Desktop Engine* (MSDE) et *Jet*. Il présente plusieurs avantages par rapport à ces produits :

- Une intégration poussée avec le CLR. *SQL Server Express* est une version simplifiée du SGBD phare (mais payant) de *Microsoft, SQL Server 2005*.
- Déploiement d'application XCopy possible. Une base de donnée peut se résumer à un fichier `.mdf` présent dans l'arborescence de répertoires d'une application .NET déployée.
- L'installation de *SQL Server 2005 Express Edition* peut faire partie des prérequis du déploiement *ClickOnce* d'une application .NET. Pas besoin de l'installation de MDAC.
- Des outils graphiques facilitent l'utilisation du produit.
- Une intégration poussée avec *Visual Studio 2005*. En outre, *SQL Server Express Edition* est automatiquement installé lors de l'installation de *Visual Studio 2005*.
- Support poussé du XML. Possibilité de typer une colonne d'une table en XML.
- Meilleure performance et meilleure gestion de la sécurité.

Plus d'information sur ce produit peuvent être trouvées sur le site de *Microsoft*.

# 20

# *Les transactions*

## *Introduction à la notion de transaction*

Une succession de commandes sur des données est dite *transactionnelle*, si à la fin de ces modifications tous les changements sont validés (*commit*) ou aucun des changements n'est validé (*rollback*). L'exemple standard utilisant une transaction est le transfert d'une somme d'un compte bancaire `A` vers un compte bancaire `B`. Cette opération est le résultat de deux commandes :

- Le débit du compte `A` de la somme.
- Le crédit du compte `B` de la somme.

Si une des deux commandes échoue, l'autre commande doit absolument échouer. Les deux commandes doivent donc être exécutées au sein d'une même transaction.

Les transactions servent à garantir l'intégrité d'un ensemble de données après une phase de mises à jour.

Cet exemple illustre aussi le fait que les commandes exécutées au sein d'une transaction peuvent se faire sur une même base de données ou sur des bases de données différentes. En effet, on peut supposer que les deux comptes appartiennent à la même banque, auquel cas les commandes se font sûrement sur la même base de données. Si les deux comptes appartiennent à des banques différentes, les commandes se font sur des bases de données distinctes. On dit que la transaction est *distribuée*.

La théorie des transactions est assez complexe. Un milieu transactionnel supporte idéalement quatre propriétés, dites *propriétés ACID* :

| | |
|---|---|
| **A**tomicity | Toutes les actions s'effectuent ou aucune. |
| **C**onsistency | La transaction laisse les données dans un état cohérent. |
| **I**solation | Les transactions s'effectuent d'une manière isolée les unes des autres. |
| **D**urable | Les résultats sont stockés de manière permanente. |

Nous allons voir dans cette section quelles sont les possibilités présentées par le *framework* pour effectuer des transactions locales ou distribuées. Nous verrons que l'ensemble des propriétés ACID peut dans certains cas être amoindri pour ne pas trop pénaliser les performances.

## *Moteur transactionnel (TM), Resource Manager (RM) et sources de données*

Les bases de données ne représentent qu'un cas d'utilisation des transactions. En effet, cette idée de coordination de mise à jour de données peut se révéler utile pour d'autres sources de données telles que des queues de messages, des services web, des systèmes de fichiers ou même des représentations en mémoire de données. Toutes ces sources de données peuvent participer dans une transaction grâce à une couche logicielle spécifique à chaque source en général nommée avec l'acronyme *RM* (pour *Resource Manager*) que nous reprendrons par la suite.

Dans les applications modernes la gestion des transactions est déléguée à une couche logicielle nommée *moteur transactionnel* (*transaction manager* en anglais ou *TM*). Vis-à-vis de son moteur transactionnel, une application qui a recours à des transactions a les responsabilités suivantes :

- Lui demander de créer une transaction.
- Lui communiquer qu'elles sont les sources de données impliquées dans la transaction. Durant cette étape, le RM d'une source l'enregistre auprès du moteur. On parle alors d'enrôlement d'une source de données au sein d'une transaction (*enlistement* en anglais).
- Effectuer les mises à jour des données de chaque source durant la transaction.
- Informer le moteur de la terminaison des mises à jour.

Le moteur transactionnel est responsable de la coordination des travaux des RM. Cette coordination a pour but de garantir l'intégrité des données : soit elles sont toutes été mises à jour correctement, soit le déroulement de la transaction a rencontré un problème et aucune modification de données n'a eu lieu. Pour gérer une telle coordination, du point de vue d'un moteur transactionnel les RM supportent tous une même API qui permet de les faire participer au sein d'une transaction. Toute cette architecture est exposée par la figure suivante :

## *Introduction aux transactions distribuées et aux algorithmes 2PC*

Une transaction distribuée consiste à mettre à jour des données de plusieurs RM en respectant les propriétés ACID. À cette fin, les moteurs transactionnels utilisent en général des algorithmes de type validation en 2 phases (*2 Phase Commit* en anglais, ou plus simplement *2PC* que nous utiliserons par la suite) :

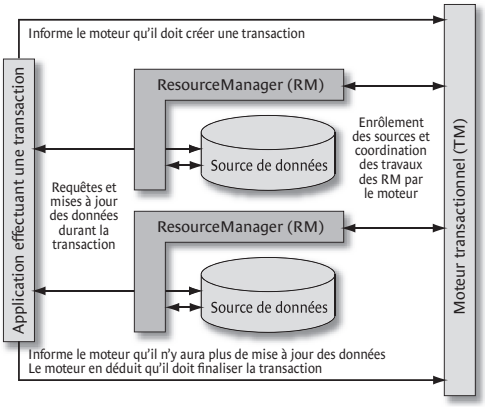


*Figure 20-1 : Interaction entre application, moteur transactionnel, RM et sources de données*

- Dans la première phase l'application enrôle chaque source de données et effectue les requêtes et mises à jour de données. Le moteur informe chacun des RM qu'il participe à une transaction distribuée. Chaque RM effectue alors une transaction locale sur ses données. Il contacte le moteur transactionnel juste avant de valider sa transaction locale si celle-ci a réussi ou dés qu'un échec est constaté. On parle de vote de la part des RM. Chaque RM vote *Succès* ou *Echec*.
- Si au moins un RM a échoué dans sa transaction locale, le moteur informe tous les RM participants que la transaction distribuée a échoué. Chaque RM doit alors annuler sa transaction locale en remettant ses données dans leur état initial. Si le succès a été voté à l'unanimité, la transaction distribuée est en passe d'être réussie et le moteur informe chaque RM qu'il doit valider sa transaction locale.

Une variante de cet algorithme existe. Dans la première phase, lorsqu'un RM constate que sa transaction locale a réussi, il peut la terminer effectivement. Si la transaction distribuée réussit aussi, le RM n'a alors rien à faire durant la seconde phase. Dans le cas contraire le RM doit refaire une transaction locale pour annuler les effets de la première transaction. On parle d'algorithme de *recouvrement* (*recovery* en anglais) de mise à jour des données.

Tel quel, l'algorithme 2PC n'est pas fiable à 100%. Une panne réseau entre le moteur et un RM qui survient au milieu de la deuxième phase laisse ce RM dans un état inconsistant. Une telle situation est qualifiée de douteuse (*in-doubt* en anglais). Les conséquences négatives de ces cas rares mais inévitables sont réparées grâce à un *service de recouvrement* qui vérifie périodiquement si une transaction distribuée a potentiellement corrompu les données en se terminant d'une manière douteuse. Ce service peut tenter de réparer automatiquement les données corrompues grâce à des informations sauvées par la source de données lors de la première phase. S'il échoue, le service de recouvrement peut, en dernier recours, informer un administrateur.

Il existe des protocoles permettant à un moteur de coordonner une transaction distribuée entre les RM. On peut citer les protocoles *OleDB*, *XA* ou *WS-AtomicTransaction*.

Au vu de la description de l'algorithme 2PC, il est clair que les transactions distribuées sont des opérations coûteuses qui nécessitent de nombreuses communications entre les différents participants. Il est donc toujours préférable d'utiliser une transaction locale si possible.

## *Transaction locale sur une connexion*

Dans l'explication de l'algorithme 2PC nous avons parlé de transaction locale. La plupart des SGBD présentent nativement la possibilité d'effectuer des transactions locales. La classe `DbConnection` (dont dérivent toutes les classes de connexion des fournisseurs de données ADO.NET) expose la méthode `BeginTransaction()` qui retourne un objet dont la classe dérive de `DbTransaction`. La plupart des fournisseurs de données encapsulent le support transactionnel du SGBD sous-jacent en réécrivant la méthode `BeginTransaction()` et en spécialisant la classe `DbTransaction`. L'exemple suivant exploite la base de données relationnelle décrite en page 710. Il montre comment effectuer une transaction locale avec le fournisseur `SqlClient`. Nous y effectuons deux commandes sur notre base de données :

- On ajoute le département *Communication* à la base de données `ORGANISATION`.
- On ajoute un employé à ce département.

Ces commandes doivent être exécutées au sein d'une transaction. En effet, si l'ajout du département échoue et si l'ajout de l'employé réussit, la base de données se retrouve dans un état incohérent car un employé appartient à un département qui n'existe pas (notez que dans ce cas précis la condition d'intégrité référentielle `FOREIGN KEY (DepID) REFERENCES DEPARTEMENTS(DepID)` empêche ce type de corruption de données).

*Exemple 20-1 :*

```
using System.Data.Common ;
using System.Data.SqlClient ;
class Program {
   static void Main() {
      string sCnx =
         "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION" ;
      string sCmd1 =
         "INSERT INTO DEPARTEMENTS VALUES ('COM','Communication')" ;
      string sCmd2 =
      "INSERT INTO EMPLOYES VALUES ('COM','Smith','Adam','0497112239')" ;
      try {
         using (SqlConnection cnx = new SqlConnection(sCnx)) {
            cnx.Open() ;
            DbTransaction tx = cnx.BeginTransaction();
            DbCommand cmd = new SqlCommand(sCmd1, cnx) ;
            cmd.Transaction = tx ;
            cmd.ExecuteNonQuery() ;
            cmd.CommandText = sCmd2 ;
            cmd.ExecuteNonQuery() ;
            tx.Commit();
            // Ici les deux commandes se sont correctement exécutées
            // et la transaction a été effectuée.
         } // end using cnx.
      } catch {
         // Si une exception a été lancée avant ou pendant l'appel à
         // tx.Commit() les données de la base n'ont pas été modifiées.
      }
```

```
    }
}
```

Dans le cas particulier du SGBD *SQL Server*, le langage T-SQL offre la possibilité d'effectuer des transactions (d'où le T pour *Transact*). Par exemple le code T-SQL suivant permet de réaliser nos deux opérations d'une manière transactionnelle. Notez la nécessité de tester après chaque opération si une erreur s'est produite :

```
BEGIN TRANSACTION
INSERT INTO DEPARTEMENTS VALUES ('COM','Communication')
If @@error <> 0 GOTO ERR_HANDLER
INSERT INTO EMPLOYES VALUES ('COM','Smith','Adam','0497112239')
If @@error <> 0 GOTO ERR_HANDLER
COMMIT TRANSACTION
RETURN

ERR_HANDLER:
   ROLLBACK TRANSACTION
   RETURN
```

Dans notre programme C# le fournisseur de données `SqlClient` exploite T-SQL pour reproduire ce comportement transactionnel. L'utilisation directe de T-SQL est donc nécessairement plus performante pour réaliser des transactions locales. En revanche, l'utilisation du fournisseur de données se prête mieux à la rédaction de transactions complexes impliquant un grand nombre de lignes. Précisons qu'il vaut mieux ne pas mettre à jour une trop grosse quantité de données lors d'une même transaction locale. Un ordre de grandeur à ne pas dépasser est le millier de lignes. En effet, dans le cas d'une transaction massive qui se termine en échec ou en situation douteuse, le temps pris pour finaliser la transaction devient prohibitif. Si vos besoins dépassent cette limite, il faut tronçonner vos mises à jour en transactions de taille raisonnable.

## *Le moteur transactionnel de Windows : le DTC*

*Microsoft* livre un moteur transactionnel avec ses OS. Ce moteur se nomme le *DTC* (*Distributed Transaction Coordinator* parfois nommé *MSDTC*). Il supporte à l'heure actuelle les protocoles *OleDB* et *XA*. La version *Windows Vista* de DTC supportera aussi les protocoles transactionnels relatifs aux *Web Services, WS-Coordination*, *WS-AtomicTransaction* et *WS-BusinessActivity*. *Windows Vista* fournira aussi des RM permettant de faire des transactions sur des sources de données *Windows* telles que des fichiers ou la base de registre.

Le DTC n'est pas forcément accessible ou installé selon votre version de *Windows*. Vous pouvez vérifier sa présence en effectuant la manipulation suivante : *Démarrer* ▶ *Panneau de configuration* ▶ *Outils d'administration* ▶ *Services de composants* ▶ *Services de composants* ▶ *Ordinateur* ▶ *Poste de travail* ▶ *Distributed Transaction Coordinator*.

Sous *Windows XP SP2* le DTC est installé par défaut mais toutes ses options sont désactivées par défaut. Vous pouvez avoir accès à la fenêtre de paramétrage du DTC comme ceci : (...) *Services de composants* ▶ *Ordinateur* ▶ *Poste de travail* ▶ Click droit *Propriétés* ▶ Onglet *MSDTC.* Activez alors les paramètres de sécurité dont vous avez besoin.

Sous *Windows 2003* le DTC n'est pas installé par défaut. Pour l'installer, il vous suffit d'effectuer cette manipulation : *Démarrer* ▶ *Panneau de configuration* ▶ *Ajout/Suppression de programmes*

*ou Ajouter/Supprimer des composants Windows* ► *Serveur d'application.* Installez alors COM+ puis MSDTC.

Si le DTC est en cours d'exécution, il est possible de lister les transactions courantes avec l'onglet *'Liste des transactions'* et d'obtenir des statistiques sur les transactions effectuées par le DTC avec l'onglet *'Statistiques de transactions'*.

Depuis plusieurs années le moteur DTC est exploitable à partir de la technologie COM+. Depuis les versions 1.x de .NET les développeurs peuvent réaliser des transactions distribuées avec le DTC. En effet la partie du *framework* .NET nommée *Service d'Entreprise* (qui fait l'objet de la section page 295) permet d'encapsuler les services de la technologie COM+ au moyen d'une API .NET. Ainsi, avec les versions 1.x du *framework* il faut utiliser les services d'entreprises pour réaliser des transactions distribuées et les fournisseurs de données ADO.NET pour réaliser des transactions locales.

# *Le framework System.Transactions*

La version 2 du *framework* .NET présente le nouvel espace de noms `System.Transactions` qui propose un modèle de programmation transactionnelle unifié. Tout assemblage qui souhaite utiliser les services de ce *framework* doit référencer l'assemblage `Systems.Transactions.dll`.

## *LTM, RM durables et RM volatiles*

Le *framework* `System.Transactions` contient un moteur transactionnel nommé *LTM* (*Lightweight Transaction Manager*) spécialisé dans la gestion de *RM volatiles*. La caractéristique d'un RM volatile est qu'il n'a pas besoin de service de recouvrement lorsqu'il participe à une transaction 2PC qui donne lieu à une situation douteuse. En conséquence, les données gérées par un RM volatile ne sont souvent stockées qu'en mémoire, là où une situation douteuse ne risque pas de les corrompre durablement. Tôt ou tard le processus contenant les données potentiellement corrompues tombera.

Le concept de RM volatile s'oppose à celui de *RM durable*. Un RM durable est donc un RM qui a besoin du service de recouvrement lorsqu'il participe à une transaction 2PC qui donne lieu à une situation douteuse. Les données gérées par un RM durable sont sauvées sur un disque dur, d'une manière durable d'où le nom. Avant la fin de la première phase, un RM durable qui participe à une transaction 2PC doit stocker l'ancienne et la nouvelle version des données. Si la transaction donne lieu à une situation douteuse, le service de recouvrement contactera le RM durable pour lui indiquer quelle version des données il doit garder. Le laps de temps entre la fin de la transaction dans un cas douteux et le contact du service de recouvrement peut se chiffrer en minutes voire en heures. Plus de détails à ce sujet sont disponibles dans l'article **Performing Recovery** des **MSDN**.

Toute transaction créée avec le *framework* `System.Transactions` est d'abord gérée par le LTM. L'utilisation du LTM est bien plus performante que celle du DTC puisque toutes les communications entre le LTM et ses RM sont réalisées à l'intérieur du même domaine d'application.

Une transaction gérée par le LTM peut impliquer plusieurs RM volatiles et au plus un RM durable qui sait accomplir sa tâche en une phase ou en deux phases selon l'algorithme utilisé par le moteur. On qualifie un tel RM durable de *Promotable Single Phase Enlistment* (*PSPE*). Le *framework* `System.Transactions` sait déléguer automatiquement une transaction du LTM vers le DTC dés qu'une des conditions suivantes survient :

- Un RM durable non PSPE enrôle sa source de données dans la transaction.
- Un RM durable PSPE participe déjà à la transaction lorsqu'un second RM durable (PSPE ou non) enrôle sa source de données. À ce moment précis, le premier RM durable PSPE doit sauver les données pour résoudre un éventuel cas douteux. Il passe du mode une phase au mode deux phases. On dit qu'il a été promu. La documentation anglo saxonne parle de *promotable enlistment*.
- La transaction est sérialisée pour être utilisée dans un autre domaine d'application.

Toutes les connexions à un SGBD sont des RM durables. Cependant, seules les connexions vers le SGBD *SQL Server 2005* sont des RM durables PSPE (pour l'instant).

La documentation anglo saxonne dit qu'une transaction est *escalated* lorsqu'elle est déléguée du LTM vers le DTC (on pourrait traduire le terme *escalated* par promue ou alourdie en français). Cette *escalation* d'une transaction se fait d'une manière complètement transparente du point de vue du code client de `System.Transactions`. Ce sont les mêmes classes et méthodes de `System.Transactions` qui sont utilisées indépendamment du fait qu'à l'exécution le LTM ou le DTC est impliqué. L'optimisation est implicite mais il est préférable que vous ayez conscience de tout ceci si les performances constituent pour vous un enjeu important. Précisons qu'en interne, la délégation d'une transaction vers le DTC n'élimine pas complètement le LTM de la partie. Ce dernier collabore en fait avec le DTC pour continuer à gérer les RM volatiles.

Nous mentionnons un malentendu classique. Comprenez bien que le LTM et le DTC sont tous deux des moteurs transactionnels distribués. Leur différence ne se situe pas au niveau du nombre de RM qu'ils peuvent gérer. Seule la qualité durable/durable PSPE/volatile des RM participant à une transaction permet au *framework* `System.Transactions` de décider d'impliquer ou non le DTC.

## *Transaction implicite avec System.Transactions*

Réécrivons l'Exemple 20-1 en utilisant les types de l'espace de noms `System.Transactions`:

*Exemple 20-2 :*

```
using System.Data.Common ;
using System.Data.SqlClient ;
using System.Transactions ;
class Program {
   static void Main() {
      string sCnx =
         "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION" ;
      string sCmd1 =
         "INSERT INTO DEPARTEMENTS VALUES ('COM','Communication')" ;
      string sCmd2 =
      "INSERT INTO EMPLOYES VALUES ('COM','Smith','Adam','0497112239')" ;
      try {
         using (TransactionScope txScope = new TransactionScope()) {
            using (SqlConnection cnx = new SqlConnection(sCnx)) {
               cnx.Open();
               DbCommand cmd = new SqlCommand(sCmd1, cnx) ;
               cmd.ExecuteNonQuery() ;
```

```
                cmd.CommandText = sCmd2 ;
                cmd.ExecuteNonQuery() ;
                txScope.Complete();
             } // end using cnx.
          } // end using txScope, la transaction s'effectue ici.
       } catch {
          // Si une exception a été lancée avant ou pendant l'appel à
          // txScope.Complete() les données de la base n'ont pas été
          // modifiées.
       }
    }
}
```

Cet exemple illustre le fait que toute connexion ouverte pendant l'existence d'un objet de type `System.Transactions.TransactionScope` est automatiquement et implicitement enrôlée dans la transaction sous-jacente. Si vous exécutez l'exemple précédent sur un autre SGBD que *SQL Server 2005* (par exemple avec *SQL Server* 7 ou 2000) vous constaterez qu'une transaction DTC est créée dans l'onglet *'Liste des transactions'* du DTC. Dans ce cas précis de transaction locale à un SGBD de type autre que *SQL Server 2005* il est plus efficace d'utiliser le modèle de programmation transactionnel ADO.NET 1.x proposé par le fournisseur de données. *Jim Johnson* présente dans une entrée de son blog disponible à l'URL `http://pluralsight.com/blogs/jimjohn/archive/2005/09/13/14795.aspx` une possibilité de contourner cette limitation en rendant PSPE un RM durable non PSPE.

Le programme suivant illustre une transaction distribuée qui porte sur deux bases de données. L'idée est ici de muter un employé de la base `ORGANISATION` vers la base `ORGANISATIOSN2`. Grâce au modèle de programmation unifié de `System.Transactions`, le code de cet exemple est similaire à celui de l'exemple précédent:

*Exemple 20-3 :*

```
using System.Data.Common ;
using System.Data.SqlClient ;
using System.Transactions ;
class Program {
   static void Main() {
      string sCnx1 =
        "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION" ;
      string sCnx2 =
        "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION2" ;
      string sCmd1 =
        "DELETE FROM EMPLOYES WHERE Nom='Smith' AND Prénom='Adam'" ;
      string sCmd2 =
      "INSERT INTO EMPLOYES VALUES ('DEV','Smith','Adam','0497112239')" ;
      try {
         using ( TransactionScope txScope = new TransactionScope() ) {
            using ( SqlConnection cnx1 = new SqlConnection(sCnx1) ) {
               cnx1.Open();
               DbCommand cmd = new SqlCommand(sCmd1, cnx1) ;
               cmd.ExecuteNonQuery() ;
            } // end using cnx1.
```

```
            using ( SqlConnection cnx2 = new SqlConnection(sCnx2) ) {
               cnx2.Open();
               DbCommand cmd = new SqlCommand(sCmd2, cnx2) ;
               cmd.ExecuteNonQuery() ;
            } // end using cnx2.
            txScope.Complete() ;
         } // end using txScope, la transaction s'effectue ici.
      } catch { }
   }
}
```

Dans ce cas le DTC est exploité quelquesoit les types des deux SGBD sous jacents.

## *Evènements déclenchés lors d'une transaction*

Vous avec la possibilité d'être averti quand une transaction est terminée au moyen de l'évènement `Transaction.TransactionCompleted`. Vous avec aussi la possibilité d'être averti quand une transaction se met à être gérée par le DTC au moyen de l'évènement `TransactionManager.DistributedTransactionStarted`. Réécrivons l'Exemple 20-3 en nous abonnant à ces évènements :

*Exemple 20-4 :*

```
...
class Program {
   static void Main() {
      ...
      try {
         using ( TransactionScope txScope = new TransactionScope() ) {
           Transaction.Current.TransactionCompleted += OnTxCompleted;
           TransactionManager.DistributedTransactionStarted +=
                                              OnDistributedTxStarted;
           using ( SqlConnection cnx1 = new SqlConnection(sCnx1) ) {
      ...
   }
   static void OnTxCompleted(object sender, TransactionEventArgs e) {
      Transaction tx = e.Transaction;
      System.Console.WriteLine("Completed! Status:" +
                          tx.TransactionInformation.Status.ToString());
   }
   static void OnDistributedTxStarted(object sender,
                                      TransactionEventArgs e) {
      Transaction tx = e.Transaction;
      System.Console.WriteLine("Distributed tx started!");
   }
}
```

## *Détails internes du fonctionnement de System.Transactions*

La magie de l'enrôlement implicite et automatique d'une connexion dans une transaction se fait en interne grâce à la propriété statique `ITransaction Transaction.Current{get;set;}`.

L'interface ITransaction présente des méthodes permettant à un RM d'enrôler sa source de données dans la transaction sous-jacente. Le code de l'activation d'un RM (par exemple l'ouverture d'une connexion) n'a alors qu'à s'enrôler si la propriété Transaction.Current retourne une transaction. Bien entendu, le fait de créer une instance de TransactionScope entraîne la création en interne de la transaction qui est retournée par la propriété Transaction.Current.

Rappelons qu'une connexion n'est marquée comme disponible au niveau de son pool de connexions que lorsqu'elle est fermée (avec la méthode Close() ou Dispose()). Cette règle est modifiée lorsqu'une connexion est enrôlée dans une transaction. Elle n'est alors marquée comme disponible au niveau de son pool que lorsque la transaction est achevée.

## *Paramétrer le niveau d'isolation entre transactions (TIL)*

La classe TransactionScope présente de nombreux constructeurs. Certains prennent en paramètre une instance de la structure TransactionOptions. Cette instance permet de paramétrer le *time out* de la transaction (qui est d'une minute par défaut) ainsi que son niveau d'isolation par rapport aux autres transactions qui s'effectuent sur les mêmes RM.

Ce niveau d'isolation est plus connu sous l'acronyme *TIL* pour *Transaction Isolation Level*. Naturellement plus le TIL est haut, plus les performances sont dégradées du fait d'un verrouillage plus strict des données. Selon le niveau d'isolation choisi, trois sortes de problèmes peuvent ou non survenir :

- *Lecture sale* (*Dirty read* en anglais) : Une transaction T1 en cours peut avoir accès en lecture aux données modifiées par une autre transaction T2 en cours. Cela pose un problème si T2 ne valide pas ses changements.
- *Lecture non répétable* (*Non repeatable read* en anglais) : Lorsqu'une transaction en cours effectue une seconde lecture d'une certaine donnée, il se peut que la valeur obtenue ait changé du fait d'une transaction qui s'est terminée durant le laps de temps entre les deux lectures.
- *Lecture fantôme* (*Phantom read* en anglais) : Lorsqu'une transaction en cours effectue une seconde fois une requête qu'elle a déjà effectuée, il se peut que l'ensemble des lignes retournées la seconde fois soit différent de l'ensemble des lignes retournées la première fois. Cela est alors du à des modifications effectuées et validées entre temps par une autre transaction.

Voici les niveaux d'isolations proposés par l'énumération IsolationLevel : du plus permissif au plus strict et donc, du moins coûteux en terme de performance au plus pénalisant :

| TIL | Lecture sale | Lecture non répétable | Lecture fantôme |
|---|---|---|---|
| ReadUncommited | possible | possible | Possible |
| ReadCommited | impossible | possible | Possible |
| RepeatableRead | impossible | impossible | Possible |
| Snapshot | impossible | impossible | impossible |
| Serialize | impossible | impossible | impossible |

La différence entre les TIL `Snapshot` et `Serialize` tient dans le fait que lors de l'utilisation du TIL `Snapshot`, une exception peut être levée (et par conséquent une transaction peut échouée) si un des trois cas problématiques est constaté. Si vous choisissez le TIL `Serialize`, sachez qu'en interne un système de verrouillage des données est mis en place. Ce système est coûteux car il gène l'exécution des autres transactions puisqu'elles doivent attendre leur tour pour avoir accès aux données verrouillées. On qualifie l'approche `Snaphot` d'*optimiste* (dans le sens où l'on est confiant que l'échec des transactions survient rarement) tandis que l'approche `Serialize` est qualifiée de *pessimiste*.

À vous de choisir le niveau d'isolation de vos transactions en soupesant l'importance de leur succès face à votre besoin de performance. Une boutade dit que seules les opérations manipulant de l'argent ont besoin du niveau d'isolation `Serialize`.

## *Paramétrer le comportement à adopter lors de transactions imbriquées*

Certains constructeurs de la classe `TransactionScope` acceptent une valeur de l'énumération `TransactionScopeOption` qui permet de déterminer comment se comporte le nouveau scope s'il est imbriqué dans un autre scope transactionnel. Les valeurs de cette énumération sont :

- `Mandatory` : Une transaction doit exister au moment où le nouveau scope est créé. Dans le cas contraire une exception est levée.
- `NotSupported` : Ce nouveau scope n'entraîne pas de création d'une nouvelle transaction et ne modifie pas une éventuelle transaction courante au moment ou il est créé.
- `Required` : Ce nouveau scope à besoin d'une transaction. Il utilise la transaction courante si elle existe, sinon il en crée une nouvelle. La valeur `Required` est la valeur prise par défaut.
- `RequiresNew` : Ce nouveau scope entraîne la création d'une nouvelle transaction indépendamment du fait qu'une transaction courante existe ou non. Le cas échéant, la transaction cachée redeviendra la transaction courante à la fermeture de ce nouveau scope.
- `Supported` : Ce nouveau scope n'entraîne pas de création d'une nouvelle transaction mais prend en compte une éventuelle transaction courante au moment ou il est créé.

## *Transaction explicite avec System.Transactions*

L'espace de noms `System.Transactions` permet d'exploiter des transactions en se passant d'une instance de la classe `TransactionScope`. Voici l'Exemple 20-2 réécrit avec cette technique :

*Exemple 20-5 :*

```
using System.Data.Common ;
using System.Data.SqlClient ;
using System.Transactions ;
class Program {
   static void Main() {
      string sCnx =
         "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION" ;
      string sCmd1 =
         "INSERT INTO DEPARTEMENTS VALUES ('COM','Communication')" ;
```

```
      string sCmd2 =
      "INSERT INTO EMPLOYES VALUES ('COM','Smith','Adam','0497112239')" ;
      try {
         CommittableTransaction tx = new CommittableTransaction();
         using (SqlConnection cnx = new SqlConnection(sCnx)) {
            cnx.Open() ;
            // Ici la connexion est enrôlée.
            cnx.EnlistTransaction(tx);
            DbCommand cmd = new SqlCommand(sCmd1, cnx) ;
            cmd.ExecuteNonQuery() ;
            cmd.CommandText = sCmd2 ;
            cmd.ExecuteNonQuery() ;
         } // end using cnx.
         tx.Commit();
      } catch { /*...*/ }
   }
}
```

Nous allons voir dans la prochaine section qu'il est parfois nécessaire d'avoir recours à une cette syntaxe explicite dans certains contextes asynchrones. Mis à part ce cas, cette pratique est en général à éviter puisque nous voyons qu'elle nécessite plus de code (puisqu'il faut explicitement enrôler les sources de données) sans apporter d'avantages.

# *Utilisation avancées de System.Transactions*

## *Traitement d'une transaction sur plusieurs threads*

Dans une application multithreads la transaction courante définie par la propriété statique `Transaction.Current` n'est valide que pour le thread qui l'a créée. Cela implique que pendant l'exécution d'une transaction par un thread `A`, un autre thread `B` obtiendrait une valeur nulle à partir de cette propriété. Si vous désirez faire participer plusieurs threads dans la même transaction, il est nécessaire de fournir aux autres threads des *transactions dépendantes* de la transaction originale. Une transaction dépendante est obtenue avec la méthode `IDependantTransaction ITransaction.DependentClone(bool)`. L'interface `IDependantTransaction` qui étend l'interface `ITransaction` n'a que la méthode `Complete()` en plus. Cette méthode sert à indiquer à la transaction mère qu'une transaction dépendante a correctement terminé son travail. Si toutes les transactions dépendantes appellent `Complete()` et si la transaction mère a aussi réussi, vous pouvez alors appeler la méthode `Commit()` sur cette dernière.

Pour vous évitez d'avoir à gérer la synchronisation entre les appels à `Complete()` et l'appel à `Commit()` vous pouvez créer les transactions dépendantes en passant la valeur `DependentCloneOption.BlockCommitUntilComplete` à la méthode `DependentClone(DependentCloneOption)`. En procédant comme ceci, l'appel à `Commit()` est bloquant jusqu'à ce que toutes les transactions dépendantes aient votées. Si vous passez la valeur `DependentCloneOption.RollbackIfNotComplete` lors de la création d'une transaction dépendante, il faut impérativement que le thread responsable de cette transaction dépendante appelle la méthode `Complete()` avant que la méthode `Commit()` soit appelée sur la transaction mère. Dans le cas contraire toute la transaction échoue. Naturellement, il suffit qu'une seule transaction (mère ou dépendante) échoue pour que la

transaction mère et toutes ses transactions dépendantes échouent. Voici un exemple exposant la syntaxe d'utilisation des transactions dépendantes :

*Exemple 20-6 :*

```
using System.Data.Common ;
using System.Data.SqlClient ;
using System.Transactions ;
using System.Threading ;
class Program {
   static string sCnx =
      "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION" ;
   static string sCmd1 =
      "INSERT INTO DEPARTEMENTS VALUES ('COM','Communication')" ;
   static string sCmd2 =
      "INSERT INTO EMPLOYES VALUES ('COM','Smith','Adam','0497112239')" ;
   static void Main() {
      try {
         using (TransactionScope txScope = new TransactionScope()) {
            using (SqlConnection cnx = new SqlConnection(sCnx)) {
               cnx.Open() ;
               DbCommand cmd = new SqlCommand(sCmd1, cnx) ;
               cmd.ExecuteNonQuery() ;
               DependentTransaction depTx =
                  Transaction.Current.DependentClone (
                     DependentCloneOption.BlockCommitUntilComplete );
               ThreadPool.QueueUserWorkItem(AsyncProc, depTx);
               txScope.Complete();
            } // end using cnx.
         } // end using txScope, l'appel à ITransaction.Commit()
           // s'effectue ici.
      } catch { }
   }
   static void AsyncProc(object state) {
     DependentTransaction depTx = state as DependentTransaction;
      try {
         using (SqlConnection cnx = new SqlConnection(sCnx)) {
            cnx.Open() ;
            cnx.EnlistTransaction(depTx) ;
            DbCommand cmd = new SqlCommand(sCmd2, cnx) ;
            cmd.ExecuteNonQuery() ;
            depTx.Complete();
         } // end using cnx.
      } catch {
         depTx.Rollback() ;
      }
   }
}
```

## Gestion asynchrone de la complétion d'une transaction

Vous pouvez gérer d'une manière asynchrone la complétion d'une transaction. Cette possibilité est utile car les blocages des threads durant l'attente de la complétion d'une transaction a un impact négatif sur les performances. Pour exploiter cette possibilité, il faut gérer explicitement vos transactions comme dans l'Exemple 20-5 et appeler la méthode `BeginCommit()` au lieu de `Commit()`. Cette méthode accepte en paramètre un délégué représentant la méthode à appeler lorsque la transaction est achevée. Au sein de cette méthode, il ne vous reste plus qu'à appeler la méthode `EndCommit()`. Cette dernière méthode lèvera une exception si la transaction n'a pas réussie. Vous avez alors une chance de fournir le code à exécuter en cas de non succès lors du rattrapage de l'exception.

*Exemple 20-7 :*

```
class Program {
   static void Main() {
      ...
      CommittableTransaction tx = new CommittableTransaction() ;
      try {
         using (SqlConnection cnx = new SqlConnection(sCnx)) {
            cnx.Open() ;

            cnx.EnlistTransaction(tx) ;
            ...
            tx.BeginCommit(OnCommited,null);
         } // end using cnx.
      } catch { /*...*/ }
      // Ici, le code peut s'exécuter alors que la transaction tx
      // n'est pas encore commitée.
   }
   static void OnCommited(System.IAsyncResult asyncResult) {
      CommittableTransaction tx = asyncResult as CommittableTransaction;
      try {
         using (tx) {
            tx.EndCommit(asyncResult) ;
         }
      } catch (TransactionException e) {
         // Ici, mettre le code a déclencher quand
         // la transaction a échoué !
      }
   }
}
```

## System.Transactions et CAS

Il est intéressant de remarquer que le code qui implique le recours au DTC pour gérer une transaction doit avoir la permission CAS `System.Transactions.DistributedTransactionPermission`. Dans le cas de la promotion d'une RM lors de la délégation d'une transaction du LTM vers le DTC, c'est le code qui a déclenché la promotion qui doit avoir la permission et non le code responsable de l'enrôlement.

# *Introduction à la création d'un RM transactionnel*

Nous allons présenter ici la conception d'un RM transactionnel au moyen d'une classe générique nommée `TxList<T>`. Cette classe constitue l'implémentation d'un type de liste où l'ajout d'un élément au moyen de la méthode `TxAdd(T)` est un changement qui peut faire partie d'une transaction. Vous pouvez ajouter plusieurs éléments durant une transaction. Ils seront tous ajoutés si la transaction est validée ou aucun d'eux ne sera ajouté si la transaction est avortée. Pour rendre possible ce comportement, une instance de `TxList<T>` maintient en interne la liste `dataToCommit` qui représente les éléments en attente d'être ajoutés. Les deux méthodes privées `OnCommit()` et `OnRollback()` contiennent les implémentations des opérations *commit* et *rollback*.

Pour que ses instances soient manipulables par `System.Transactions`, il faut que la classe `TxList<T>` implémente l'interface `System.Transactions.ISinglePhaseNotification`. Cette interface présente la méthode `SinglePhaseCommit()` qui est appelée par le moteur transactionnel lorsque la liste concernée est l'unique RM d'une transaction qui est sur le point d'être validée. Cette interface implémente l'interface `System.Transactions.IEnlistmentNotification`. `IEnlistmentNotification` présente les méthodes `Prepare()`, `Commit()`, `InDoubt()` et `Rollback()` qui sont appelées par le gestionnaire de transactions lorsque la liste concernée est enrôlée dans une transaction distribuée sur plusieurs RM. Dans ce cas, le moteur transactionnel sous-jacent (LTM ou DTC) gère automatiquement la transaction en deux phases (algorithme 2PC).

Lorsqu'un élément est ajouté à une instance de `TxList<T>`, nous testons si elle est enrôlée dans une transaction. Si ce n'est pas le cas, nous appelons la méthode `Enlist()` qui enrôle la liste dans la transaction courante si nous sommes dans un contexte transactionnel. Pour cela, le code de cette méthode appèle la méthode `Transaction.EnlistVolatile()` sur la transaction courante. Cela indique au moteur transactionnel qu'il a à faire à un RM volatile.

*Exemple 20-8 :*

```
using System ;
using System.Collections.Generic ;
using System.Transactions ;
public class TxList<T> : List<T>, ISinglePhaseNotification {
   private Transaction m_Tx;
   private List<T> dataToCommit;
   private string m_Name ;

   public TxList(string name) {
      m_Name = name ;
      dataToCommit = new List<T>() ;
   }
   public void TxAdd(T t) {
      Console.WriteLine(m_Name + ".TxAdd(" + t.ToString() +")") ;
      if ( m_Tx == null )
         Enlist();
      dataToCommit.Add(t);
   }
   private void OnCommit() {
      Console.WriteLine("   "+m_Name+".OnCommit()") ;
      foreach (T t in dataToCommit)
```

```
            base.Add(t);
         dataToCommit.Clear();
      }
      private void OnRollback() {
         dataToCommit.Clear();
      }
      private void Enlist() {
         m_Tx = Transaction.Current;
         if (m_Tx != null) {
            Console.WriteLine("   " + m_Name + ".EnlistVolatile()") ;
            m_Tx.EnlistVolatile(this, EnlistmentOptions.None);
         }
      }
      public void DisplayContent() {
         Console.Write("--> Contenu de " + m_Name + " : ") ;
         foreach (T t in this)
            Console.Write( t.ToString() + ";") ;
         Console.Write("   dataToCommit: " ) ;
         foreach (T t in dataToCommit)
            Console.Write(t.ToString() + ";") ;
         Console.WriteLine() ;
      }

   #region IEnlistmentNotification Members
      public void Prepare(PreparingEnlistment preparingEnlistment) {
         Console.WriteLine(m_Name + ".Prepare()") ;
         preparingEnlistment.Prepared();
      }
      public void Commit(Enlistment enlistment) {
         Console.WriteLine(m_Name + ".Commit()") ;
         OnCommit();
         enlistment.Done();
      }
      public void InDoubt(Enlistment enlistment) {
         Console.WriteLine(m_Name + ".InDoubt()") ;
         throw new NotImplementedException();
      }
      public void Rollback(Enlistment enlistment) {
         Console.WriteLine(m_Name + ".Rollback()") ;
         OnRollback();
         enlistment.Done();
      }
   #endregion

   #region ISinglePhaseNotification Members
      public void SinglePhaseCommit(
                              SinglePhaseEnlistment singlePhaseEnlistment) {
         Console.WriteLine(m_Name + ".SinglePhaseCommit()") ;
         OnCommit();
```

```
      singlePhaseEnlistment.Committed();
   }
#endregion
}
```

Voici un petit programme qui exploite une seule instance de `txList<string>` dans une transaction :

*Exemple 20-9 :*

```
using System.Transactions ;
class Program {
   static void Main() {
      TxList<string> txList = new TxList<string>("List") ;
      using ( TransactionScope txScope = new TransactionScope() ) {
         txList.TxAdd("A") ;  txList.TxAdd("B") ;
         txScope.Complete();
         txList.DisplayContent() ;
      }
      txList.DisplayContent() ;
   }
}
```

On peut facilement suivre le déroulement des opérations grâce à l'affichage du programme :

```
List.TxAdd(A)
   List.EnlistVolatile()
List.TxAdd(B)
--> Contenu de List :    dataToCommit: A;B;
List.SinglePhaseCommit()
   List.OnCommit()
--> Contenu de List : A;B ;   dataToCommit:
```

Si l'on ne valide pas la transaction en mettant en commentaire l'appel à `txScope.Complete()`, la transaction est automatiquement avortée et le programme affiche ceci :

```
List.TxAdd(A)
   List.EnlistVolatile()
List.TxAdd(B)
--> Contenu de List :    dataToCommit: A;B;
List.Rollback()
--> Contenu de List :    dataToCommit:
```

Réécrivons notre programme de façon à ce qu'il gère deux listes transactionnelles au sein d'une même transaction :

*Exemple 20-10 :*

```
using System.Transactions ;
class Program {
   static void Main() {
      TxList<string> txList1 = new TxList<string>("List1") ;
```

```
      TxList<string> txList2 = new TxList<string>("List2") ;
      using ( TransactionScope txScope = new TransactionScope() ) {
         txList1.TxAdd("A") ;  txList1.TxAdd("B") ;
         txList2.TxAdd("C") ;  txList2.TxAdd("D") ;
         txScope.Complete();
         txList1.DisplayContent() ; txList2.DisplayContent() ;
      }
      txList1.DisplayContent() ; txList2.DisplayContent() ;
   }
}
```

Le moteur transactionnel sous jacent (LTM en l'occurrence) passe automatiquement en 2PC avec une phase de préparation et une phase de validation. Ceci devient clair avec l'affichage du programme :

```
List1.TxAdd(A)
   List1.EnlistVolatile()
List1.TxAdd(B)
List2.TxAdd(C)
   List2.EnlistVolatile()
List2.TxAdd(D)
--> Contenu de List1 :    dataToCommit: A;B;
--> Contenu de List2 :    dataToCommit: C;D;
List1.Prepare()
List2.Prepare()
List1.Commit()
   List1.OnCommit()
List2.Commit()
   List2.OnCommit()
--> Contenu de List1 : A;B ;   dataToCommit:
--> Contenu de List2 : C;D ;   dataToCommit:
```

Vous pouvez aussi déclarer au moteur transactionnel que votre RM est durable. Pour cela, il suffit d'appeler la méthode `Transaction.EnlistDurable()` au lieu de `Transaction.EnlistVolatile()` au moment de l'enrôlement. La méthode `EnlistDurable()` prend en argument l'identifiant unique du RM que le service de recouvrement devra appeler si la transaction donne lieu à une situation douteuse.

Vous pouvez aussi développer un RM durable PSPE en implémentant l'interface `System.Transaction.IPromotableSinglePhaseNotification` plutôt que l'interface `ISinglePhaseNotification`. Plus d'informations à ce sujet et plus généralement sur le développement de RM sont disponibles dans les articles **Optimization using Single Phase Commit and Promotable Single Phase Notification** et **Implementing a Resource Manage**r des **MSDN**.

# 21

# *XML*

## *Introduction à XML*

### *Les problèmes résolus par XML*

*XML* est l'acronyme de *eXtensible Markup Langage*. XML est la réponse au besoin de standardisation du formatage des données échangées entre systèmes hétérogènes. Avant XML, les données étaient échangées dans des formats propriétaires. Les applications devaient connaître ce format propriétaire pour exploiter les données et cela posait les trois problèmes suivants.

- **Les documents XML sont écrits avec du texte :**
  Ces formats propriétaires s'appuyaient sur les formats des types primitifs des systèmes sous-jacents. Mais la représentation d'un double ou d'une chaîne de caractères peut être fort différente d'un système à l'autre. Par exemple, le formatage des types primitifs .NET est standardisé mais son utilisation reste limitée à .NET.
  En XML, un paquet d'information s'appelle un *document XML*. Un document XML est un fichier texte, lisible par n'importe quel éditeur de texte sur n'importe quelle plateforme. Ce problème est résolu car chaque donnée est codée sous forme d'une chaîne de caractères *a priori* non typée. On verra que l'on peut néanmoins typer les données avec des types spécifiques à XML, qui sont indépendants du système sous-jacent.
- **Les documents XML ont la capacité d'auto-décrire leur structure :**
  Un autre gros problème de ces formats propriétaires concerne la structure dans laquelle sont stockées les informations. Il est normal qu'une application qui reçoit un numéro de commande sache interpréter et utiliser ce numéro de commande dans sa logique métier. Cependant, il est regrettable que l'application doive aussi savoir où ce numéro est stocké dans le paquet d'information reçu. On dit du paquet d'information reçu par l'application qu'il n'est pas *auto-descriptif*.
  Ce problème est résolu grâce à un système de balises extensibles (d'où les mots *Extensible Markup*). Grâce aux schémas, qui décrivent la structure d'un document, le chemin d'accès

à une donnée spécifique à l'intérieur du document XML peut être connu par l'application en même temps que l'instance du document. On peut voir ces schémas comme des types de documents XML.

- **Les documents XML sont semi structurés :**
  Un dernier problème de ces formats propriétaires est que, bien souvent, ils manquent de flexibilité dans la structuration des données. Concrètement, si la plupart des commandes contiennent un numéro de commande, un identifiant client et une description, il est difficile d'envoyer une commande sans description. En général, on n'a pas d'autres choix que d'adopter une convention du type : *si la description vaut « N/A » alors cela signifie qu'il n'y a pas de description pour cette commande*.
  Ce problème est résolu car les documents XML sont semi structurés. Concrètement, chaque donnée est encadrée par des balises, et une donnée peut être constituée de plusieurs autres données. Si une donnée commande n'a pas de balise description, l'application considère que la commande n'a pas de description. Autrement dit, l'application obtient une information (i.e pas de description pour cette commande) à partir d'un manque d'information (i.e pas de balise description pour cette commande).

## *Unification du monde des documents et du monde des données*

Le constat suivant peut être fait : la plupart des données sont stockées dans des documents exploitables par des humains. En effet, la plupart des informations se trouvent dans des documents textuels (`.html`, `.doc`, `.txt` etc).

Un « casse-tête » classique pour les développeurs est d'automatiser l'extraction de ces informations de ces documents textuels, afin de pouvoir les traiter avec des logiciels. Quel développeur n'a jamais été tenté d'extraire un numéro de commande d'un document HTML en se basant sur le fait que c'est la seule partie du document qui est écrite en bleue avec la police Verdana de taille 14 ?

Parce que XML est flexible et extensible, les éditeurs de documents XML peuvent à la fois prévoir des balises pour la présentation des données et des balises pour les données elles-mêmes. Il devient alors aisé de produire le document et d'extraire les données. Cette vision devient de plus en plus une réalité. Par exemple, la plupart des logiciels *Microsoft* de la gamme *Office 2003* (*Word*, *Excel* etc) produisent des documents dans des formats XML.

## *Structure d'un document XML*

Le système de balises XML est similaire au système de balises HTML, mis à part que vous pouvez créer autant de balises que vous souhaitez, d'où la notion d'extensibilité. Nous exposons ci-dessous le contenu d'un fichier XML représentant un extrait de l'inventaire d'une bibliothèque fictive (ce fichier est extrait de la documentation du *framework*):

*Exemple 21-1 :* `books.xml`

```
<?xml version="1.0" encoding="utf-8" ?>
<bookstore>
  <book genre="autobiography" publicationdate="1981" ISBN="1-861003-11-0">
    <title>The Autobiography of Benjamin Franklin</title>
    <author>
      <first-name>Benjamin</first-name>
```

```
      <last-name>Franklin</last-name>
    </author>
    <price>8.99</price>
  </book>
  <book genre="novel" publicationdate="1967" ISBN="0-201-63361-2">
    <title>The Confidence Man</title>
    <author>
      <first-name>Herman</first-name>
      <last-name>Melville</last-name>
    </author>
    <price>11.99</price>
  </book>
  <book genre="philosophy" publicationdate="1991" ISBN="1-861001-57-6">
    <title>The Gorgias</title>
    <author>
      <name>Plato</name>
    </author>
    <price>9.99</price>
  </book>
</bookstore>
```

Plusieurs remarques s'imposent :

- Il y a autant de balises que de types de données. Il y a une balise pour la notion de bibliothèque `<bookstore>`, une balise pour la notion de livre `<book>` une balise pour la notion de titre de livre `<title>` etc. Un groupe « `<balise>données</balise>` » est appelé *élément XML*. On voit qu'un élément XML peut contenir soit une donnée soit (exclusif) d'autres éléments XML.
- Un élément qui contient d'autres éléments peut également contenir des données grâce à la notion d'*attribut XML* (attention, cela n'a rien à voir avec la notion d'attribut .NET). Par exemple, le genre d'un livre est un attribut nommé `genre` de l'élément `<book>`.
- Le fait qu'un élément XML puisse en contenir d'autres permet de présenter les données sous une forme hiérarchique. Une application qui veut obtenir les titres des livres parus en 1967 doit d'abord trouver les éléments `<book>` ayant un attribut `publicationdate` égal à 1967 puis énumérer les éléments `<title>` correspondant. Peu importe l'ordre dans lequel les livres sont stockés dans le document XML, l'application saura toujours retrouver l'information dont elle a besoin.
- On peut insérer des commentaires dans un fichier XML avec la syntaxe `<!-- commentaire -->`.
- L'entête d'un document XML peut spécifier le type d'encodage du texte avec l'attribut `encoding`. On voit que le fichier texte `books.xml` est codé au format *UTF-8*.
- Bien que cet exemple ne le montre pas, certains attributs particuliers peuvent définir des *espaces de noms XML* dans un élément XML. On exploite les espaces de nom XML pour éviter qu'il y ait collision de noms des éléments ou des attributs lorsque plusieurs langages XML sont utilisés simultanément. On associe en général une URI (tel que *http://www.smacchia.com*) à un espace de noms XML pour être certain de l'unicité du nom de l'espace de noms. Modifions notre exemple pour illustrer la syntaxe :

```
...
<bookstore xmlns:SMA="http://www.smacchia.com" >
  <SMA:book genre="autobiography" publicationdate="1981"
   ISBN="1-861003-11-0">
    ...
  </SMA:book>
...
```

# *Introduction à XSD, XPath, XSLT et XQuery*

## *Typage des données, schéma et XSD*

*A priori*, les données d'un document XML ne sont pas typées. L'application peut décider de garder le nombre d'employés dans une chaîne de caractères. Elle peut aussi décider de convertir cette chaîne en un nombre entier. Dans ce cas l'application ne sait pas forcément que ce nombre ne peut pas être négatif.

Pour imposer des contraintes, il faut pouvoir décrire le schéma des données, c'est-à-dire décrire le type de chaque élément (chaîne de caractères, entier etc) et les contraintes sur chaque élément (entier positifs, inférieur à un million etc). Il existe plusieurs techniques pour décrire le schéma d'un document XML, et la technique *XSD* (*XML Schema Definition*) est celle qui a été retenue pour .NET. L'idée est d'associer à un document XML un schéma XSD décrivant les éléments. Par exemple, on aurait pu associer le schéma XSD suivant à notre document `books.xml` :

*Exemple 21-2 :* *books.xsd*

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
           elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="bookstore">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="book">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="title" type="xs:string" />
              <xs:element name="author">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element minOccurs="0" name="name"
                                type="xs:string" />
                    <xs:element minOccurs="0" name="first-name"
                                type="xs:string" />
                    <xs:element minOccurs="0" name="last-name"
                                type="xs:string" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
```

```
            <xs:element name="price" type="xs:decimal" />
          </xs:sequence>
          <xs:attribute name="genre" type="xs:string" use="required"/>
          <xs:attribute name="publicationdate" type="xs:unsignedShort"
                        use="required" />
          <xs:attribute name="ISBN" type="xs:string" use="required" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

On voit que l'on peut typer chaque élément d'un document XML, mais ce n'est pas une obligation. Les types disponibles et la conversion entre ces types et les types .NET sont présentés dans l'article **Data Type Support between XML Schema (XSD) Types and .NET Framework Types** des **MSDN**.

## *XPath*

*XPath* est un langage permettant de rédiger des requêtes pour sélectionner des nœuds dans un document XML. Certains l'ont surnommé « SQL pour XML » car la finalité de XPath vis à vis des données représentées au format XML est similaire à la finalité de SQL vis à vis des données représentées dans une base de données relationnelle.

XPath permet de sélectionner des noeuds dans un document XML avec une syntaxe comparable à l'écriture du chemin d'un fichier. Voici des exemples d'expressions XPath (en gras) appliquées sur le document `books.xml` suivies par la liste des nœuds sélectionnés :

```
/bookstore/book/author/first-name
<first-name>Benjamin</first-name>
<first-name>Herman</first-name>

/bookstore/book/author/*
<first-name>Benjamin</first-name>
<last-name>Franklin</last-name>
<first-name>Herman</first-name>
<last-name>Melville</last-name>
<name>Plato</name>

/bookstore/book[@publicationdate>1980]/title
<title>The Autobiography of Benjamin Franklin</title>
<title>The Gorgias</title>
```

## *XSLT*

L'acronyme *XSLT* veut dire *eXtensible Stylesheet Language Transformation*. XSLT est un langage permettant d'exploiter les données contenues dans un document XML afin d'obtenir un autre document. Le document produit peut être, par exemple, au format HTML, dans un autre format XML ou simplement au format texte.

Un programme écrit en XSLT (i.e un *stylesheet*) est un document XML. XSLT est construit sur un système de *template*. À l'exécution, XSLT sélectionne les nœuds du document source qui correspondent aux nœuds du *stylesheet* puis exécute le corps du *template* pour chaque noeud sélectionné. La sélection se fait par l'intermédiaire d'une expression XPath.

Le *stylesheet* suivant appliqué au document `books.xml` illustre ceci. Notez la présence de l'espace de noms caractéristique `xsl`. Les expressions en gras sont rédigées avec XPath.

*Exemple 21-3 :* *books.xslt*

```
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match= "/bookstore" >
    <xsl:for-each select= "book[@publicationdate>1980]" >
Titre : <xsl:value-of select= "title" />
Publié il y a <xsl:value-of select= "2005 - (@publicationdate)" /> ans
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

Le document produit est celui-ci :

```
Titre : The Autobiography of Benjamin Franklin
Publié il y a 24 ans

Titre : The Gorgias
Publié il y a 14 ans
```

Pour une présentation complète des technologies XPath et XSLT nous vous conseillons de consulter l'ouvrage suivant :

**Comprendre XSLT**

O'Reilly 2002

Bernd Amann, Philippe Rigaud

ISBN : 2-84177-148-2

## *XQuery*

Le langage *XQuery* a la même finalité que XSLT : transformer un document XML en un autre document texte (qui n'est pas nécessairement un document XML). Remarquez que cette finalité peut être aussi appréhendée comme une façon de faire des requêtes sur un document XML, d'où le nom de *XQuery*.

XQuery est considéré comme plus convivial que XSLT notamment parce qu'il n'est pas lui-même un langage XML. La raison principale de cet engouement pour XQuery réside cependant dans le fait que le flot d'exécution des instructions est comparable à celui des langages impératifs bien connus tels que C# ou Java.

À l'instar d'XSLT, XQuery exploite le langage XPath pour la sélection de nœud. Voici un programme XQuery qui transforme notre document `books.xml` en un fichier texte qui énumère dans l'ordre de leurs dates de publication les titres des livres autobiographiques :

```
for $b in document("books.xml")/book where
$b/genre="autobiography" return $b/title" sortby(publicationdate)
```

Clairement cette logique de sélection est très proche de celle proposée par le langage SQL.

XQuery 1.0 ayant été normalisé quelques mois après la sortie du *framework* .NET 2.0, ce dernier ne supporte malheureusement pas cette technologie. On peut cependant signaler que *SQL Server 2005* supporte un sous-ensemble de XQuery pour interroger les données XML stockées dans une base de données.

# *Les approches pour parcourir et éditer un document XML*

Le parcours d'un document XML (i.e la consommation par un programme des informations contenues dans un document XML) peut se faire selon deux approches :

- L'approche type **curseur** : Le document XML est parcouru à l'aide d'un curseur. Le curseur peut être vu comme la position dans le document XML de l'information en cours de lecture. Cette position peut désigner un élément XML ou un attribut XML. Le programme est responsable de déplacer le curseur à partir de la position courante vers la position des prochaines informations à consommer (aller vers mon prochain élément « frère », aller vers mon premier élément enfant etc).
- L'approche type **arbre** : Lors d'une phase de préchargement du document XML un arbre d'objets est créé. Chaque objet dans l'arbre représente une entité du document XML. On peut alors exploiter les informations contenues dans le document en parcourant l'arbre des objets.

L'approche type curseur a plusieurs avantages sur l'approche type arbre. Elle ne nécessite pas une phase de préchargement et la mémoire nécessaire pour exploiter le document XML ne dépend pas de la taille du document. Il est clair que l'approche type curseur est plus efficace en termes de performance. Cependant, la programmation objet fait qu'il est parfois plus aisé d'exploiter des informations contenues dans un arbre d'objets. Les deux modèles ont donc chacun leurs raisons d'être implémenter par le *framework* .NET.

Les opérations de génération et de modification d'un document XML peuvent aussi se faire selon ces deux approches. Les avantages et les inconvénients sont les mêmes mis à part que la phase de préchargement du modèle arbre est remplacé par une phase de post génération lorsque l'on souhaite finaliser les changements.

Enfin, nous allons expliquer que les deux approches ne sont pas incompatibles. Par exemple, il est possible de parcourir un arbre chargé en mémoire avec un curseur. Notamment, nous verrons que l'on peut se servir de requêtes XPath sur l'arbre afin d'obtenir un curseur sur l'ensemble des nœuds sélectionné. Tout l'avantage de ces pratiques réside dans le fait que l'on peut écrire du code de manipulation de données indépendant du mode de stockage XML. Les données XML sont consommées par une approche type curseur que le document soit pré chargé en mémoire ou chargé au fur et à mesure à partir d'une source telle qu'un fichier.

Si le *framework* standard `System.Xml` que nous allons présenter ne vous satisfait pas sur certains points, vous pouvez envisager d'avoir recours au *framework* gratuit et open-source `Mvp.Xml`. Plus d'information sont disponibles à l'URL `http://sourceforge.net/projects/mvp-xml/`.

# Parcours et édition d'un document XML avec un curseur (XmlReader et XmlWriter)

## Lecture des donnée avec XmlReader

Les classes abstraites `System.Xml.XmlReader` et `System.Xml.XmlWriter` permettent respectivement de parcourir et de modifier des données stockées dans un format XML avec une approche type curseur *forward only*. Un curseur *forward only* est un curseur qui ne peut se déplacer qu'en avant.

Ces classes sont abstraites pour s'affranchir de la façon dont sont stockées les données en interne. Les classes concrètes qui dérivent de `XmlReader` et `XmlWriter` ont la responsabilité de gérer le flot des données XML en lecture et en écriture. La paire de classes concrètes `System.Xml.XmlTextReader` et `System.Xml.XmlTextWriter` permettent de gérer un flot textuel de données stockées au format XML respectivement en lecture et en écriture.

La classe concrète `System.Xml.XmlNodeReader` permet de lire des données XML qui sont stockées dans un arbre avec une approche type curseur. L'utilisation de cette classe cumule les inconvénients de l'approche type curseur (complexité dans la rédaction du code) et de l'approche type arbre (pénalité au chargement des données et totalité des données stockée en mémoire). En revanche cette classe vous permet d'écrire un algorithme de lecture de données XML indépendamment de la source de données.

Il est conseillé de ne pas instancier directement les classes qui dérivent de `XmlReader`. En effet, les nombreuses surcharges de la méthode statique `XmlReader XmlReader.Create()` retourne l'objet adéquate selon vos besoin.

D'après cette description, les lecteurs connaissant déjà XML peuvent penser que `XmlReader` est une implémentation du protocole *SAX* de lecture d'un document XML. Cette assertion est fausse et vous pouvez vous référer à l'article **Comparing XmlReader to SAX Reader** des **MSDN** pour comprendre les différences.

Voici un programme qui lit le document `books.xml` avec une instance d'une classe dérivée de `XmlReader`. Notez la gestion d'un mécanisme d'indentation pour exposer la structure hiérarchique lors de l'affichage :

*Exemple 21-4 :*

```
using System ;
using System.Xml ;
class Program {
   static void Main() {
      XmlReader xtr =  XmlReader.Create(@"C:\books.xml") ;
      string sIndent = string.Empty ;
      string sElem = string.Empty ;
      while ( xtr.Read() ) {
         if ( xtr.NodeType == XmlNodeType.Element ) {
            sIndent = string.Empty ;
            for (int i = 0 ; i < xtr.Depth ; i++) sIndent += "   " ;
            sElem = xtr.Name ;
            if ( xtr.MoveToFirstAttribute() )
```

```
                do
                   Console.WriteLine("{0}{1} Attr:{2}" ,
                                     sIndent , sElem , xtr.Value ) ;
                while ( xtr.MoveToNextAttribute() ) ;
            }
            else if (xtr.NodeType == XmlNodeType.Text)
               Console.WriteLine("{0}{1} Val:{2}",sIndent,sElem,xtr.Value) ;
         }
      }
   }
```

Cet exemple affiche ceci :

```
bookstore Attr:http://www.contoso.com/books
   book Attr:autobiography
   book Attr:1981
   book Attr:1-861003-11-0
      title Val:The Autobiography of Benjamin Franklin
         first-name Val:Benjamin
         last-name Val:Franklin
      price Val:8.99
   book Attr:novel
   book Attr:1967
   book Attr:0-201-63361-2
      title Val:The Confidence Man
...
```

La classe `XmlReader` présente de nombreuses facilités pour la lecture des données telles que le typage des valeurs récupérées, la prise en compte des espaces de noms, éviter les espaces et les commentaires etc. Ceci est possible grâce à de multiples méthodes telles que `MoveToContent()`, `ReadContentAsBoolean()`, `ReadStartElement()` etc.

## *Validation des données durant la lecture*

Pour valider les données durant leur lecture, il faut avoir recours à une surcharge de la méthode `XmlReader.Create()` qui accepte dans sa liste d'arguments une instance de la classe `System.Xml.XmlReaderSettings`. Cette instance permet de préciser les vérifications qui doivent être faites lors de la lecture des données. Reprenons l'Exemple 21-4 en validant les données XML en entrée auprès de du schéma `books.xsd` (celui de l'Exemple 21-2). Notez la nécessité de fournir une méthode *callback* (en l'occurrence la méthode `ValidatingProblemHandler()`) pour récupérer les problèmes de validation :

*Exemple 21-5 :*

```
...
using System.Xml.Schema ;
class Program {
   static void Main() {
      XmlReaderSettings settings = new XmlReaderSettings() ;
      settings.Schemas.Add(String.Empty,
                           @"C:\books.xsd") ;
```

```
      settings.Schemas.Compile();
      settings.ValidationType = ValidationType.Schema;
      settings.ValidationEventHandler += ValidatingProblemHandler;
      settings.ValidationFlags =
         XmlSchemaValidationFlags.ReportValidationWarnings;
      XmlReader xtr = XmlReader.Create(@"C:\books.xml",settings) ;
      ...
   }
   static void ValidatingProblemHandler(object sender,
                                        ValidationEventArgs e) {
      if ( e.Severity == XmlSeverityType.Warning ) {
         Console.Write("WARNING: ") ;  Console.WriteLine(e.Message) ;
      } else if ( e.Severity == XmlSeverityType.Error ) {
         Console.Write("ERROR: ") ;    Console.WriteLine(e.Message) ;
      }
   }
}
```

## *Ecriture des données avec XmlWriter*

Dans le même esprit que la classe `XmlReader`, la classe `System.Xml.XmlTextWriter` permet de créer un fichier XML en précisant les éléments les uns après les autres. Voici un petit exemple qui crée un fichier XML (nous avons volontairement indenté le code pour en faciliter la lecture) :

*Exemple 21-6 :*

```
using System.Xml ;
using System.Text ;
class Program {
   static void Main() {
      XmlTextWriter xtw = new
         XmlTextWriter(@"C:\book2.xml", Encoding.UTF8) ;
      xtw.Formatting = Formatting.Indented;
      xtw.WriteStartDocument(true) ;
        xtw.WriteStartElement("books") ;
          xtw.WriteStartElement("book") ;
            xtw.WriteAttributeString("ISBN", "2-84177-245-4") ;
            xtw.WriteElementString("title", "Pratique de .NET et C#") ;
          xtw.WriteEndElement() ;
        xtw.WriteEndElement() ;
      xtw.WriteEndDocument() ;
      xtw.Flush() ;
      xtw.Close() ;
   }
}
```

Voici le fichier XML créé :

*Exemple :* *books2.xml*

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<books>
  <book ISBN="2-84177-245-4">
    <title>Pratique de .NET et C#</title>
  </book>
</books>
```

# *Parcours et édition d'un document XML avec DOM (XmlDocument)*

## *Chargement et parcours d'un document avec XmlDocument*

La classe `System.Xml.XmlDocument` représente l'implémentation de la norme W3C *DOM* (*Document Object Model*). Cette norme décrit la représentation d'un document au format XML en mémoire sous la forme d'un arbre d'objets. Cette arborescence est constituée de nœuds instances de `System.Xml.XmlNode` (pour les feuilles) et `System.Xml.XmlNodeList` (pour les composites). La classe `XmlDocument` dérive de la classe `XmlNode` étant donnée qu'on peut toujours voir un document XML comme son nœud racine.

Contrairement à la classe `XmlTextReader`, l'arborescence est complètement construite lors du chargement du document XML par l'appel à une méthode `Load()` sur une instance de `XmlDocument`. Il vaut mieux avoir une idée de la taille du document à charger, car dans le cas d'un fichier volumineux le chargement complet en mémoire peut être prohibitif. Si le document XML source a au moins une erreur de syntaxe, une exception est lancée. Aussi, il est recommandé d'appeler la méthode `Load()` au sein d'un bloc `try/catch`.

Voici un programme qui parcourt récursivement la structure d'un `XmlDocument` initialisé au préalable avec le document `books.xml`. Cet exemple affiche le nom, la valeur et les attributs de chaque nœud avec une indentation pour montrer la structure hiérarchique :

*Exemple 21-7 :*

```
using System ;
using System.Xml ;
public class Program {
   static void DisplayNode(XmlNode xNode, string sIndent) {
      Console.WriteLine("{0}Node: {1}({2})",
                  sIndent, xNode.Name, xNode.Value) ;
      if (xNode.Attributes != null)
         foreach (XmlAttribute xAtt in xNode.Attributes)
            Console.WriteLine("{0} Attribute: {1}",
                        sIndent, xAtt.Value) ;
      if (xNode.HasChildNodes)
         foreach (XmlNode _xNode in xNode.ChildNodes)
            DisplayNode(_xNode, sIndent + "   ") ;
   }
   static public void Main() {
```

```
      XmlDocument xDoc = new XmlDocument() ;
      try { xDoc.Load(@"C:\books.xml") ; }
      catch { }
      foreach (XmlNode xNode in xDoc.ChildNodes)
         DisplayNode(xNode, string.Empty) ;
   }
}
```

Voici un extrait de l'affichage :

```
Node: xml(version="1.0" encoding="utf-8")
Node: #comment( This file represents a fragment of a book store
inventory database )
Node: bookstore()
   Node: book()
    Attribute: autobiography
    Attribute: 1981
    Attribute: 1-861003-11-0
      Node: title()
         Node: #text(The Autobiography of Benjamin Franklin)
      Node: author()
         Node: first-name()
            Node: #text(Benjamin)
         Node: last-name()
            Node: #text(Franklin)
      Node: price()
         Node: #text(8.99)
...
```

## *Edition et sauvegarde des données avec XmlDocument*

La classe `XmlDocument` présente des facilités d'édition avec des méthodes telles que `InsertAfter()`, `InsertBefore()`, `AppendChild()`, `CreateAttribute()` etc. Vous pouvez à tous moment récupérer une chaîne de caractères contenant le document XML avec la propriété `string InnerXml{get;set;}`. Vous pouvez aussi sauvegarder le document XML avec les différente surcharges de la méthode `Save()` (qui acceptent en entrée différents types de flots de données ou le nom d'un fichier cible).

## *Validation d'un arbre DOM*

La méthode `XmlDocument.Validate()` permet de valider un arbre DOM auprès d'un schéma XSD. Par exemple :

*Exemple 21-8 :*

```
using System ;
using System.Xml ;
using System.Xml.Schema ;
using System.Xml.XPath ;
public class Program {
```

```
   static public void Main() {
      XmlDocument xDoc = new XmlDocument() ;
      try { xDoc.Load(@"C:\books.xml") ; } catch { }
      xDoc.Schemas.Add( String.Empty, @"C:\books.xsd") ;
      xDoc.Schemas.Compile() ;
      ValidationEventHandler validator = ValidatingProblemHandler ;
      xDoc.Validate(validator) ;
   }
   static void ValidatingProblemHandler(object sender,
                                        ValidationEventArgs e) {
      if (e.Severity == XmlSeverityType.Warning) {
         Console.Write("WARNING: ") ; Console.WriteLine(e.Message) ;
      } else if (e.Severity == XmlSeverityType.Error) {
         Console.Write("ERROR: ") ; Console.WriteLine(e.Message) ;
      }
   }
}
```

Une version surchargée de la méthode `XmlDocument.Validate()` accepte en argument un nœud de l'arbre afin de procéder à une validation partielle de l'arbre.

### *Evènements de la classe XmlDocument*

La classe `XmlDocument` présente les évènements suivants qui vous permettent de déclencher une action lors d'un changement sur l'arbre DOM :

```
public event XmlNodeChangedEventHandler NodeChanged ;
public event XmlNodeChangedEventHandler NodeChanging ;
public event XmlNodeChangedEventHandler NodeInserted ;
public event XmlNodeChangedEventHandler NodeInserting ;
public event XmlNodeChangedEventHandler NodeRemoved ;
public event XmlNodeChangedEventHandler NodeRemoving ;
```

## *Parcours et édition d'un document XML avec XPath*

### *Appliquer une expression XPath sur un arbre DOM*

Grâce à la méthode `XmlNode.SelectNodes()` il est aisé de sélectionner un ensemble de noeuds avec une expression XPath. Ainsi, l'exemple suivant sélectionne tous les prénoms des auteurs contenus dans le document `books.xml` :

*Exemple 21-9 :*

```
using System.Xml ;
public class Program {
   static public void Main() {
      XmlDocument xDoc = new XmlDocument() ;
      try { xDoc.Load(@"C:\books.xml") ; } catch { }
      XmlNodeList books = xDoc.SelectNodes(
```

```
                              @"/bookstore/book/author/first-name");
      foreach (XmlNode book in books)
         System.Console.WriteLine(book.OuterXml) ;
   }
}
```

Cet exemple affiche ceci :

```
<first-name>Benjamin</first-name>
<first-name>Herman</first-name>
```

## *Parcours d'un document XPathDocument avec XPathNavigator*

Les instances de la classe `System.Xml.XPath.XPathNavigator` permettent de parcourir (et éventuellement de modifier) un arbre DOM chargé en mémoire au moyen d'expressions XPath. Vous pouvez récupérer un tel objet à partir de la méthode `XPathNavigator CreateNavigator()` de l'interface `IXPathNavigable`. Les classes `XmlDocument` et `System.Xml.XPath.XPathDocument` implémentent cette interface.

La classe `XPathDocument` est comparable à la classe `XmlDocument` car ses instances permettent de stocker un arbre DOM représentant un document XML. En revanche, un arbre DOM stocké dans une instance de `XPathDocument` est accessible en lecture seule. Aussi, une instance de `XPathNavigator` peut modifier un arbre DOM que si elle agit sur un document chargé avec une instance de `XmlDocument`. L'implémentation de `XPathNavigator` est cependant plus adaptée dans certaines situations car elle apporte en moyenne un gain de performance significatif.

L'exemple suivant montre comment parcourir récursivement à l'aide d'une instance de `XPathNavigator`, un document XML chargé dans une instance de `XPathDocument` :

*Exemple 21-10 :*

```
using System ;
using System.Xml ;
using System.Xml.XPath ;
class Program {
   static void Main() {
      XPathDocument doc = new XPathDocument(@"C:\books.xml") ;
      XPathNavigator navigator = doc.CreateNavigator();
      navigator.MoveToRoot() ;
      DisplayRecursive(navigator, string.Empty) ;
   }
   static public void DisplayRecursive(XPathNavigator navigator,
                                       string indent) {
      if ( navigator.HasChildren ) {
         navigator.MoveToFirstChild() ;
         DisplayNode(navigator,indent+"   ") ;
         DisplayRecursive(navigator, indent + "   ") ;
         navigator.MoveToParent() ;
      }
      while (navigator.MoveToNext()) {
         DisplayNode(navigator, indent) ;
```

```
            DisplayRecursive(navigator, indent) ;
         }
      }
      static private void DisplayNode(XPathNavigator navigator,
                                      string indent) {
            if (navigator.NodeType == XPathNodeType.Text)
               Console.WriteLine(indent+navigator.Value) ;
            else if (navigator.Name != String.Empty)
               Console.WriteLine(indent + "<" + navigator.Name + ">") ;
      }
   }
```

Cet exemple affiche ceci :

```
  <bookstore>
      <book>
         <title>
            The Autobiography of Benjamin Franklin
         <author>
...
```

## *Traversée d'une sélection XPath avec XPathNodeIterator*

Une alternative à la méthode `XmlNode.SelectNodes()` pour sélectionner un ensemble de noeuds avec une expression XPath est d'utiliser une instance de `XPathNodeIterator` obtenue à partir de la méthode `XPathNodeIterator XPathNavigator.Select(<< XPathExpression >>)`. Une instance de cette classe permet d'énumérer les éléments sélectionnés à partir d'une expression XPath. Pour chaque élément, la propriété `XPathNavigator XPathNodeIterator.Current{get;}` vous renvoie un navigateur positionné à l'élément courant. Tout ceci est exposé par l'exemple suivant :

*Exemple 21-11 :*

```
using System.Xml.XPath ;
class Program {
   static void Main() {
      XPathDocument document = new XPathDocument(@"C:\books.xml") ;
      XPathNavigator navigator = document.CreateNavigator() ;
      XPathNodeIterator iterator =
         navigator.Select(@"/bookstore/book/author/first-name") ;
      while( iterator.MoveNext() )
         System.Console.WriteLine("<" + iterator.Current.Name + ">" +
                                  iterator.Current.Value) ;
   }
}
```

Cet exemple affiche ceci :

```
<first-name>Benjamin
<first-name>Herman
```

## Edition d'un arbre DOM XmlDocument avec XPathNavigator

L'exemple suivant montre comment utiliser une instance de `XPathNavigator` afin de modifier un arbre DOM stocké dans une instance de `XmlDocument`. En l'occurrence, nous insérons un nouvel élément `<book>` dans notre document `XmlDocument` :

*Exemple 21-12 :*

```
using System ;
using System.Xml ;
using System.Xml.XPath ;
class Program {
   static void Main() {
      XmlDocument xDoc = new XmlDocument() ;
      try { xDoc.Load(@"C:\books.xml") ; } catch { }
      XPathNavigator navigator = xDoc.CreateNavigator() ;
      navigator.MoveToRoot() ;              // Sélectionne la racine.
      if (navigator.MoveToFirstChild())     // Sélectionne <bookstore>.
         if (navigator.MoveToFirstChild()) // Sélectionne
                                           // <book>Autobiography...
            navigator.InsertElementBefore( string.Empty, "book",
                                 string.Empty, "Pratique de .NET et C#");
      xDoc.Save(@"C:\new_books.xml") ;
   }
}
```

Voici un aperçu du fichier `books.xml` modifié :

*Exemple :* *Books.xml*

```
<?xml version="1.0" encoding="utf-8" ?>
<bookstore>
  <book>Pratique de .NET et C#</book>
  <book genre="autobiography" publicationdate="1981" ISBN="1-861003-11-0">
    <title>The Autobiography of Benjamin Franklin</title>
...
```

# Transformer un document XML avec XSLT

La classe `System.Xml.Xsl.XslCompiledTransform` peut être utilisée pour transformer un document XML au moyen d'un programme rédigé avec XSLT 1.0. Cette nouvelle classe du *framework* .NET 2.0 remplace la classe `XslTransform` maintenant devenue obsolète. Comme son nom l'indique, son atout principal et de compiler les programmes XSLT en code MSIL avant d'appliquer une transformation. Le coût initial de la compilation est rapidement amorti après quelques transformations.

L'exemple suivant applique le programme `books.xslt` (Exemple 21-3) sur le document `books.xml` (Exemple 21-1) et affiche le résultat sur la console :

*Exemple 21-13 :*

```
using System.Xml.Xsl ;
class Program {
   static void Main() {
      System.Xml.XmlDocument xDoc = new System.Xml.XmlDocument() ;
      xDoc.Load(@"C:\books.xml") ;
      XslCompiledTransform xTrans = new XslCompiledTransform();
      xTrans.Load(@"C:\books.xslt");
      xTrans.Transform(xDoc, null, System.Console.Out);
   }
}
```

Dans la dernière section du présent chapitre, nous exposons les facilités proposées par *Visual Studio* pour l'édition et le débogage d'un programme XSLT.

# *Ponts entre le relationnel et XML*

## *Obtenir un document XML à partir d'un DataSet*

Vous pouvez aisément obtenir un document XML décrivant les données contenues dans les tables d'un `DataSet`. Pour cela, il suffit d'utiliser la méthode `WriteXml()` de la classe `DataSet`. Vous pouvez aussi obtenir un schéma XSD décrivant les schémas des tables contenues dans un `DataSet` avec la méthode `WriteXmlSchema()` de la classe `DataSet`. Voici un exemple qui remplit un `DataSet` à partir d'une base de données (celle décrite page 710) puis affiche les données et le schéma sur la console, aux formats XML et XSD :

*Exemple 21-14 :*

```
using System.Data ;
using System.Data.SqlClient ;
class Program {
   static void Main() {
      string sCnx =
         "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION" ;
      using( SqlConnection cnx = new SqlConnection(sCnx) ) {
         using( SqlDataAdapter dataAdapter = new SqlDataAdapter() ) {
            DataSet dataSet = new DataSet() ;
            string sCmd = "SELECT * FROM EMPLOYES" ;
            dataAdapter.SelectCommand = new SqlCommand(sCmd, cnx) ;
            dataAdapter.Fill(dataSet, "EMPLOYES") ;

            dataSet.WriteXml( System.Console.Out );
            System.Console.WriteLine("------------------------------") ;
            dataSet.WriteXmlSchema( System.Console.Out );
         } // end using SqlDataAdapter
      } // end using SqlConnection
   }
}
```

Voici un extrait de l'affichage de ce programme :

```
<NewDataSet>
  <EMPLOYES>
    <EmployeID>1</EmployeID>
    <DepID>MKT</DepID>
    <Nom>Lafleur</Nom>
    <Prénom>Léon</Prénom>
    <Tél>0497112233</Tél>
  </EMPLOYES>
  ...
</NewDataSet>------------------------------
<?xml version="1.0" encoding="ibm850"?>
<xs:schema id="NewDataSet" xmlns=""
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="NewDataSet" msdata:IsDataSet="true"
                                msdata:Locale="fr-FR">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="EMPLOYES">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="EmployeID" type="xs:int" minOccurs="0" />
              <xs:element name="DepID" type="xs:string" minOccurs="0" />
              <xs:element name="Nom" type="xs:string" minOccurs="0" />
              <xs:element name="Prénom" type="xs:string" minOccurs="0" />
              <xs:element name="Tél" type="xs:string" minOccurs="0" />
...
```

## *Remplir un DataSet à partir d'un document XML*

Il est possible d'insérer des lignes dans une table d'un `DataSet` à partir d'un fichier XML. Dans l'exemple suivant, nous ajoutons les données sur de nouveaux employés dans la base de données décrite page 710. Les données sources sur ces employés sont contenues dans le fichier *DataFile.xml* :

*Exemple 21-15 :*

```
using System.Data ;
using System.Data.SqlClient ;
class Program {
   static void Main() {
      string sCnx =
         "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION" ;
      using( SqlConnection cnx = new SqlConnection(sCnx) ) {
         using( SqlDataAdapter dataAdapter = new SqlDataAdapter() ) {
            DataSet dataSet = new DataSet() ;

            // Génère automatiquement les commandes pour
```

```
            // mettre à jour la base.
            string sCmd = "SELECT * FROM EMPLOYES WHERE EmployeID=-1" ;
            dataAdapter.SelectCommand = new SqlCommand(sCmd, cnx) ;
            SqlCommandBuilder cmdBuilder =
               new SqlCommandBuilder(dataAdapter) ;
            // Construction de la table EMPLOYES et insertion des
            // lignes à  partir du fichier 'DataFile.xml'.
            dataSet.ReadXml(@"C:/DataFile.xml");

            // Met à jour la base.
            dataAdapter.Update(dataSet, "EMPLOYES") ;
         } // end using SqlDataAdapter
      } // end using SqlConnection
   }
}
```

Voici le fichier *DataFile.xml.* Il n'est pas nécessaire de préciser une valeur pour le champ `EmployeID` puisque celle-ci est déterminée par la base de données.

*Exemple 21-16 :* *`DataFile.xml`*

```
<?xml version="1.0" encoding="utf-8" ?>
<NewDataSet>
   <EMPLOYES>
      <DepID>MKT</DepID>
      <Nom>Pointcarre</Nom>
      <Prénom>Paul</Prénom>
      <Tél>0497112283</Tél>
   </EMPLOYES>
   <EMPLOYES>
      <DepID>DEV</DepID>
      <Nom>Plom</Nom>
      <Prénom>Olivier</Prénom>
      <Tél>0497112285</Tél>
   </EMPLOYES>
</NewDataSet>
```

## *La classe System.Xml.XmlDataDocument*

Nous avons vu dans les deux sections précédentes comment transférer des données au format XML dans les tables d'un `DataSet` et vice-versa. La classe `System.Xml.XmlDataDocument` (qui dérive de la classe `XmlDocument`) a été spécialement conçue à cet effet. Concrètement, on lie une instance de `XmlDataDocument` avec un `DataSet`, puis on manipule les données par l'intermédiaire du `XmlDataDocument` ou par l'intermédiaire du `DataSet`. Il faut bien comprendre qu'en interne, ce sont les mêmes données qui sont manipulées. Les changements faits par l'intermédiaire du `DataSet` sont immédiatement visibles par l'intermédiaire du `XmlDataDocument` et vice versa.

Il est légitime de se demander pourquoi utiliser la classe `XmlDataDocument` alors que les deux sections précédentes ont montré que la classe `DataSet` pouvait gérer le format XML. Nous pouvons identifier les trois raisons suivantes :

- L'utilisation de la classe `XmlDataDocument` permet de faire des requêtes XPath sur les données.
- L'utilisation de la classe `XmlDataDocument` permet de rester fidèle à un document XML source. Si vous chargez un document XML dans un `DataSet` puis que vous le sauvez, il peut y avoir des différences de formatage entre les documents XML sources et destinations. Avec l'utilisation de la classe `XmlDataDocument` le document XML destination sera strictement identique au document XML source (par exemple les espaces, l'ordre des éléments ou les commentaires n'auront pas été modifiés).
- L'utilisation de la classe `XmlDataDocument` permet de tenir compte des relations entre les tables du `DataSet` lors du formatage XML. Concrètement, dans une relation « *one to many* », les éléments enfants d'un élément parent seront physiquement inclus dans l'élément parent dans le document XML. La notion de relation entre tables d'un `DataSet` est présentée page 728. Avec cette technique, une relation est prise en compte seulement si sa propriété `Nested` est positionnée à `true`, ce qui n'est pas le cas par défaut.

## *XML et SQL Server*

Depuis la version 2000, le SGBD *SQL Server* présente un composant nommé *SQLXML* qui permet de formater les requêtes SQL ainsi que les réponses au format XML. SQLXML se retrouve aussi dans le serveur IIS. Ainsi, il est possible d'envoyer des requêtes XML et de recevoir des réponses XML par HTTP. Autrement dit, cette infrastructure permet de concevoir des *services web* basiques d'interrogation de bases de données.

Le fournisseur de données SqlClient de ADO.NET permet d'exploiter SQLXML. L'exemple suivant permet de lister les lignes de la table `EMPLOYES` au format XML. Notez l'ajout de l'expression `FOR XML AUTO` à la fin de notre requête SQL. Cette syntaxe constitue une extension du langage T-SQL et permet de préciser que l'on souhaite avoir le résultat de la requête dans un format XML :

*Exemple 21-17 :*

```
using System.Data.Common ;
using System.Data.SqlClient ;
using System.Xml ;
class Program {
   static void Main() {
      string sCnx =
        "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION" ;
      string sCmd = "SELECT * FROM EMPLOYES FOR XML AUTO" ;
      using (DbConnection cnx = new SqlConnection(sCnx)) {
         using (SqlCommand cmd = new SqlCommand(sCmd,
                                                cnx as SqlConnection)) {
            cnx.Open() ;
            System.Xml.XmlReader reader = cmd.ExecuteXmlReader() ;
            while ( reader.Read() )
               System.Console.WriteLine( reader.ReadOuterXml() ) ;
            reader.Close() ;
         } // end using cmd.
      } // end using cnx.
      System.Console.Read() ;
```

```
    }
}
```

Cet exemple affiche ceci :

```
<EMPLOYES EmployeeID="1" DepID="MKT" Nom="Lafleur" Prénom="Léon"
Tél="0497112233"/>
<EMPLOYES EmployeeID="3" DepID="DEV" Nom="Schmol" Prénom="Jean" />
...
```

La version 2005 de *SQL Server* apporte des nouvelles facilités quant à l'intégration de données XML dans une base de données relationnelle. Notamment, une colonne d'une table peut être typée en XML. Chaque élément d'une telle colonne est un document XML. Le type du *framework* .NET correspondant à une telle colonne est `System.Data.SqlTypes.SqlXml`. Il y a aussi possibilité de typer fortement une colonne XML par un schéma XSD et vous pouvez effectuer des requêtes *XQuery* sur les éléments d'une telle colonne. Une présentation détaillée de ces fonctionnalités de *SQL Server 2005* dépasse le cadre du présent ouvrage.

# *Ponts entre l'objet et XML (sérialisation XML)*

## *La classe System.Xml.XmlSerializer*

Grâce à la classe `System.Xml.XmlSerializer` vous avez la possibilité de sérialiser au format XML n'importe quel objet .NET. Ce mécanisme de sérialisation admet cependant quelques limitations :

- `XmlSerializer` ne sérialise que les champs et attributs publics contrairement à la sérialisation d'objets binaire décrite page 790.
- `XmlSerializer` ne tient pas compte de l'attribut `[Serializable]` ou de l'interface `ISerializable`.
- `XmlSerializer` ne peut sérialiser un graphe d'objets qui admet au moins une référence circulaire.

Voici un programme qui illustre l'utilisation de la classe `XmlSerializer` en sauvant l'état d'une instance de la classe `Entreprise` dans un fichier XML, et en chargeant cet état dans une autre instance de la classe `Entreprise`.

*Exemple 21-18 :*

```
using System.Xml ;
using System.Xml.Serialization ;
using System.IO ;
public class book {
   public string genre  { get { return m_genre ; }
                          set { m_genre = value ; } }
   private string m_genre ;
   public string title  { get { return m_title ; }
                          set { m_title = value ; } }
   private string m_title ;
   public decimal price { get { return m_price ; }
```

```
                          set { m_price = value ; } }
   private decimal m_price ;
}
public class Program {
   static public void Main() {
      book b1 = new book() ;
      b1.genre = "autobiography" ;
      b1.title = "The Autobiography of Benjamin Franklin" ;
      b1.price = 8.99M ;
      //Sauve une instance de la classe book dans le fichier 'book.xml'.
      FileStream fs1 = File.OpenWrite("book.xml");
      XmlSerializer xmls = new XmlSerializer(typeof(book));
      xmls.Serialize(fs1, b1);
      fs1.Close() ;
      // Crée une instance de la classe book...
      // ...à partir du fichier 'book.xml'.
      FileStream fs2 = File.OpenRead("book.xml");
      book b2 = (book)xmls.Deserialize(fs2);
      fs2.Close() ;
   }
}
```

Le fichier XML créé est celui-ci :

*Exemple :* *book.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<book xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <genre>autobiography</genre>
  <title>The Autobiography of Benjamin Franklin</title>
  <price>8.99</price>
</book>
```

Vous avez la possibilité de sérialiser une information de type nullable dans un document XML. Dans ce cas là, la valeur nulle sera prise si l'élément ou l'attribut est absent dans le document XML ou si l'élément est présent mais contient un attribut `xsi:nil="true"`. Dans le schéma XSD, le fait qu'un élément ou un attribut peut être nul se traduit par la présence de l'attribut `nil="true"` dans l'élément `xs:element` correspondant

Vous pouvez sérialiser un type générique. Cela ne pose aucun problème lors de la sérialisation mais il faut faire attention de spécifier les types paramètres adéquats lors de la construction d'un `XmlSerializer` destiné à la désérialisation.

## *Attributs spécialisés pour la sérialisation XML*

Ne confondez pas la notion d'attribut .NET et la notion d'attribut XML qui sont différentes.

Vous avez la possibilité d'utiliser des attributs .NET pour modifier le comportement par défaut de la sérialisation dans un fichier XML. En général il faut se plier à un schéma XSD lorsque

l'on sérialise un objet au format XML. Par exemple vous pourriez souhaiter ne pas sérialiser la propriété `price` et faire en sorte que la propriété `genre` soit un attribut XML et non un élément. Voici les attributs .NET que vous pouvez utiliser :

- `XmlRoot` : Permet d'identifier une classe ou une structure comme le nœud racine. En général on utilise cet attribut pour assigner un nom différent du nom de la classe à l'élément XML correspondant.
- `XmlElement` : Spécifie qu'un champ ou une propriété publique doit être sérialisé comme un élément. Comme il s'agit du comportement par défaut, on utilise en général cet attribut pour assigner un nom différent du nom du champ ou de la propriété, à l'élément XML correspondant.
- `XmlAttribute` : Spécifie qu'un champ ou une propriété publique doit être sérialisé comme un attribut XML et non comme un élément, ce qui est le comportement par défaut. On peut en profiter pour assigner un nom différent du nom du champ ou de la propriété, à l'élément XML correspondant.
- `XmlArray` : Spécifie qu'un champ ou une propriété publique doit être sérialisé comme un tableau. En général on utilise cet attribut pour faire en sorte qu'un tableau d'objets soit sérialisé.
- `XmlArrayItem` : Spécifie que les instances d'un type peuvent être sérialisées dans un tableau.
- `XmlIgnore` : Indique qu'il ne faut pas sérialiser un champ ou une propriété publique.

Modifions la classe `book` avec certains de ces attributs :

*Exemple 21-19 :*

```
...
[XmlRoot(ElementName = "livre")]
public class book {
   [XmlAttribute(AttributeName = "genre")]
   public string genre  { get { return m_genre ; }
                          set { m_genre = value ; } }
   private string m_genre ;
   [XmlElement(ElementName = "titre")]
   public string title  { get { return m_title ; }
                          set { m_title = value ; } }
   private string m_title ;
   [XmlIgnore]
   public decimal price { get { return m_price ; }
                          set { m_price = value ; } }
   private decimal m_price ;
}
...
```

Le fichier XML produit est maintenant celui-ci :

*Exemple :* *book.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<livre xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      genre="autobiography">
  <titre>The Autobiography of Benjamin Franklin</titre>
</livre>
```

## *L'outil sgen.exe*

Lorsque vous sérialisez au format XML un objet dont le type n'a pas encore eu d'instances sérialisées, la classe `XmlSerializer` crée en interne une nouvelle classe spécialisée dans la sérialisation des instances de ce type. Vous pouvez améliorer les performances en pré-générant ces classes spécialisées dans la sérialisation XML grâce à l'outil `sgen.exe` (*XML Serializer Generation Tool*). Cet outil accepte un assemblage `XXX` en entrée (avec option `/a`) et crée un nouvel assemblage `XXX.XmlSerializers.dll` qui contient ces classes spécialisées dans la sérialisation XML. Dans le cas de notre class `book` de l'Exemple 21-19, une classe `Microsoft.Xml.Serialization.GeneratedAssembly.bookSerializer` est créée et vous pouvez l'utiliser comme ceci :

*Exemple 21-20 :*

```
using Microsoft.Xml.Serialization.GeneratedAssembly;
...
public class Program {
   static public void Main() {
      ...
      FileStream fs1 = File.OpenWrite("book.xml") ;
      bookSerializer bookS = new bookSerializer();
      bookS.Serialize(fs1, b1);
      ...
      book b2 = (book)bookS.Deserialize(fs2) ;
      fs2.Close() ;
   }
}
...
```

## *L'outil xsd.exe*

Nous avons montré que l'utilisation de certains attributs .NET permet de se plier à un schéma XSD lors de la sérialisation d'un objet au format XML. L'outil `xsd.exe` fourni avec le *framework* .NET va plus loin et permet les opérations suivantes :

- `xsd.exe` peut générer le code d'une classe C# (dérivant éventuellement de la classe `DataSet`) à partir d'un schéma XSD. Notamment, en page 731 nous expliquons comment cet outil permet de créer des `DataSet` typés.
- `xsd.exe` permet de générer un schéma XSD à partir d'un fichier XML ou à partir de types contenus dans un assemblage.

L'utilisation de l'outil `xsd.exe` est décrite dans l'article **XML Schema Definition Tool (Xsd.exe)** des **MSDN**.

# *Visual Studio .NET et XML*

## *Visualisation et édition des documents XML et XSD*

*Visual Studio* permet d'éditer des documents XML sous la forme de documents textes. Il sait détecter les erreurs lexicales et syntaxiques XML. Lorsque de telles erreurs sont détectées, elles sont soulignées dans les documents et reportées dans la liste des erreurs. Il est possible d'inférer un schéma XSD à partir du document XML couramment visualisé. Il suffit pour cela d'aller dans le menu *XML* ▶ *Create Schema*.

Les schémas XSD sont quant à eux visualisables et éditables par l'intermédiaire d'un *designer* graphique exposé par la Figure 21-1 :

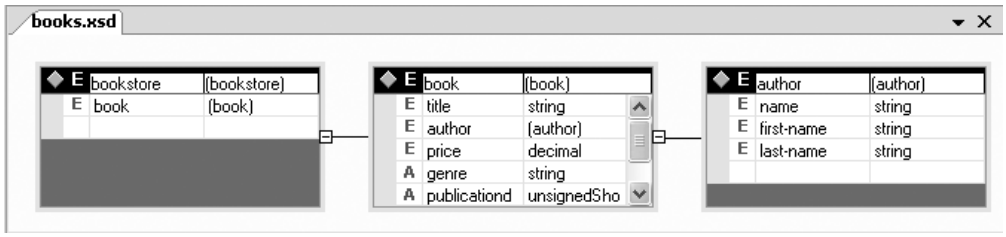


*Figure 21-1 : designer de schémas XSD*

*Visual Studio* crée un fichier d'extension `.xsx` pour chaque schéma XSD visualisé par cet éditeur. Ce fichier contient les informations exploitées par l'éditeur pour organiser la vue du document (coordonnées des tables, niveau de zoom etc). Notez qu'il est toujours possible de retourner à une vue textuelle d'un schéma XSD.

## *Validation d'un document XML auprès d'un schéma XSD*

Pour que *Visual Studio* puisse effectuer une telle validation, il faut d'abord qu'il ait accès au schéma XSD concerné. *Visual Studio* tient compte des schémas XSD contenus dans le projet courant et des schémas XSD stockés dans les répertoire `[Rep d'installation de VS]\Xml\Schemas` et `[Rep d'installation de VS]\Common7\IDE\Policy\Schemas`. Vous pouvez associer un schéma XSD connu par *Visual Studio* à un document XML d'une manière naturelle en précisant l'espace de nom cible du schéma XSD dans le document XML. Revoyons nos fichiers exemples `books.xml` et `books.xsd` :

*Exemple 21-21 :* *books.xml*

```
<?xml version="1.0" encoding="utf-8" ?>
<bookstore xmlns ="http://www.contoso.com/books">
  <book genre="autobiography" publicationdate="1981" ISBN="1-861003-11-0">
...
```

*Exemple 21-22 :* *books.xsd*

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
```

```
            elementFormDefault="qualified"
            targetNamespace="http://www.contoso.com/books"
            xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="bookstore">
...
```

Vous pouvez aussi associer un schéma XSD à un document XML par l'intermédiaire du menu *propriété* ► *Schemas* du document XML. Ce menu propose une fenêtre d'aide au choix du schéma XSD parmi tous ceux connus de *Visual Studio*.

Lorsqu'un schéma XSD est associé à un document XML, *Visual Studio* réalise non seulement la validation en détectant les erreurs et en les mettant en évidence, mais il active aussi le mécanisme très pratique d'*intellisense* pour faciliter l'édition du document XML.

## *Édition et débogage d'un programme XSLT*

Comme pour tous documents XML, *Visual Studio* vérifie les erreurs lexicales et syntaxiques lorsque vous éditez un programme XSLT. En outre, L'éditeur XML peut valider en temps réel votre document auprès du fichier `xslt.xsd` qui définit le schéma de ce langage. Il suffit pour cela d'inclure l'espace de noms `http://www.w3.org/1999/XSL/Transform` dans l'élément racine.

*Visual Studio* vous permet aussi de déboguer vos programmes XSLT de deux façons :

- Durant le débogage d'un programme .NET lors de l'appel à la méthode `XslCompiledTransform.Transform()` le débogueur va directement au point d'entrée du programme XSLT. Ceci est possible seulement si vous avez utiliser le constructeur `XslCompiledTransform(bool enableDebug)` avec la valeur `true` pour construire votre instance.
- Directement à partir de *Visual Studio* en utilisant le menu *XML* ► *Debug XSLT* qui est disponible lors de l'édition d'un programme XSLT. Pour exploiter cette possibilité, il faut au préalable avoir fourni un fichier XML en entrée et un fichier texte en sortie (qui accueillera le produit de la transformation XSLT) dans les sous menu *input* et *output* du menu *propriété* du document XSLT.

# 22

# .NET Remoting

## Introduction

### Qu'est ce que .NET Remoting ?

Nous avons défini la notion de domaine d'application page 87 comme étant un conteneur à assemblage durant l'exécution. Nous rappelons qu'un processus contient un ou plusieurs domaines d'applications, isolés les uns des autres par le CLR. L'assemblage `mscorlib` contenant les types de bases de la plateforme .NET est physiquement chargé dans le processus hors de tout domaine d'application. L'isolation entre les domaines d'application se fait principalement au niveau des types, de la sécurité et de la gestion des exceptions. Cette isolation ne se fait pas au niveau des threads.

Si vous avez assimilé cette notion de domaine d'application, vous pouvez comprendre la définition de .NET Remoting qui est :

> .NET Remoting est l'infrastructure de la plateforme .NET qui permet à des objets situés dans des domaines d'applications différents, de pouvoir se connaître et de pouvoir communiquer entre eux. L'objet appelant est nommé client, l'objet appelé est nommé serveur ou objet serveur.

Deux domaines d'applications différents peuvent se trouver :

- dans le même processus ;
- dans deux processus différents sur la même machine ;
- dans deux processus différents sur deux machines différentes.

.NET Remoting masque au développeur ces trois aspects du problème. La localisation d'un objet se fait en général après la compilation des sources, pendant l'installation de l'application. Plus précisément, l'information « tel objet est accessible sur telle machine » peut ne pas faire partie du code source. Dans ce cas cette information se situe dans un fichier de configuration accessible par le client de l'objet.

Nous précisons, à l'attention des développeurs ayant déjà utilisé les technologies *Microsoft* que .NET Remoting peut être vu comme le successeur de *DCOM*.

## *FAQ*

**Q : Peut-on choisir le protocole de communication sous-jacent à la communication entre deux objets ?**

R : Oui. Avec .NET Remoting chaque protocole de communication est encapsulé dans un objet appelé *canal* (*channel* en anglais). Les canaux encapsulant les protocoles HTTP, TCP ainsi qu'un protocole de communication entre processus d'une même machine (IPC pour *Inter Processus Communication*) sont implémentés nativement. Vous pouvez néanmoins développer vos propres canaux pour d'autres protocoles.

**Q : Sous quelle forme les données nécessaires à un appel de méthode transitent-elles sur le réseau ?**

R : Les données nécessaires à un appel de méthode sont principalement les valeurs des paramètres entrants, un identificateur de la méthode, un identificateur de l'objet avant l'appel et les valeurs des paramètres sortants au retour de l'appel. En .NET Remoting les objets responsables de l'emballage de ces données sont appelés *formateurs* (*formaters* en anglais). Les formateurs emballant les données dans un format binaire ou dans un format XML nommé SOAP sont implémentés nativement. Vous pouvez néanmoins développer vos propres formateurs pour vos besoins spécifiques (cryptage des données, élimination de redondances etc).

**Q : L'architecture utilisant ces objets formateurs et canaux est-elle aussi utilisée lorsque le client et l'objet serveur sont dans le même domaine d'application ou dans le même processus ?**

Dans ces deux cas, il n'est évidemment pas nécessaire d'utiliser un canal qui utilise le réseau, et .NET Remoting adopte automatiquement ce comportement. Il existe donc un canal spécifique aux appels au sein du même espace d'adressage.

**Q : Comment le code source du client, qui n'est pas dans le même assemblage que la classe représentant l'objet serveur, peut-il connaître la classe de cet objet ?**

R : Avec .NET Remoting, au moins trois techniques sont envisageables. Soit l'assemblage client référence l'assemblage serveur à la compilation. Dans ce cas l'assemblage serveur doit être installé avec l'assemblage client. Cette technique est donc très contraignante. On lui préfère l'encapsulation d'interfaces dans un assemblage tierce, qui est référencé à la fois par le serveur et le client. Une 3ème technique permet de construire automatiquement cet assemblage tierce à partir de l'assemblage du serveur, même si les classes de celui-ci n'implémentent pas d'interfaces. Cette 3ème technique nécessite l'utilisation d'un outil du *framework* .NET.

**Q : Ne serait-il pas plus simple de récupérer une copie de l'objet serveur dans le domaine d'application du client, et de travailler sur cette copie, plutôt que d'utiliser le réseau pour chaque appel de méthode ?**

R : .NET Remoting offre cette possibilité et la réponse à la question est : ça dépend. Ceci implique que l'assemblage contenant la classe de l'objet serveur est accessible par le client et ce n'est en général pas souhaitable. En outre, beaucoup d'objets serveurs ne peuvent pas être déplacés dans un autre domaine d'application. La raison classique est que ces objets serveurs contiennent des références vers d'autres objets du domaine, qui sont inamovibles. Par exemple une instance de la classe `Thread` peut être considérée comme un objet inamovible de son processus. Enfin, dans le cas de la récupération de l'objet par le client, on se prive de la possibilité de pouvoir utiliser un objet d'une manière concurrente entre plusieurs clients.

**Q : Qui est responsable de la création d'un objet serveur ? Le client ou le serveur lui-même ?**

R : Les deux cas de figure sont envisageables avec .NET Remoting. Dans le cas où le serveur est responsable de la création d'un objet, cet objet est identifié par un URI. Le client contacte cet objet en utilisant cet URI, un peu comme vous contactez un proche en utilisant son numéro de téléphone. Dans ce cas, l'objet est effectivement créé lors du premier appel du client.

**Q : Qui est responsable de la destruction d'un objet serveur ? Le client ou le serveur lui-même ?**

R : Malgré un mécanisme de ping évolué, DCOM a montré qu'il ne faut pas faire confiance au client quant à la destruction d'un objet serveur. DCOM a aussi montré qu'un tel mécanisme de ping nuit gravement aux performances. Avec .NET Remoting, les objets serveurs sont automatiquement détruits par le serveur après une certaine durée durant laquelle l'objet n'a pas été utilisé. Un client qui essaye de contacter un objet qui n'existe plus récupère une exception.

**Q : .NET Remoting supporte-t-il les appels asynchrones ? (Reformulé : Y a t-il un mécanisme pour qu'un client ne soit pas obligé d'attendre que l'appel d'une méthode par .NET Remoting se termine avant de continuer ?)**

R : Oui. .NET Remoting supporte le mécanisme d'appel asynchrone décrit page 171. Lors d'un appel asynchrone, vous pouvez choisir ou non d'être averti par le serveur lorsque celui-ci a exécuté votre appel. Vous pouvez ainsi récupérer des informations relatives à votre traitement par le serveur, d'une manière asynchrone.

# *Marshaling By Reference (MBR)*

.NET Remoting présente deux solutions architecturalement très différentes, pour permettre à un client d'appeler une méthode sur un objet distant. Ces deux solutions portent les noms de *Marshalling By Value* (*MBV*) et *Marshalling By Reference* (*MBR*). Par défaut, les instances d'une classe ne sont pas utilisables d'une manière distante.

Le *Marshalling By Reference* (*MBR*) consiste à obtenir un nouvel objet appelé *proxy transparent* (*transparent proxy* en anglais) dans le domaine d'application du client. Pour le code du client, tout se passe comme si un proxy transparent était une référence classique vers un objet, d'où la notion de transparence. En fait même le compilateur `csc.exe` ne sait pas que certaines références seront en fait des proxys transparents à l'exécution.

Une condition nécessaire pour pouvoir utiliser dans un assemblage des proxys transparents à la place des références d'un type, est que les métadonnées de ce type soient accessibles à la compilation de l'assemblage. Nous exposerons plusieurs façons de réaliser cette condition.

À ce stade, les lecteurs curieux doivent sûrement se poser des questions comme :

- Qui est responsable de la création de l'objet distant ?
- Comment récupère-t-on un proxy transparent ?
- Comment la classe proxy fait-elle pour faire transiter les données entrantes et sortantes des appels sur un réseau ?

Toutes les réponses à ces questions se trouvent dans ce chapitre. L'analyse du programme suivant va apporter quelques éléments de réponse. Nous nous intéressons au cas où le client et la classe de l'objet distant sont dans le même assemblage et où les domaines d'application du client et de l'objet distant sont dans le même processus.

*Exemple 22-1 :* *MBRTest.cs*

```
using System ;
using System.Runtime.Remoting.Contexts ;
using System.Runtime.Remoting ;
using System.Threading ;

public class Foo : MarshalByRefObject {
   public void AfficheInfo(string s) {
      Console.WriteLine(s) ;
      Console.WriteLine("  Nom du domaine : " +
                AppDomain.CurrentDomain.FriendlyName) ;
      Console.WriteLine("  ThreadID      : " +
                Thread.CurrentThread.ManagedThreadId) ;
   }
}

public class Program {
   static void Main() {
      // obj1
      Foo obj1 = new Foo() ;
      obj1.AfficheInfo("obj1:") ;
      Console.WriteLine("  IsObjectOutOfAppDomain(obj1)=" +
         RemotingServices.IsObjectOutOfAppDomain(obj1)) ;
      Console.WriteLine("  IsTransparentProxy(obj1)=" +
         RemotingServices.IsTransparentProxy(obj1)) ;

      // obj2
      AppDomain appDomain = AppDomain.CreateDomain("Autre domaine.") ;
      Foo obj2 = (Foo)appDomain.CreateInstanceAndUnwrap(
            "MBRTest",  // Nom de l'asm contenant le type.
            "Foo") ;     // Nom du type.

      obj2.AfficheInfo("obj2:") ; // <- Ici, le code client ne sait pas
                                  // qu'il manipule un proxy transparent.
      Console.WriteLine("  IsObjectOutOfAppDomain(obj2)=" +
         RemotingServices.IsObjectOutOfAppDomain(obj2)) ;
      Console.WriteLine("  IsTransparentProxy(obj2)=" +
         RemotingServices.IsTransparentProxy(obj2)) ;
```

```
    }
}
```

Ce programme affiche :

```
obj1:
  Nom du domaine : MBRTest.exe
  ThreadID       : 6116
  IsObjectOutOfAppDomain(obj1)=False
  IsTransparentProxy(obj1)=False
obj2:
  Nom du domaine : Autre domaine.
  ThreadID       : 6116
  IsObjectOutOfAppDomain(obj2)=True
  IsTransparentProxy(obj2)=True
```

Notez l'utilisation des méthodes statiques `IsObjectOutOfAppDomain()` et `IsTransparentProxy()` de la classe `RemotingServices`.

La Figure 22-1 illustre l'architecture mise en place par le CLR pour exécuter ce programme. La notion de contexte est expliquée un peu plus loin dans le présent chapitre :

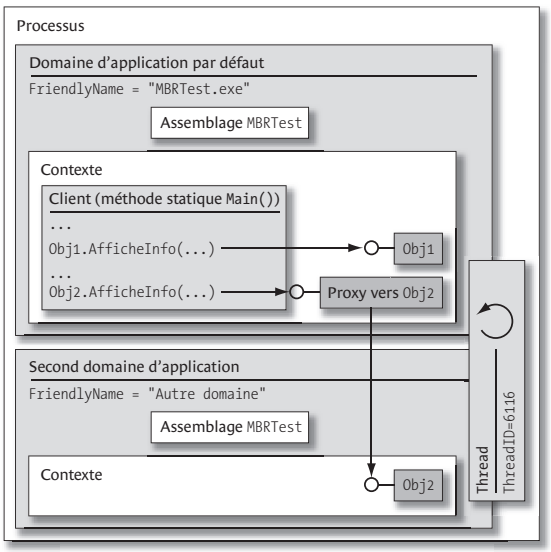


*Figure 22-1 : Exemple limité utilisant le Marshalling By Reference*

Dans le programme précédent, vous avez certainement remarqué que la classe `Foo` dérive de la classe `System.MarshalByRefObject`. Lorsque le compilateur JIT rencontre une référence typée par une classe qui dérive de `MarshalByRefObject`, il sait que cette référence peut être éventuellement un proxy transparent. Une conséquence est que le compilateur JIT s'interdit de mettre en ligne les méthodes appelées à partir d'une telle référence. Une telle pratique supposerait

que l'objet référencé n'est pas distant, ce qui peut être mis en défaut. Cela revient à dire que faire dériver une classe de la classe `MarshalByRefObject` permet d'indiquer au compilateur JIT qu'une instance de cette classe peut potentiellement être utilisée d'une manière distante grâce à un proxy transparent. Dans le cas où cette instance et son client sont dans le même domaine d'application, le client travaille avec une référence vers l'instance et non avec un proxy transparent.

# *Marshalling By Value (MBV) et serialisation binaire*

Le *Marshalling By Value* (*MBV*) consiste à fabriquer un clone de l'objet distant, dans le même domaine d'application que le client. Le CLR fait en sorte que ce clone ait exactement le même état que l'objet distant. Précisons que l'on désigne par *état d'un objet* à un moment donné, l'ensemble des valeurs prises par les champs de types valeur de l'objet. Selon l'application, l'état d'un objet contient aussi l'ensemble des états pris par les objets référencés par les champs de types référence de l'objet.

Le clone n'est pas un objet distant. Le client n'a pas besoin d'un proxy transparent pour y accéder. Notez qu'aucun constructeur n'est appelé sur le clone. Ce comportement est logique puisque le clone doit être exactement comme l'objet original, sur lequel un constructeur a déjà été appelé.

Une condition nécessaire pour utiliser la technique MBV est que le domaine d'application du client doit pouvoir charger l'assemblage contenant la classe de l'objet distant original. Eventuellement, le client et cette classe peuvent faire partie du même assemblage. Une autre condition nécessaire est que l'objet distant original ne doit pas contenir des références vers des objets qui ne sont pas clonables. On entend par là qu'il n'y a aucun intérêt à cloner certains objets. Par exemple, pourquoi cloner une instance de la classe `Thread` hors du processus qui contient physiquement le thread auquel fait référence cette instance ?

En interne, le CLR reflète l'état de l'objet distant dans un *stream* binaire puis envoie ce *stream* binaire au domaine d'application du client. Du côté client, le CLR reçoit ce *stream* binaire et reconstitue l'état dans le clone. Ces opérations sont appelées respectivement *sérialisation* et *désérialisation* de l'objet. En .NET, pour qu'un objet soit sérialisable il faut que sa classe ait l'attribut `System.Serializable` ou (exclusif) que sa classe étende l'interface `System.Runtime.Serialization.ISerializable`. Dans le premier cas vous signalez au CLR d'utiliser le mécanisme de sérialisation standard, dans le second cas vous avez la possibilité d'implémenter votre propre mécanisme de sérialisation. Par défaut tous les types primitifs sont sérialisables.

Une fois l'objet cloné dans le domaine d'application du client, les états du clone et de l'objet distant original sont indépendants. Les changements faits sur l'un ne seront pas reflétés sur l'autre. Le terme *MBValue* prend alors tout son sens puisque ce comportement est similaire au comportement d'un argument de type valeur passé dans une méthode. Les changements faits sur un tel argument dans la méthode ne se reflètent pas sur la variable initiale déclarée dans l'appelant. La Figure 22-2 illustre ceci.

Dans l'Exemple 22-1, il est intéressant de constater qu'en modifiant comme ceci la classe `Foo` de façon à en faire une classe MBV :

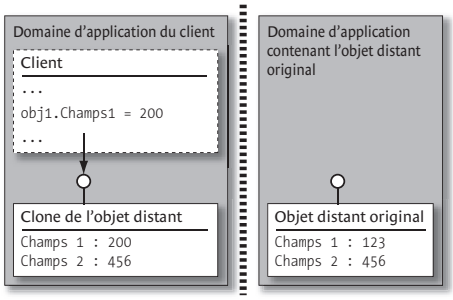


*Figure 22-2 : Marshalling By Value*

*Exemple 22-2 :* *MBVTest.cs*

```
...
[Serializable]
public class Foo{  // : MarshalByRefObject  <- en commentaire
...
```

...on obtient l'affichage suivant :

```
obj1:
  Nom du domaine : MBVTest.exe
  ThreadID       : 3620
  IsObjectOutOfAppDomain(obj1)=False
  IsTransparentProxy(obj1)=False
obj2:
  Nom du domaine : MBVTest.exe
  ThreadID       : 3620
  IsObjectOutOfAppDomain(obj2)=False
  IsTransparentProxy(obj2)=False
```

Ceci prouve que l'objet distant a bien été cloné dans un nouvel objet situé dans le domaine d'application du client.

Dans le chapitre consacré à la généricité, en page 499 nous expliquons que la sérialisation binaire sait traiter les types génériques.

## *Sérialisation tolérante aux changement de version*

Lorsque l'on exploite la sérialisation d'objet, on constate qu'un problème courant est du à l'évolution des classes à sérialiser et notamment, l'ajout de nouveaux champs. En effet, après mise à jour de l'application, lorsque l'on tente de désérialiser un objet une exception est lancée car les valeurs des nouveaux champs de la classe concernée ne sont pas fournies. Pour éviter ce problème vous pouvez marquer les nouveaux champs avec l'attribut `System.Runtime.Serialization.OptionalFieldAttribute`. Les champs concernés prendront la valeur par défaut de leur type.

L'espace de noms `System.Runtime.Serialization` contient aussi les quatre champs suivants qui permettent de marquer une méthode afin qu'elle soit invoquée durant une certaine étape de

la sérialisation/désérialisation. Vous pouvez ainsi profiter de telles méthodes pour fournir une valeur aux nouveaux champs :

- `OnDeserializingAttribute` : La méthode est appelée avant que l'état de l'objet désérialisé ait été récupéré et positionné.
- `OnDeserializedAttribute` : La méthode est appelée après que l'état de l'objet désérialisé ait été récupéré et positionné.
- `OnSerializingAttribute` : La méthode est appelée avant que l'état de l'objet ait été sérialisé.
- `OnSerializedAttribute` : La méthode est appelée après que l'état de l'objet ait été sérialisé.

# *La classe ObjectHandle*

Dans le code de l'exemple de la section de présentation MBR, à la place d'utiliser la méthode `CreateInstanceAndUnwrap()` on aurait pu écrire :

```
...
   ObjectHandle HObj2 = appDomain.CreateInstance("Remoting1","Foo") ;
   Foo obj2 = (Foo)HObj2.Unwrap() ;
   ...
```

Une instance de la classe `System.Runtime.Remoting.ObjectHandle` contient les informations nécessaires pour pouvoir utiliser un objet distant. En appelant la méthode `UnWrap()` vous récupérez un proxy transparent si l'objet distant est MBR ou un clone si l'objet distant est MBV.

Vous pouvez optimiser vos applications en utilisant le fait que .NET scinde l'opération de récupération d'un objet distant en deux étapes : récupération d'une instance de la classe `ObjectHandle` puis « *unwrappage* » de cette instance. En effet, le chargement des métadonnées de type décrivant la classe de l'objet distant ne se fait qu'à la seconde étape, lors du « *unwrappage* ». Si le client courant n'utilise pas l'objet distant et ne fait que le transmettre à une autre partie de l'application, il n'a pas besoin de réaliser le « '*unwrappage* ». On économise ainsi le chargement des métadonnées et donc, le chargement de l'assemblage. Voici un programme qui illustre ceci. Notez que ce programme exploite le fait que le *constructeur statique d'une classe* est appelé par le CLR, lorsque celui-ci charge les métadonnées de type de la classe concernée dans un domaine d'application.

*Exemple 22-3 :* *WrapTest.cs*

```
using System ;
using System.Runtime.Remoting ;

[Serializable]
public class Foo {
   // Constructeur de la classe.
   static Foo() {
      Console.WriteLine(
         "Chargement des métadonnées de 'Foo' dans le domaine:" +
         AppDomain.CurrentDomain.FriendlyName) ;
   }
   // Constructeur des instances de la classe.
   public Foo() {
```

```
      Console.WriteLine(
         "Appel au constructeur de 'Foo' dans le domaine:" +
         AppDomain.CurrentDomain.FriendlyName) ;
   }
}

public class Program {
   static void Main() {
      Console.WriteLine("-->Avant CreateDomain()") ;
      AppDomain appDomain = AppDomain.CreateDomain("Autre domaine.");
      Console.WriteLine("-->Avant CreateInstance()") ;
      ObjectHandle hObj = appDomain.CreateInstance("WrapTest", "Foo");
      Console.WriteLine("-->Après CreateInstance() et avant UnWrap()") ;
      Foo obj = (Foo) hObj.Unwrap();
      Console.WriteLine("-->Après UnWrap()") ;
   }
}
```

Voici ce que ce programme affiche :

```
-->Avant CreateDomain()
-->Avant CreateInstance()
Chargement des métadonnées de 'Foo' dans le domaine:Autre domaine.
Appel au constructeur de 'Foo' dans le domaine:Autre domaine.
-->Après CreateInstance() et avant UnWrap()
Chargement des métadonnées de 'Foo' dans le domaine:WrapTest.exe
-->Après UnWrap()
```

Ce programme illustre aussi le fait que le constructeur n'est pas appelé lors de la construction d'un clone.

Lorsque l'on exécute ce programme en faisant de `Foo` une classe MBR, l'avant dernière ligne de l'affichage n'apparaît pas. Ceci souligne clairement que l'objet proxy n'est pas du même type que l'objet distant.

# *Introduction à l'activation des objets*

Si vous avez lu les sections précédentes, vous pouvez entrer dans le vif du sujet, autrement dit : faire franchir les frontières virtuelles entre les processus et les frontières physiques entre les machines à vos appels d'objets dans le cas MBR, et à vos objets dans le cas MBV.

## *Eléments d'une architecture distribuée*

Dans l'architecture d'une application distribuée qui utilise .NET Remoting avec les objets serveurs en MBR, il y a quatre grandes sortes d'entités :

- Les clients, qui appellent les objets serveurs ;
- les hôtes, qui hébergent les objets serveur ;
- les métadonnées des types des objets serveur ;
- les implémentations des objets serveurs.

En règle générale, on isole chacune de ces entités dans un ou plusieurs assemblages. Cette règle n'est absolument pas une contrainte. Physiquement vous pouvez mélanger toutes sortes d'entités dans un assemblage. Cependant, utiliser cette règle présente des avantages que vous découvrirez dans les pages suivantes. En outre, nous expliquerons pourquoi et comment séparer les métadonnées de type des objets serveurs des implémentations de ceux-ci. La Figure 22-3 illustre la répartition des assemblages dans une architecture distribuée classique.

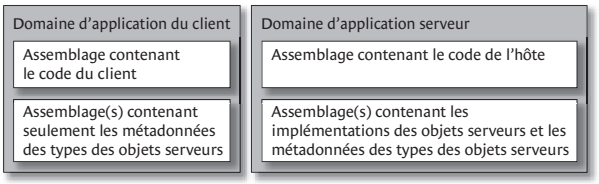


*Figure 22-3 : Eléments d'une architecture distribuée*

## *Responsabilité d'un hôte*

Un hôte a plusieurs responsabilités :

- Créer un ou plusieurs *canaux* (*channels* en anglais).
- Exposer des classes ou des objets serveurs accessibles par des clients, au travers d'URIs.
- Maintenir le processus qui contient les objets serveurs.

Il peut être surprenant qu'un hôte puisse exposer des objets mais aussi des classes. Cela résulte du fait qu'un objet serveur peut être soit activé par le serveur, soit activé par le client. Dans le premier cas un client doit connaître un objet pour pouvoir l'utiliser alors que dans le second cas, le client doit connaître une classe pour pouvoir activer une instance de cette classe.

## *Brève description des canaux*

Un canal est un objet qui permet de communiquer à travers le réseau. Il existe aussi des canaux spécialisés dans la communication intra processus. Un canal contient trois grands paramètres qui sont le port réseau de la machine qu'il utilise, le protocole de communication qu'il utilise (TCP, HTTP, IPC) et la façon dont les données sont formatées pour les faire transiter (format binaire standard, SOAP, format binaire ou XML propriétaire). Par défaut le formatage binaire est associé aux protocoles TCP et IPC, et le formatage SOAP est associé au protocole HTTP, mais ce comportement est facilement modifiable. Pour utiliser .NET Remoting il faut deux canaux avec le même protocole et la même façon de formater les données. Un canal du côté client et un canal du côté serveur. Pour aborder cette section vous n'avez pas besoin d'en savoir plus sur les canaux. Mais nous revenons sur ce sujet en détail un peu plus loin dans ce chapitre, dans une section consacrée aux canaux.

## *Appels synchrones, asynchrones et asynchrones sans retour*

Les appels d'objets distants peuvent être synchrones, asynchrones ou asynchrones sans retour. Il est appréciable de constater que la technique pour appeler un objet d'une manière asynchrone est la même, que l'objet soit distant ou non. Cette technique est décrite page 171.

## *Activation d'un objet vs. Création d'un objet*

Précisons qu'en .NET Remoting on parle plutôt d'*activation d'un objet* que de création d'un objet. Cela vient du fait que lors de la création d'un objet destiné à être utilisé d'une façon distante, il y a potentiellement un grand nombre d'actions qui ont lieu entre la demande de création et la disponibilité effective du nouvel objet. C'est le déroulement de ces actions qui porte le nom d'activation.

# *Service d'activation par le serveur (WKO)*

*Les sources et les exécutables de cette section sont dans les répertoires :*

*Chap22 .NET Remoting/WKO singlecall*

*Chap22 .NET Remoting/WKO singleton*

Une étape important, lorsque l'on spécifie une architecture distribuée, est de définir pour chaque objet distant qui, du client ou du serveur (i.e de l'hôte) va l'activer. .NET Remoting permet aux clients d'activer un objet distant sur le serveur ou d'utiliser un objet déjà existant sur le serveur. Nous soulignions que lorsqu'un objet est activé par le serveur, le client n'a pas à appeler le constructeur de la classe de l'objet. Cet aspect peut être déterminant dans le choix de l'activation par le serveur ou par le client. Nous exposons ici le cas où l'objet est activé par le serveur et nous présenterons ensuite le cas de l'activation par le client.

Dans le cas où un client utilise un objet distant activé par le serveur, il est fortement conseillé de faire en sorte que le client ne connaisse pas la classe dont l'objet distant est instance. Pour cela, il est judicieux de définir les interfaces supportées par la classe de l'objet distant dans un assemblage spécialement prévu à cet effet. Dans notre vue d'ensemble des éléments d'une architecture distribuée (celle de la Figure 22-3), cet assemblage joue le rôle des métadonnées de type des objets serveurs. Concrètement, cet assemblage sera présent à la fois dans le domaine d'application du client et dans le domaine d'application du serveur. Voici par exemple le code d'un tel assemblage :

*Exemple 22-4 :* `Interface.cs`

```
namespace NommageInterface {
   public interface IAdditionneur {
      double Add(double d1, double d2) ;
   }
}
```

Intéressons nous maintenant à l'hôte et aux classes des objets serveurs. Nous allons placer ces deux types d'entités dans le même assemblage. Cependant nous verrons qu'il existe des hôtes standard, et que pour les utiliser, il faut que les classes des objets serveurs soient isolées dans des assemblages. Voici le code d'un assemblage contenant à la fois la classe d'objet serveur et un hôte propriétaire. Notez bien l'ordonnancement des trois tâches dans le code de l'hôte : création d'un canal, activation d'un objet serveur, maintient du processus.

*Exemple 22-5 :* `Serveur.cs`

```
using System ;
using System.Runtime.Remoting ;
using System.Runtime.Remoting.Channels ;
```

```
using System.Runtime.Remoting.Channels.Http ;
using NommageInterface ;

namespace NommageServer {
   // Implémentation de la classe des objets serveurs.
   public class CAdditionneur : MarshalByRefObject, IAdditionneur {
      public CAdditionneur() {
         Console.WriteLine("CAdditionneur ctor") ;
      }
      public double Add(double d1, double d2) {
         Console.WriteLine("CAdditionneur Add( {0} + {1} )", d1, d2) ;
         return d1 + d2 ;
      }
   }

   // Implémentation de l'hôte.
   class Program {
      static void Main() {
         // 1)Création d'un canal http sur le port 65100
         //   enregistre ce canal dans le domaine d'application courant.
         HttpChannel canal = new HttpChannel(65100);
         ChannelServices.RegisterChannel(canal, false);

         // 2) Ce domaine d'application présente un objet de type
         //    IAdditionneur associé au point terminal 'ServiceAjout'.
         RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(CAdditionneur),
            "ServiceAjout",
            WellKnownObjectMode.SingleCall);

         // 3) Maintient du processus courant.
         Console.WriteLine(
            "Pressez une touche pour stopper le serveur.");
         Console.Read() ;
      }
   }
}
```

Notez l'utilisation du *point terminal* (*end point* en anglais) `"ServiceAjout"` associé à l'objet activé. En concaténant les informations de protocole, de machine, de port et de point terminal on obtient un URI qui localise complètement l'objet activé par le serveur. En l'occurrence cet URI est *http://localhost:65100/ServiceAjout*. Pour cette raison, on qualifie d'objet *bien connu* (*well-known object* en anglais) un objet activé par le serveur et qui a un point terminal. Dans la suite on utilisera l'acronyme *WKO (Well-Known Object)* pour parler d'un objet activé par le serveur. Lorsque l'on décrira plus en détail le mécanisme interne de .NET Remoting, on verra que le serveur peut publier un objet qu'il active sans utiliser de point terminal.

Il ne nous reste plus qu'à coder l'assemblage qui contient le code du client. Il est nécessaire de créer un canal, puis de récupérer un proxy transparent vers l'objet distant associé à l'URI

*http://localhost:65100/ServiceAjout*. À ce stade on peut utiliser le proxy transparent, matérialisé par une référence, exactement comme une référence vers un objet non distant.

*Exemple 22-6 :* `Client.cs`

```
using System ;
using System.Runtime.Remoting ;
using System.Runtime.Remoting.Channels ;
using System.Runtime.Remoting.Channels.Http ;

using NommageInterface ;

namespace NommageClient {
   class Program {
      static void Main() {
         // Crée un canal HTTP puis enregistre
         // ce canal dans le domaine d'application courant.
         // (la valeur 0 pour le numéro de port côté client signifie
         // que ce numéro de port est choisi automatiquement
         // par le CLR).
         HttpChannel canal = new HttpChannel(0);
         ChannelServices.RegisterChannel(canal, false);

         // Obtient un proxy transparent sur l'objet distant à partir de
         // son URI, puis transtype le proxy distant en IAdditionneur.
         MarshalByRefObject objRef = (MarshalByRefObject)
                           RemotingServices.Connect(
                                 typeof(IAdditionneur),
                                 "http://localhost:65100/ServiceAjout");
         IAdditionneur obj = objRef as IAdditionneur;

         // Appel d'une méthode sur l'objet distant.
         double d = obj.Add(3.0, 4.0);
         Console.WriteLine("Valeur retournée:" + d) ;
      }
   }
}
```

Nous disposons maintenant de trois fichiers sources C#, qui vont chacun permettre de produire un assemblage avec les trois instructions en ligne de commande suivantes :

```
>csc.exe /target:library Interface.cs
>csc.exe Serveur.cs /r:Interface.dll
>csc.exe Client.cs  /r:Interface.dll
```

Nous aurions pu aussi utiliser *Visual Studio* pour placer ces trois projets dans un même espace de travail.

Il vous suffit maintenant de lancer `Serveur.exe` puis `Client.exe`. Voici les affichages de ces programmes :

Affichage de Serveur.exe

```
Pressez une touche pour stopper le serveur.
CAdditionneur ctor
CAdditionneur Add( 3 + 4 )
```

Affichage de Client.exe

```
Valeur retournée:7
```

## *Service d'activation par le serveur : simple appel (single call) ou singleton*

Dans l'exemple de la section précédente, nous avons volontairement effectué un seul appel de méthode sur l'objet serveur. Examinons ce qu'aurait affiché le serveur si nous avions effectué consécutivement deux appels à `Add()`, du côté client :

Client.cs

```
...
   obj.Add( 3.0 , 4.0 ) ;
   obj.Add( 5.0 , 6.0 ) ;
...
```

Affichage du serveur :

```
Pressez une touche pour stopper le serveur.
CAdditionneur ctor
CAdditionneur Add( 3 + 4 )
CAdditionneur ctor
CAdditionneur Add( 5 + 6 )
```

D'après cet affichage, il semblerait que le serveur active un objet à chaque appel du client puisque le constructeur est appelé à chaque appel. C'est en fait, la description exacte de la réalité. Ce comportement est adopté par le serveur car nous avions déclaré l'objet WKO en mode *simple appel* (*single call* en anglais) lorsque nous avions écrit :

```
...WellKnownObjectMode.SingleCall...
```

Il existe un autre *mode d'appel* pour les objets activés par le serveur, appelé le mode *singleton*. Ce mode d'appel est choisi en remplaçant la valeur `SingleCall` par la valeur `Singleton`. Le mode d'appel singleton oblige le serveur à activer l'objet au premier appel d'un client, puis à maintenir cet objet pour tous les prochains appels de tous les clients. Une conséquence immédiate est que tous les clients partagent le même objet. Plusieurs threads du pool de thread peuvent exécuter les méthodes de cet objet simultanément. Il est donc nécessaire de prévoir un mécanisme de synchronisation pour protéger ces ressources des accès concurrents. Voici ce que le serveur aurait affiché si nous avions déclaré l'objet en mode d'appel singleton :

```
Pressez une touche pour stopper le serveur.
CAdditionneur ctor
CAdditionneur Add( 3 + 4 )
CAdditionneur Add( 5 + 6 )
```

Le serveur attend un premier appel d'un client avant d'activer l'objet dans les deux modes d'appel, singleton et simple appel.

Nous préférons parler de service d'activation d'un objet par le serveur que de parler d'activation d'objet par le serveur. En effet, le client n'a pas vraiment une référence vers un objet distant mais bien une référence vers un service d'activation d'objets distants. Concrètement si l'objet utilisé par un client est détruit, au prochain appel le client utilisera un autre objet, activé automatiquement par le serveur.

# *Activation par le client (CAO)*

*Les sources et les exécutables de cette section sont dans les répertoires :*

*Chap22 .NET Remoting/CAO*

*Chap22 .NET Remoting/CAO avec le mot-clé new*

Nous allons utiliser l'acronyme *CAO* pour parler d'un objet activé par un client (*Client Activated Object*). Pour activer un objet à partir du client, ce dernier doit connaître la classe de l'objet. Ce point constitue une différence fondamentale par rapport à l'architecture mise en œuvre lors du service d'activation d'objets par le serveur. En effet, dans le cas WKO, le client se contente de connaître les interfaces qu'il souhaite utiliser sur l'objet. Dans le cas d'un objet CAO, pour rendre accessible au client les métadonnées de type de la classe de l'objet, il faut lui fournir l'assemblage qui contient la classe de l'objet. Cette contrainte vous choque sûrement, aussi sachez que la section suivante va expliquer comment la contourner. En attendant, définissons le code C# d'un assemblage qui contient la classe `CAdditionneur` :

*Exemple 22-7 :* *ObjServeur.cs*

```
using System ;

namespace NommageObjServeur {
   public interface IAdditionneur {
      double Add(double d1, double d2) ;
   }
   // Implémentation de la classe des objets serveurs.
   public class CAdditionneur : MarshalByRefObject, IAdditionneur {
      public CAdditionneur() {
         Console.WriteLine("CAdditionneur ctor") ;
      }
      public double Add(double d1, double d2) {
         Console.WriteLine("CAdditionneur Add( {0} + {1} )", d1, d2) ;
         return d1 + d2 ;
      }
   }
}
```

Intéressons-nous maintenant au code source de l'assemblage contenant l'hôte. Cet hôte est très similaire à celui que nous avons utilisé pour activer un objet par le serveur. La seule différence vient du fait que l'on utilise la méthode statique `RegisterActivatedServiceType()` pour indiquer que la classe est activable à partir d'un autre domaine d'application au lieu d'utiliser

la méthode `RegisterWellKnownServiceType()` qui permet d'enregistrer le service d'activation d'un objet bien connu (WKO).

*Exemple 22-8 :* *Serveur.cs*

```
using System ;
using System.Runtime.Remoting ;
using System.Runtime.Remoting.Channels ;
using System.Runtime.Remoting.Channels.Http ;

using NommageObjServeur ;

namespace NommageServer {
   class Program {
      static void Main() {
         HttpChannel canal = new HttpChannel(65100) ;
         ChannelServices.RegisterChannel(canal, false) ;

         // Fait en sorte que ce domaine d'application présente
         // la classe CAdditionneur, qui peut ainsi être instanciée
         // par des clients distants.
         RemotingConfiguration.RegisterActivatedServiceType(
            typeof(CAdditionneur));

         Console.WriteLine(
            "Pressez une touche pour stopper le serveur.") ;
         Console.Read() ;
      }
   }
}
```

Le client est lui aussi, très similaire à un client qui utilise un objet activé par le serveur. La seule différence vient du fait que l'on utilise la méthode statique `Activator.CreateInstance()` pour récupérer un proxy transparent sur l'objet que l'on active.

*Exemple 22-9 :* *Client.cs*

```
using System ;
using System.Runtime.Remoting ;
using System.Runtime.Remoting.Channels ;
using System.Runtime.Remoting.Channels.Http ;
using System.Runtime.Remoting.Activation ;
using NommageObjServeur ;

namespace NommageClient {
   class Program {
      static void Main() {
         HttpChannel canal = new HttpChannel(0) ;
         ChannelServices.RegisterChannel(canal, false) ;
```

```
         // Active une instance du type distant et obtient un
         // proxy transparent sur cette nouvelle instance.
         IAdditionneur obj = Activator.CreateInstance(
             typeof(CAdditionneur),
             null,
             new Object[] { new UrlAttribute("http://localhost:65100")})
            as IAdditionneur;

         // Appel d'une méthode sur l'objet distant.
         double d = obj.Add(3.0, 4.0) ;
         Console.WriteLine("Valeur retournée:" + d) ;
      }
   }
}
```

Nous disposons maintenant de trois fichiers sources C#, qui vont chacun permettre de produire un assemblage, avec les trois instructions en ligne de commande suivante :

```
>csc.exe /target:library ObjServeur.cs
>csc.exe Serveur.cs /r:ObjServeur.dll
>csc.exe Client.cs  /r:ObjServeur.dll
```

Il vous suffit maintenant de lancer `Serveur.exe` puis `Client.exe`. Vous obtiendrez les mêmes affichages que ceux obtenus lors de l'activation du serveur. Cet exemple ne suffit donc pas à donner une différence perceptible par rapport au mode d'activation serveur. Cependant, la différence de comportement avec le cas où l'objet est activé par le serveur en mode singleton, est que chaque client disposera de son propre objet. La différence de comportement avec le cas où l'objet est activé par le serveur en mode simple appel, est que c'est le même objet qui est utilisé lors de plusieurs appels.

## *Activation de l'objet à partir du client avec le mot-clé new*

Le client précédent souffre de gros défauts : le code n'active pas l'objet distant avec la syntaxe C# utilisant le mot-clé `new`. Pire encore, le client ne peut pas choisir quel constructeur de la classe `CAdditionneur` doit utiliser le serveur. La méthode statique `RemotingConfiguration.RegisterActivatedClientType()` permet de remédier à ces deux problèmes. Elle permet de spécifier que chaque instanciation de la classe `CAdditionneur` dans le domaine d'application courant se fasse sur le serveur.

*Exemple 22-10 :* *`Client.cs`*

```
...
      static void Main() {
         HttpChannel canal = new HttpChannel(0) ;
         ChannelServices.RegisterChannel(canal, false) ;

         RemotingConfiguration.RegisterActivatedClientType(
            typeof(CAdditionneur),
            "http://localhost:65100");
         IAdditionneur obj = (IAdditionneur)new CAdditionneur();
```

```
         double d = obj.Add(3.0, 4.0) ;
         Console.WriteLine("Valeur retournée:" + d) ;
      }
...
```

## *Problème potentiel quant à la durée de vie*

Contrairement au comportement de la technologie COM/DCOM, le client n'est pas responsable de la durée de vie d'un objet qu'il active. Une conséquence fâcheuse est qu'un client peut chercher à utiliser un objet distant qu'il a activé, mais qui n'existe plus. Dans ce cas une exception de type `System.Runtime.Remoting.RemotingException` est retournée au client. Pour cette raison il faut toujours prévoir le traitement des exceptions du côté client.

La technique utilisée par le serveur pour gérer la durée de vie d'un objet est présentée un peu plus loin.

# *Le design pattern factory et l'outil soapsuds.exe*

Lors de la présentation de la technique d'activation d'un objet par le client (CAO), nous nous sommes aperçus qu'il fallait que l'assemblage du client référence l'assemblage contenant la classe de l'objet. On rappelle que cette contrainte a pu être évitée dans le cas WKO, puisque le client peut se contenter de ne connaître que des interfaces. Néanmoins si la classe de l'objet serveur ne supporte pas d'interfaces, on serait *a priori* obligé de référencer l'assemblage contenant cette classe dans l'assemblage client.

Il est souvent inacceptable de déployer l'assemblage contenant l'implémentation de l'objet serveur chez les clients. Cette implémentation doit absolument être diffusée le moins possible. Dans cet assemblage, le client n'a besoin que des métadonnées de la classe de l'objet serveur. Il existe deux solutions pour résoudre ce problème : le *design pattern* (Gof) *factory* ou l'outil `Soapsuds.exe`.

## *Le design pattern factory*

*Les sources et les exécutables de cette section sont dans le répertoire :*

*Chap22 .NET Remoting/CAO design pattern factory*

L'idée du *design pattern factory* (*usine* ou *fabrique* en français) est de faire construire l'objet CAO par un objet WKO. L'hôte n'a alors besoin d'exposer que le service WKO. Comme nous avons vu que dans ce cas le client peut se satisfaire d'une interface, le problème est résolu.

Modifions le code présenté pour activer l'objet à partir du serveur. Il faut tout d'abord présenter une nouvelle interface `IFabrique` au client :

*Exemple 22-11 :* *`Interface.cs`*

```
namespace NommageInterface {
   public interface IAdditionneur {
      double Add(double d1, double d2) ;
   }
   public interface IFabrique {
```

```
        IAdditionneur FabriqueNouvelAdditionneur();
    }
}
```

Ensuite il faut prévoir la classe `CFabrique` qui implémente `IFabrique`, et exposer un objet `CFabrique` en mode d'appel singleton (il vaut mieux ne pas utiliser le mode simple appel dans ce cas) :

*Exemple 22-12 :* *Serveur.cs*

```
...
   public class CAdditionneur : MarshalByRefObject, IAdditionneur {
      public CAdditionneur() {
         Console.WriteLine("CAdditionneur ctor") ;
      }
      public double Add(double d1, double d2) {
         Console.WriteLine("CAdditionneur Add( {0} + {1} )", d1, d2) ;
         return d1 + d2 ;
      }
   }
   public class CFabrique : MarshalByRefObject, IFabrique {
      public IAdditionneur FabriqueNouvelAdditionneur() {
         return (IAdditionneur)new CAdditionneur();
      }
   }
   class Program {
      static void Main() {
         HttpChannel canal = new HttpChannel(65100) ;
         ChannelServices.RegisterChannel(canal, false) ;
         RemotingConfiguration.RegisterWellKnownServiceType(
                typeof(CFabrique),
               "ServiceFabrique",
               WellKnownObjectMode.Singleton);
         Console.WriteLine(
            "Pressez une touche pour stopper le serveur.") ;
         Console.Read() ;
      }
   }
...
```

Enfin, on peut activer une instance de `CAdditionneur` à partir du client, sans connaître les classes `CAdditionneur` et `CFabrique`. C'est exactement le but de cette manipulation :

*Exemple 22-13 :* *Client.cs*

```
...
   class Program {
      static void Main() {
         HttpChannel canal = new HttpChannel(0) ;
         ChannelServices.RegisterChannel(canal, false) ;
```

```
         MarshalByRefObject tmpObj = (MarshalByRefObject)
                     RemotingServices.Connect(
                           typeof(IFabrique),
                           "http://localhost:65100/ServiceFabrique");
         IFabrique fabrique = tmpObj as IFabrique;
         IAdditionneur obj = fabrique.FabriqueNouvelAdditionneur();

         double d = obj.Add(3.0, 4.0) ;
         Console.WriteLine("Valeur retournée:" + d) ;
      }
   }
...
```

Ce *design pattern* marche car la classe `CAdditionneur` et la classe `CFabrique` héritent toutes les deux de la classe `MarshalByRefObject`.

## *L'outil Soapsuds.exe*

*Les sources et les exécutables de cette section sont dans le répertoire :*

*Chap22 .NET Remoting/CAO utilisation de soapsuds.exe*

L'outil `Soapsuds.exe`, fourni avec le *framework* .NET est capable d'extraire les métadonnées de la classe de l'objet serveur à partir de l'assemblage contenant cette classe. À partir de ces métadonnées, `Soapsuds.exe` peut fabriquer soit un fichier C# prêt à être compilé, soit un assemblage qui ne contient que ces métadonnées. Voici la ligne de commande à saisir pour fabriquer l'assemblage `ObjServeurPourClient.dll` à partir de l'assemblage `ObjServeur.dll`.

```
>soapsuds /ia:ObjServeur /oa:ObjServeurPourClient.dll
```

Voici la ligne de commande à saisir pour fabriquer le fichier source C# `ObjServeur.cs` à partir de l'assemblage `ObjServeur.dll`.

```
>soapsuds /ia:ObjServeur /gc
```

Voici un extrait du fichier source C# `ObjServeur.cs` :

*Exemple :* *`ObjServeur.cs`*

```
...
public class CAdditionneur :
System.Runtime.Remoting.Services.RemotingClientProxy, IAdditionneur {
   // Constructor
   public CAdditionneur() { }
   public Object RemotingReference { get { return (_tp) ; } }
   [SoapMethod(SoapAction =
            @"http://schemas.microsoft.com/clr/nsassem/
              NommageInterface.CAdditionneur/Interface#Add")]
   public virtual Double Add(Double d1, Double d2) {
      return ((CAdditionneur)_tp).Add(d1, d2) ;
   }
}
...
```

On a donc bien une nouvelle classe `CAdditionneur` présentant les mêmes méthodes publiques que l'originale et dérivant de la classe `System.Runtime.Remoting.Services.RemotingClientProxy`. Nous vous conseillons de consulter la documentation relative à cette classe dans les **MSDN**, car vous pouvez l'utiliser pour avoir un mécanisme d'authentification du client ou pour spécifier quel serveur proxy utiliser si le client est derrière un *pare feu* (*firewall* en anglais). Vous pouvez néanmoins indiquer à `Soapsuds.exe` que vous ne souhaitez pas que les classes dérivent de `RemotingClientProxy` en précisant l'option `/nowp`. Dans ce cas elles dériveront directement de `MarshalByRefObject`.

L'outil `Soapsuds.exe` permet aussi de fabriquer l'assemblage contenant les métadonnées des classes serveurs à partir d'un serveur distant qui expose une instance de cette classe, via un canal HTTP. Voici le code du serveur :

```
...
class Program {
   static void Main() {
      HttpChannel canal = new HttpChannel(65100) ;
      ChannelServices.RegisterChannel(canal, false) ;
      RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(CAdditionneur),
            "MonRep/ServiceAjout.soap",
            WellKnownObjectMode.SingleCall);
      ...
   }
}
...
```

Voici la ligne de commande à utiliser du côté client :

```
>Soapsuds /url:http://localhost:65100/MonRep/ServiceAjout.soap?wsdl
 /oa:ObjServeurPourClient.dll
```

Notez l'extension `soap` associée au point terminal et l'utilisation du suffixe `?wsdl` sur l'URL qui permet d'indiquer au serveur ce qu'on attend de lui, à savoir obtenir les métadonnées de type.

Précisons enfin que l'outil `soapsuds.exe` ne prend pas en compte les constructeurs avec arguments. L'utilisation du *design pattern factory* est donc plus adaptée aux classes qui doivent absolument être utilisées avec des constructeurs avec arguments.

# *Service de durée de vie*

La question de la durée de vie d'un objet WKO en mode simple appel ne se pose pas, puisque l'objet est détruit immédiatement après le premier appel. Nous allons exposer ici la solution mise en place par .NET Remoting pour déterminer quand un objet activé par une entité distante doit être détruit. Cette solution repose sur un mécanisme de *bail* (*lease* en anglais). Il est important de noter que cette solution concerne à la fois les objets CAO et les objets WKO en mode singleton. Il est aussi important de noter que tout ce que nous allons expliquer ici ne s'applique qu'aux objets dont la classe dérive de `MarshalByRefObject` et qui sont accédés par une entité située hors de leurs domaines d'application. Rappelons que les objets qui n'entrent pas dans cette catégorie voient leurs durées de vie gérées par le ramasse-miettes.

Dans un domaine d'application, il n'existe pas de références fortes vers un objet qui est accédé par une entité située hors de son domaine d'application. Pour empêcher le ramasse-miettes de collecter un tel objet, le CLR fait en sorte que chaque domaine d'application contienne un *administrateur de baux* (*lease manager* en anglais). Un administrateur de baux alloue une durée de bail, au moment de l'activation de chaque objet MBR utilisé d'une manière distante. L'administrateur de baux vérifie périodiquement le bail de chacun de ces objets. Ceux dont le bail n'est plus valide seront automatiquement détruits lors de la prochaine collecte du ramasse-miettes.

Cependant, .NET Remoting rend ce mécanisme de bail assez souple. Concrètement, la durée du bail d'un objet peut être prolongée de trois façons différentes :

- La durée du bail est automatiquement prolongée à chaque appel de l'objet.
- La durée du bail peut être prolongée directement par un appel à une certaine méthode.
- Enfin, lorsque l'administrateur de baux détecte un bail invalide, il consulte séquentiellement les sponsors de l'objet concerné avant de prendre la décision de le détruire. Les sponsors sont des objets qui ont la capacité de prolonger le bail d'un objet qui est sur le point d'être détruit. La classe d'un sponsor doit dériver de `MarshalByRefObject`. Les sponsors peuvent être des objets distants. Si la consultation d'un sponsor dépasse une certaine durée, il ne sera plus consulté et l'administrateur de baux passe à la consultation du prochain sponsor.

L'espace de noms `System.Runtime.Remoting.Lifetime` contient les deux interfaces `ILease` et `ISponsor`, spécialement conçues pour la gestion des mécanismes que nous venons de décrire :

```
interface System.Runtime.Remoting.Lifetime.ILease{
   // Etat du bail. L'énumération LeaseState a les valeurs :
   //    Null     Le bail n'est pas initialisé.
   //    Initial  Le bail est en cours d'initialisation.
   //    Active   Le bail est initialisé et valide.
   //    Renewing Le bail n'est plus valide et les sponsors sont en train
   //             d'être consultés, les uns après les autres.
   //    Expired  Le bail a expiré.
   LeaseState   CurrentState{get;}

   // Valeur initiale de la durée bail.
   // Cette valeur est modifiable seulement pendant
   // l'initialisation du bail.
   TimeSpan   InitialLeaseTime{get;set;}

   // Valeur du prolongement de la durée bail lors
   // de l'appel d'une méthode.
   TimeSpan   RenewOnCallTime{get;set;}

   // Valeur courante de la durée du bail.
   TimeSpan   CurrentLeaseTime{get;}

   // Prolonge la durée du bail.
   TimeSpan   Renew(TimeSpan) ;

   // Méthodes pour gérer l'ensemble des sponsors de l'objet.
```

```
   void      Register(ISponsor) ;
   void      Register(ISponsor,TimeSpan) ;
   void      UnRegister(ISponsor) ;

   // Durée maximale d'attente pour la consultation d'un sponsor.
   TimeSpan   SponsorshipTimeout{get;set;}
}

interface System.Runtime.Remoting.Lifetime.ISponsor{
   Timespan Renewal(ILease) ;
}
```

Comme vous le constatez, l'utilisation de ces interfaces est particulièrement intuitive. Vous pouvez obtenir le bail d'un objet en appelant la méthode `object MarshalByRefObject.GetLifetimeService()` sur l'objet, ou en appelant la méthode statique `object RemotingServices.GetLifetimeService(object)`. Par exemple :

```
...
ILease Bail = (ILease) obj.GetLifetimeService() ;
Bail.Renew(TimeSpan.FromSeconds(30)) ;
...
```

L'implémentation du bail est la classe interne `Lease` de `mscorlib.dll`. Vous n'avez pas accès à cette classe.

Les trois paramètres du bail d'un objet sont la durée initiale du bail, la durée de prolongement lors de l'appel d'une méthode et la durée maximale d'attente pour la consultation d'un sponsor. Pour un objet donné, ces trois paramètres prennent leurs valeurs par défaut dans le fichier de configuration de la machine. Par défaut ces durées sont respectivement de 5 minutes, 2 minutes et 2 minutes. Les propriétés statiques de la classe `System.Runtime.Remoting.Lifetime.LifetimeServices` permettent de configurer ces valeurs par défaut au niveau du domaine d'application courant. Pour affecter ces valeurs au niveau des objets d'une classe, il faut réécrire la méthode virtuelle `object MarshalByRefObject.GetLifetimeService()`. Par exemple :

*Exemple 22-14 :*

```
...
using System.Runtime.Remoting.Lifetime ;
...
   public class CAdditionneur : MarshalByRefObject, IAdditionneur {
      public override object InitializeLifetimeService() {
         ILease bail = (ILease)base.InitializeLifetimeService();
         if (bail.CurrentState == LeaseState.Initial) {
            bail.InitialLeaseTime = TimeSpan.FromSeconds(50);
            bail.RenewOnCallTime = TimeSpan.FromSeconds(20);
            bail.SponsorshipTimeout = TimeSpan.FromSeconds(20);
         }
         return bail ;
      }
   ...
}
...
```

Vous avez la possibilité de spécifier une durée de bail infinie en affectant la valeur `TimeSpan.Zero` à la propriété `InitialLeaseTime` d'un bail.

Vous pouvez définir votre propre classe de sponsors. Nous rappelons qu'une telle classe doit dériver de `MarshalByRefObject` et implémenter l'interface `ISponsor`. L'utilité d'une classe de sponsor apparaît durant la spécification de l'architecture distribuée, lorsque la vie d'un objet dépend logiquement d'une condition. La condition la plus courante est que certains clients soient toujours actifs. Il suffit alors d'avoir un sponsor pour chaque client, qui prolonge le bail de l'objet tant que le client associé au sponsor le désire. Méfiez-vous cependant de ce type de mécanisme, dit de *ping*. La technologie DCOM utilise ce type de mécanisme. L'expérience a prouvé que son coût n'est pas acceptable dès que le nombre de clients dépasse un certain seuil fonction de l'architecture.

# *Configurer la partie Remoting d'une application*

*Les sources et les exécutables de cette section sont dans le répertoire :*

*Chap22 .NET Remoting/Configurer la partie remoting d'une application*

Les exemples fournis pour exposer les mécanismes d'activation d'objets distants souffrent d'un problème majeur : ils ne sont pas configurables. C'est-à-dire qu'une fois compilés, on ne peut plus changer les numéros de port des canaux ou le nom de la machine du serveur. Nous allons montrer comment faire en sorte que les paramètres de configuration soient consultables et modifiables, aussi bien du côté de l'hôte que du côté du client. Comme vous pouvez vous en douter, ces paramètres vont être spécifiés dans un document XML.

Pour illustrer ces possibilités de configuration, nous allons nous servir de trois classes définies dans un nouvel assemblage :

*Exemple 22-15 :* *ObjServeur.cs*

```
using System ;

namespace NommageObjServeur {
   public interface IAdditionneur {
      double Add(double d1, double d2) ;
   }
   public class CAdditionneur : MarshalByRefObject, IAdditionneur {
      public CAdditionneur() {
         Console.WriteLine("CAdditionneur ctor") ;
      }
      public double Add(double d1, double d2) {
         Console.WriteLine("CAdditionneur Add( {0} + {1} )", d1, d2) ;
         return d1 + d2 ;
      }
   }
   public interface IMultiplicateur {
      double Mult(double d1, double d2) ;
   }
   public class CMultiplicateur : MarshalByRefObject, IMultiplicateur {
      public CMultiplicateur() {
```

```
         Console.WriteLine("CMultiplicateur ctor") ;
      }
      public double Mult(double d1, double d2) {
         Console.WriteLine("CMultiplicateur Mult( {0} + {1} )", d1, d2) ;
         return d1 * d2 ;
      }
   }
   public interface IDiviseur {
      double Div(double d1, double d2) ;
   }
   public class CDiviseur : MarshalByRefObject, IDiviseur {
      public CDiviseur() {
         Console.WriteLine("CDiviseur ctor") ;
      }
      public double Div(double d1, double d2) {
         Console.WriteLine("CDiviseur Div( {0} + {1} )", d1, d2) ;
         return d1 + d2 ;
      }
   }
}
```

## *Configuration d'un hôte*

Le document XML suivant permet d'exposer la classe `CAdditionneur` en mode d'appel WKO singleton et la classe `CMultiplicateur` en mode WKO simple appel. Parallèlement la classe `CDiviseur` est exposée de façon à pouvoir être activée par des clients distants. Ces trois expositions se font par le canal HTTP, accessible via le port 65100. L'attribut `ref` aurait pu être positionné à l'une des valeurs `tcp` ou `ipc` si nous souhaitions exploiter un canal de type TCP ou IPC. Dans le cas IPC, nous n'aurions pas eu besoin de fournir un numéro de port.

Chaque type exposé doit être préfixé par son espace de noms. De plus, ce nom de classe doit être suivi par le nom de l'assemblage dans lequel il est défini. Si cet assemblage a un nom fort, vous devez mettre le nom fort tout entier :

*Exemple 22-16 :* *Hote.config*

```
<configuration>
   <system.runtime.remoting>
      <application name = "Serveur">
         <service>
            <wellknown type="NommageObjServeur.CAdditionneur,ObjServeur"
                       mode ="Singleton" objectUri="Service1.rem" />
            <wellknown type="NommageObjServeur.CMultiplicateur,ObjServeur"

                       mode ="SingleCall" objectUri="Service2.rem" />
            <activated type="NommageObjServeur.CDiviseur,ObjServeur" />
         </service>
         <channels>
            <channel port="65100" ref="http" />
         </channels>
```

```
      </application>
   </system.runtime.remoting>
</configuration>
```

Grâce à la méthode statique `void RemotingConfiguration. Configure(string)` qui prend en paramètre le nom du document XML, le code de l'hôte est vraiment réduit au minimum :

*Exemple 22-17 :* *Serveur.cs*

```
...
   class Program {
      static void Main() {
         RemotingConfiguration.Configure("Hote.config", false) ;
         Console.WriteLine("Pressez une touche pour stopper le serveur.") ;
         Console.Read() ;
      }
   }
...
```

## *Configuration du client*

Le document XML suivant permet d'utiliser à partir d'un client les deux services WKO et la classe exposée par le serveur précédent :

*Exemple 22-18 :* *Client.config*

```
<configuration>
   <system.runtime.remoting>
      <application name = "Client">
         <client>
            <wellknown
             type="NommageObjServeur.CAdditionneur,ObjServeur"
             url="http://localhost:65100/Service1.rem" />
            <wellknown
             type="NommageObjServeur.CMultiplicateur,ObjServeur"
             url="http://localhost:65100/Service2.rem" />
         </client>
         <client url="http://localhost:65100/">
            <activated type ="NommageObjServeur.CDiviseur,ObjServeur"/>
         </client>
      </application>
   </system.runtime.remoting>
</configuration>
```

Les URLs spécifiées commencent par le mode d'accès `http://` mais elles auraient pu commencer par un des modes d'accès `tcp://` ou `ipc://` si notre canal récepteur était de type TCP ou IPC. Dans le cas IPC, il aurait fallu remplacer la paire nom de la machine hôte/port (en l'occurrence `localhost:65100`) par le nom du pipe nommé sous jacent. Ce nom est fourni par l'attribut `portName` dans la déclaration du canal dans le fichier de configuration du serveur. Pour charger les paramètres de configuration de ce fichier XML dans le domaine d'application du client, on utilise également la méthode statique `void RemotingConfiguration.Configure(string)` :

*Exemple 22-19 :*

```
...
using NommageObjServeur ;
...
   class Program {
      static void Main() {
         RemotingConfiguration.Configure("Client.config", false);

         CAdditionneur objA = new CAdditionneur() ;
         double dA = objA.Add(3.0, 4.0) ;

         CMultiplicateur objM = new CMultiplicateur() ;
         double dM = objM.Mult(3.0, 4.0) ;

         CDiviseur objD = new CDiviseur() ;
         double dD = objD.Div(3.0, 4.0) ;
      }
   }
...
```

Les appels aux constructeurs des objets WKO (`objA` et `objM`) ne sont là que pour récupérer une référence (en fait un proxy transparent) vers le type adéquat. Concrètement ils ne provoquent aucun accès au réseau.

À partir d'un fichier de configuration serveur, vous pouvez configurer les paramètres de l'administrateur de baux. Pour plus d'informations à ce sujet nous vous conseillons de consulter l'article **<lifetime> Element** des **MSDN**.

*A priori*, on ne peut pas utiliser d'interfaces puisqu'on est obligé d'appeler le constructeur. Le *design pattern factory* ne peut s'appliquer puisque même dans le cas de l'activation du serveur, le client ne se contente pas d'une interface. Il faut alors utiliser l'outil `Soapsuds.exe` pour éviter de fournir les assemblages contenant les implémentations des objets serveurs aux clients. Cependant nous allons montrer une « astuce » permettant l'utilisation d'interfaces en mode WKO, et donc, en mode CAO aussi en utilisant le *design pattern factory*.

## *Utilisation d'interfaces et de fichiers de configuration*

*Les sources et les exécutables de cette section sont dans le répertoire :*

*Chap22 .NET Remoting/Utilisation d'interfaces et de fichiers de configuration*

Tout d'abord, plaçons les interfaces dans un assemblage qui va être référencé à la fois par le client et par le serveur :

*Exemple 22-20 :* *Interface.cs*

```
namespace NommageInterface {
   public interface IAdditionneur {
      double Add(double d1, double d2) ;
   }
   public interface IMultiplicateur {
```

```
      double Mult(double d1, double d2) ;
   }
   public interface IDiviseur {
      double Div(double d1, double d2) ;
   }
   public interface IFabrique {
      IDiviseur FabriqueNouvelDiviseur() ;
   }
}
```

Ensuite, toute l'astuce repose sur l'utilisation d'une table, qui associe les services WKO configurés par le serveur à des interfaces configurées dans le client. Voici donc le code du client, avec la gestion de la table `dicoTypes` :

*Exemple 22-21 :* *Client.cs*

```
using System ;
using System.Runtime.Remoting ;
using System.Runtime.Remoting.Channels ;
using System.Runtime.Remoting.Channels.Http ;
using System.Runtime.Remoting.Activation ;
using System.Collections ;

using NommageInterface ;

namespace NommageClient {
   class NotreActivateur {
      private static bool bInit ;
      // Table d'association : interfaces/services distants WKO.
      private static IDictionary dicoTypes ;

      public static Object GetObject(Type type) {
         if (!bInit)
            InitdicoTypes() ;
         WellKnownClientTypeEntry entry = (WellKnownClientTypeEntry)
                                                   dicoTypes[type] ;
         return Activator.GetObject(entry.ObjectType, entry.ObjectUrl) ;
      }

      private static void InitdicoTypes() {
         bInit = true ;
         dicoTypes = new Hashtable() ;
         foreach ( WellKnownClientTypeEntry entry in
            RemotingConfiguration.GetRegisteredWellKnownClientTypes() )
               dicoTypes.Add(entry.ObjectType, entry) ;
      }
   }

   class Program {
      static void Main() {
```

```
            RemotingConfiguration.Configure("Client.config", false) ;

            IAdditionneur objA = (IAdditionneur)
               NotreActivateur.GetObject(typeof(IAdditionneur));
            double dA = objA.Add(3.0, 4.0) ;

            IMultiplicateur objM = (IMultiplicateur)
               NotreActivateur.GetObject(typeof(IMultiplicateur));
            double dM = objM.Mult(3.0, 4.0) ;

            // Utilisation du design pattern factory pour un objet CAO.
            IFabrique objFabrique = (IFabrique)
               NotreActivateur.GetObject(typeof(IFabrique));
            IDiviseur objD = objFabrique.FabriqueNouvelDiviseur();
            double dD = objD.Div(3.0, 4.0) ;
         }
      }
   }
```

Le fichier de configuration du client ressemble alors à :

*Exemple 22-22 :* *Client.config*

```
<configuration>
   <system.runtime.remoting>
      <application name = "Client">
         <client>
            <wellknown type="NommageInterface.IAdditionneur,Interface"
                       url="http://localhost:65100/Service1.rem" />
            <wellknown type="NommageInterface.IMultiplicateur,Interface"
                       url="http://localhost:65100/Service2.rem" />
            <wellknown type="NommageInterface.IFabrique,Interface"
                       url="http://localhost:65100/Service3.rem" />
         </client>
      </application>
   </system.runtime.remoting>
</configuration>
```

Comprenez bien que le contenu de la table d'associations interfaces/services WKO distants est écrit explicitement dans ce fichier de configuration. Ce contenu est récupéré dans le code du client par la méthode `RemotingConfiguration.GetRegisteredWellKnownClientTypes()`.

Le fichier de configuration du serveur présente trois services WKO. Nous rappelons que le troisième service sert à utiliser des objets CAO par l'intermédiaire du *design pattern factory* :

*Exemple 22-23 :* *Hote.config*

```
<configuration>
   <system.runtime.remoting>
      <application name = "Serveur">
         <service>
```

```
            <wellknown type="NommageServer.CAdditionneur,Serveur"
                       mode ="Singleton" objectUri="Service1.rem" />
            <wellknown type="NommageServer.CMultiplicateur,Serveur"
                       mode ="SingleCall" objectUri="Service2.rem" />
            <wellknown type="NommageServer.CFabrique,Serveur"
                       mode ="SingleCall" objectUri="Service3.rem" />
         </service>
         <channels>
            <channel port="65100" ref ="http" />
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

Bien évidemment, les classes `CAdditionneur`, `CMultiplicateur`, `CFabrique`, `CDiviseur` peuvent maintenant être placées dans le code du serveur :

*Exemple 22-24 :* *Serveur.cs*

```
using System ;
using System.Runtime.Remoting ;
using System.Runtime.Remoting.Channels ;
using System.Runtime.Remoting.Channels.Http ;

using NommageInterface ;

namespace NommageServer{
   public class CAdditionneur : MarshalByRefObject, IAdditionneur {
   ...
   public class CMultiplicateur : MarshalByRefObject, IMultiplicateur {
   ...
   public class CDiviseur : MarshalByRefObject, IDiviseur {
   ...
   public class CFabrique : MarshalByRefObject, IFabrique {
      public IDiviseur FabriqueNouvelDiviseur() {
         return new CDiviseur() ;
      }
   }
   class Program {
      static void Main() {
         HttpChannel canal = new HttpChannel(65100) ;
         ChannelServices.RegisterChannel(canal, false) ;
         RemotingConfiguration.RegisterWellKnownServiceType(
                typeof(CFabrique),
               "ServiceFabrique",
               WellKnownObjectMode.Singleton) ;
         Console.WriteLine("Pressez une touche pour stopper le serveur.") ;
         Console.Read() ;
      }
```

```
    }
}
```

# *Déploiement d'une application distribuée .NET Remoting*

Lors du déploiement d'une application distribuée basée sur .NET Remoting, il est fortement recommandé de fournir au moins un fichier de configuration pour chaque composant. Vous disposez ainsi d'un moyen standard pour configurer l'application sans avoir à recompiler aucun assemblage.

Les problèmes de versioning des classes des objets serveurs peuvent aussi être gérés à partir de ces fichiers de configuration. En effet, lorsque vous précisez un type dans un attribut XML `<type>`, vous devez préciser le nom de son assemblage. Or, ce nom peut contenir le numéro de version de l'assemblage.

Tous les hôtes de ce chapitre sont des exécutables en mode console. Dans une application réelle, on préfère utiliser soit un *service Windows* soit ASP.NET pour héberger nos objets serveur.

## *Services Windows*

L'avantage principal d'un service *Windows* par rapport à une application s'exécutant en mode console, réside dans le fait qu'il n'est pas nécessaire qu'un utilisateur soit logué sur la machine pour exécuter l'application hébergée par le service *Windows*. Un autre avantage est que la manipulation d'un service en exécution peut se faire directement à partir d'une interface graphique qui prévoit les commandes « démarrer / arrêter / suspendre / reprendre / redémarrer ». Un autre argument en faveur de cette pratique est le fait qu'un service peut être exécuté automatiquement après un *reboot* de la machine. Nous ne détaillons pas dans cet ouvrage la conception d'un service. Sachez qu'avec les classes contenues dans l'espace de noms `System.ServiceProcess`, cette tâche a été grandement simplifiée par rapport à l'utilisation des fonctions spécialisées win32. Tout ceci est décrit dans les **MSDN** à l'article **System.ServiceProcess Namespace**.

## *IIS*

L'utilisation de IIS constitue une autre alternative au mode console et à l'utilisation d'un service *Windows*, pour héberger vos objets serveurs. Les avantages principaux de ce choix sont les suivants :

- Vous avez accès à la gestion de la sécurité de IIS. Notamment, si votre serveur IIS supporte les certificats SSL, vous pouvez avoir accès au service d'encryption de vos données. Vous pouvez aussi avoir accès au mécanisme d'authentification de Windows.
- L'utilisation d'IIS vous décharge de la responsabilité d'écrire un hôte. Concrètement, vous n'avez qu'à développer vos classes dérivant de `MarshalByRefObject` et prévoir un fichier de configuration.
- Les problèmes de versioning sont aussi pris en charge par IIS. Celui-ci détecte quand vous installez une nouvelle version et s'occupe entièrement de la transition vers cette nouvelle version. Installer une nouvelle version signifie remplacer un assemblage par un autre, et remplacer le fichier de configuration par un autre, sans avoir à stopper IIS. IIS ne bloque

pas l'accès en écriture à ces fichiers lorsqu'il les utilise, car en fait, IIS travaille en interne avec des copies de ces fichiers. Ce mécanisme, très pratique, est nommé *shadow copy*.

Cependant toutes ces options ont un coût, et le principal problème de IIS est qu'il a un impact non négligeable sur les performances. Sachez aussi que vous ne pouvez pas utiliser de canaux TCP ou IPC avec IIS. Pour utiliser IIS deux étapes sont nécessaires :

- Sous `mmc` (la console générique d'administration de *Windows*), avec le snap-in IIS, créer un nouveau *répertoire virtuel IIS*. Nous précisons qu'un répertoire virtuel IIS permet de faire en sorte qu'un répertoire de la machine soit accessible à partir d'une URL. Dans ce répertoire, vous devez placer votre fichier de configuration. Ce fichier ne peut pas prendre n'importe quel nom. Son nom doit être `web.config`.
- Créer un sous répertoire `bin` dans le répertoire de la première étape. Placez les assemblages contenant les classes de vos objets serveur dans cet assemblage. Une autre option consiste à placer ces assemblages dans le répertoire GAC. Ils doivent alors avoir un nom fort.

Lorsque vous utilisez ce type de déploiement, l'élément `<application>` du fichier de configuration ne doit pas avoir d'attribut `Name`. De plus vous ne devez pas spécifier de port dans le fichier de configuration, pour ne pas interférer avec la gestion des numéros de port d'IIS.

# *Sécuriser une conversation .NET Remoting*

## *Sécurisation d'un canal TCP*

Si vous utilisez un canal de type TCP, vous avez la possibilité d'utiliser les protocoles NTLM et Kerberos pour authentifier l'utilisateur *Windows* sous lequel s'exécute le client et pour crypter les données échangées. Pour fournir cette possibilité, le canal TCP exploite en interne la classe `NegociatedStream` qui elle-même exploite les services de l'API win32 relative à ces protocoles connue sous le nom de SSPI. Ces protocoles ainsi que cette classe sont détaillés en page 660.

Pour avoir recours à cette possibilité il suffit de positionner l'attribut `secure` du canal TCP à `true` dans le fichier de configuration du client et du serveur :

```
<configuration>
   <system.runtime.remoting>
      <application name = "XXX">
         <channels>
            <channel port="65100" ref="tcp" secure="true"/>
         </channels>
...
```

Vous pouvez aussi positionner la valeur de l'attribut `secure` à `true` programmatiquement grâce à la liste des propriétés d'un canal :

*Exemple 22-25 :* *`Client.cs`*

```
...

      static void Main() {
         IDictionary properties = new Hashtable();
```

```
            properties.Add("secure", true);
            TcpChannel canal = new TcpChannel(properties,null,null) ;
            ChannelServices.RegisterChannel(canal, true) ;

...
```

Vous pouvez également positionner du coté serveur l'attribut `tokenImpersonationLevel` d'un canal TCP à la valeur `Impersonation` si vous souhaitez que la requête du cliente s'exécute du coté serveur avec le compte de l'utilisateur client (autrement dit la requête est impersonifiée).

## *Sécurisation d'un canal HTTP*

Si vous souhaitez sécuriser les données échangées lorsque vous avez recours au protocole HTTP, deux solutions sont possibles :

- Héberger le serveur avec IIS et utiliser le protocole SSL (donc HTTPS) au niveau d'IIS.
- Créer votre propre canal sécurisé HTTP.

# *Proxys et messages*

Le but de cette section est d'analyser de près les proxys transparents et de présenter les notions de proxy réel et d'intercepteur de messages. Résumons tout ce que nous avons déjà exposé à propos des proxys transparents :

- Un proxy transparent est une instance d'une classe interne à l'assemblage `mscorlib.dll`. Cette classe ne nous est donc pas accessible.
- Un proxy transparent est créé automatiquement par le CLR pour référencer un objet distant dont la classe dérive de `MarshalByRefObjectet`.
- Dans un assemblage, la gestion interne d'une référence vers un objet varie selon que l'assemblage est chargé dans le même domaine d'application que l'objet ou non. Dans le cas où l'assemblage et l'objet sont distants, on a une référence vers un proxy transparent. Dans le cas où ils sont dans le même domaine d'application, on a une référence directe vers l'objet. On a vu que la méthode statique `bool RemotingServices. IsTransparentProxy(object)` permet de savoir si une référence vers un objet est un proxy transparent ou est une référence directe.
- Les métadonnées du type par lequel on manipule l'objet distant doivent être chargées dans le domaine d'application, lorsque le CLR construit le proxy transparent vers cet objet distant. On a vu plusieurs façons d'obtenir ces métadonnées.

## *Transformer un appel de méthode en un message*

Comme le montre la Figure 22-4, un appel de méthode classique peut être vu comme une transformation de la pile du thread effectuant l'appel.

Cette transformation se fait en deux étapes : avant l'exécution de la méthode le CLR dépile les arguments pour en faire des variables locales à la méthode, après l'exécution le CLR empile la valeur de retour et les arguments de retour.

Lors de l'appel d'une méthode sur un objet distant, situé dans un autre domaine d'application, voire dans un autre processus, il est impossible d'utiliser cette technique de passage d'arguments

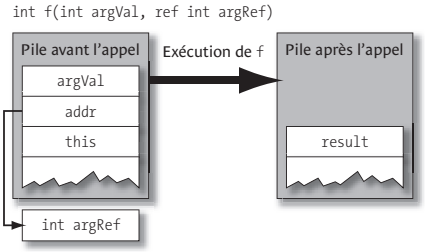


*Figure 22-4 : Appel de méthode = transformation de la pile*

par la pile. En effet, le thread qui appelle la méthode peut être différent du thread qui exécute la méthode. Les arguments sont passés par une technique d'échange de messages, un à l'appel de la méthode (contenant les informations consommées par la méthode) et un au retour (contenant les informations produites par la méthode).

Le rôle d'un proxy transparent est de gérer du côté client la transition entre le mode de passage d'arguments par la pile et le mode de passage d'arguments par messages. Un objet interne au CLR, appelé *constructeur de pile* (*stack builder sink* en anglais) réalise l'opération inverse du côté serveur. Tout ceci est illustré par la Figure 22-5.

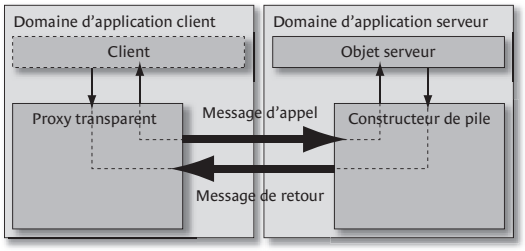


*Figure 22-5 : Proxy transparent et constructeur de pile*

## *Manipulation d'un message*

Nous allons voir dans la prochaine section que vous pouvez intercepter à différents niveaux ces messages échangés entre un proxy transparent et un constructeur de pile. On s'apercevra alors que ces messages sont en fait des objets .NET. Intéressons-nous ici aux interfaces qui permettent de manipuler ces messages. Voici leur hiérarchie :

```
System.Runtime.Remoting.Messaging.IMessage
   System.Runtime.Remoting.Messaging.IMethodMessage
      System.Runtime.Remoting.Messaging.IMethodCallMessage
         System.Runtime.Remoting.Activation.IConstructionCallMessage
      System.Runtime.Remoting.Messaging.IMethodReturnMessage
         System.Runtime.Remoting.Activation.IConstructionReturnMessage
```

Voici la description des interfaces `IMessage` et `IMethodMessage` :

```
using System.Runtime.Remoting.Messaging ;
using System.Collections ;
public interface IMessage {
   IDictionary Properties { get;}
}
public interface IMethodMessage : IMessage {
   object               GetArg(int index) ;
   string               GetArgName(int index) ;
   int                  ArgCount { get;}
   object[]             Args { get;}
   bool                 HasVarArgs { get;}
   string               MethodName { get;}
   object               MethodSignature { get;}
   string               TypeName { get;}
   string               Uri { get;}
   LogicalCallContext   LogicalCallContext { get;}
   MethodBase           MethodBase {get;}
}
```

L'interface `IMethodCallMessage` permet essentiellement de parcourir les données contenues dans un message fabriqué par un proxy transparent. L'interface `IMethodReturnMes\-sage` permet essentiellement de parcourir les données contenues dans un message fabriqué par un constructeur de pile.

## *Proxy transparent, proxy réel et la classe ObjRef*

Pour ne pas compliquer notre exposé, nous vous avons caché jusqu'ici l'existence des *proxys réels* (*real proxy* en anglais). Un proxy réel est un objet, instance de la classe `System.Runtime.Remoting.Proxies.RealProxy`, ou d'une classe dérivée de celle-ci. À chaque proxy transparent correspond un et un seul proxy réel. Le proxy transparent et le proxy réel se partagent le travail lors d'un appel sur l'objet distant. Comme nous l'avons vu, le proxy transparent gère le passage du mode « passage d'arguments par la pile » au mode « passage d'arguments par message ». La tâche du proxy réel est de trouver le canal adéquat pour envoyer le message fabriqué par le proxy transparent, et de l'envoyer. Une autre tâche du proxy réel est de récupérer du canal le message représentant le retour de l'appel de la méthode, et de transférer ce message de retour au proxy transparent. Contrairement aux proxys transparents, vous pouvez concevoir vos propres classes de proxys réels, comme on le verra un peu plus loin.

### *Type et proxy réel*

Une autre responsabilité d'un proxy réel est de connaître le type de l'objet distant. Le constructeur d'un proxy réel prend en argument un type. Ce type est obligatoirement soit une interface, soit une classe qui dérive de la classe `MarshalByRefObject` ; dans le cas contraire le constructeur du proxy réel lance une exception. La méthode `Type GetProxiedType()` est automatiquement appelée par le CLR sur un proxy réel, lorsque l'utilisateur transtype ce qu'il croit être une référence, mais est en fait un proxy transparent.

### *Création du couple proxy transparent/proxy réel et la classe ObjRef*

Dans un couple proxy transparent/proxy réel, c'est toujours le proxy réel qui est fabriqué en premier. Le proxy transparent est ensuite fabriqué par le proxy réel. Un proxy réel est fabriqué par

le CLR soit à partir d'une instance de la classe `System.Runtime.Remoting.ObjRef` soit à partir d'informations de localisation d'un objet distant. De telles informations peuvent contenir par exemple l'URI d'un service d'activation d'objets par le serveur. Les instances de la classe `ObjRef` sont MBV et peuvent donc faire traverser des informations à travers les frontières des domaines d'application. Une instance de `ObjRef` contient les informations nécessaires à un proxy réel, pour référencer un objet distant. Ces informations sont : le type de l'objet distant, les informations de localisation de l'objet distant, le type de canal à utiliser pour contacter l'objet distant et des informations utilisées en internes par le CLR. Une instance de la classe `ObjRef` référençant un objet distant peut être obtenue de différentes manières.

Dans le cas d'un objet activé par le client, une instance de la classe `ObjRef` est effectivement obtenue par le client lorsqu'il active l'objet. Ce comportement concerne à la fois l'utilisation du mot-clé `new`, l'utilisation de la méthode `AppDomain.CreateInstance()` ou l'utilisation de la méthode `Activator.CreateInstance()`. Nous vous avons parlé au début de ce chapitre de la classe `ObjectHandle` dont les instances permettent de « wrapper » un objet MBV ou une référence MBR. Nous pouvons maintenant préciser que si une instance de `ObjectHandle` contient une référence MBR, vous pouvez obtenir cette référence en appelant la méthode `ObjRef ObjectHandle.CreateObjRef(Type)`.

Dans le cas d'un objet WKO, le client n'a pas besoin d'une instance de la classe `ObjRef` initialisée par le serveur. Ceci est logique puisque seule la connaissance de l'URI du service d'activation d'objets par le serveur suffit au client pour avoir accès au service. Une conséquence est que, contrairement à ce que l'on pourrait penser, l'appel à une des méthodes `Activator.GetObjet()` ou `RemotingServices.Connect()` ne provoquent pas l'envoi d'un message sur le réseau. En plus d'optimiser l'utilisation du réseau, ce comportement permet au serveur de ne pas créer d'objets tant qu'il n'y a pas eu un appel de la part d'un client. Cette situation concerne à la fois les objets activés par le serveur en mode singleton et ceux activés par le serveur en mode simple appel. Comprenez bien que la mise en œuvre de ce mécanisme économise un aller/retour réseau, par rapport au mécanisme d'activation de l'objet par le client.

Enfin, sachez qu'une instance de la classe `ObjRef` est retournée par le domaine d'application serveur pour chaque valeur de retour de type référence et chaque argument de retour de type référence, lors de l'appel d'une méthode sur un objet distant. Bien évidemment, il faut que les types de ces arguments soient des interfaces ou des classes dérivants de `MarshalByRefObject`.

## *Une utilisation intéressante de la classe ObjRef : publier un objet*

*Les sources et les exécutables de cette section sont dans le répertoire :*

*Chap22 .NET Remoting/Publication d'un objet*

Une instance de la classe `ObjRef` contient toutes les informations nécessaires pour localiser et identifier un objet distant. D'après cette constatation, en sérialisant une telle instance dans un fichier, on dispose d'une référence vers un objet. L'idée est de rendre accessible ce fichier à un client, par exemple en lui envoyant par e-mail. En désérialisant l'instance de la classe `ObjRef`, le client dispose d'une référence vers l'objet distant et peut ainsi l'utiliser. Cette pratique s'appelle la *publication d'un objet*. Grâce à certaines méthodes de .NET Remoting, la publication d'un objet est une tâche particulièrement aisée à implémenter.

Voici le code du serveur qui sérialise le référence d'un objet dans le fichier *Additionneur.txt*. Notez qu'il faut que la classe de l'objet dérive de la classe `MarshalByRefObject` :

*Exemple 22-26 :* *Serveur.cs*

```
...
      static void Main() {
         HttpChannel canal = new HttpChannel(65100) ;
         ChannelServices.RegisterChannel(canal, false) ;

         CAdditionneur obj = new CAdditionneur() ;
         ObjRef objRef = RemotingServices.Marshal(obj) ;
         FileStream fStream = new
            FileStream("Additionneur.txt", FileMode.Create);
         SoapFormatter soapFormatter = new SoapFormatter();
         soapFormatter.Serialize(fStream, objRef);
         fStream.Close();

         Console.WriteLine("Pressez une touche pour stopper le serveur.") ;
         Console.Read() ;
      }
...
```

Voici le code du client qui désérialise le référence vers l'objet distant à partir du fichier *Additionneur.txt* :

*Exemple 22-27 :* *Client.cs*

```
using System.IO ;
using System.Runtime.Serialization.Formatters.Soap ;
...
      static void Main() {
         HttpChannel canal = new HttpChannel(0) ;
         ChannelServices.RegisterChannel(canal, false) ;

         FileStream fStream = new FileStream("Additionneur.txt",
                                        FileMode.Open);
         SoapFormatter soapFormatter = new SoapFormatter();
         IAdditionneur obj =
            soapFormatter.Deserialize(fStream) as IAdditionneur;

         double d = obj.Add(3.0, 4.0) ;
      }
...
```

Cette application nous permet de vérifier que .NET Remoting assigne une identité propre à chaque instance d'une classe dérivant de `MarshalByRefObject`. C'est-à-dire que chaque objet a un URI unique au monde, basé sur un *GUID* (*Global Unique Identity*). Un tel URI ressemble à ceci :

```
/6b03659f_0164_43e2_99cf_f36eda31adae/367459709_1.rem
```

Lorsqu'un objet publié est détruit, le fichier qui lui servait de référence n'est plus d'aucune utilité. Cet URI n'est communiqué au client que si l'objet est activé par le client ou publié par le serveur. Autrement dit, cet URI n'est pas communiqué au client dans le cas d'un objet WKO.

La technique de publication d'un objet est donc une solution intermédiaire entre la technique de service d'activation d'un objet distant par le serveur et la technique d'activation d'un objet distant par le client. L'objet est effectivement activé par le serveur mais le client tient compte de l'identité de l'objet. Nous récapitulons les différences de ces quatre modes d'activation d'un objet à la fin de ce chapitre.

Le fichier de publication d'un objet est constitué d'une trentaine de lignes au format XML, fastidieuses à lire. En voici quelques extraits pertinents :

*Exemple :* *Additonneur.txt*

```
...
<uri id="ref-2">/6b03659f_0164_43e2_99cf_f36eda31adae/367459709_1.rem</uri>
...
<serverType id="ref-5">NommageServer.CAdditionneur, Server,
 Version=1.0.1140.28752, Culture=neutral, PublicKeyToken=null</serverType>
...
<item id="ref-8">NommageInterface.IAdditionneur, Interface,
 Version=1.0.1138.27103, Culture=neutral, PublicKeyToken=null</item>
...
<a3:CrossAppDomainData id="ref-9" xmlns:a3=
 "http://schemas.microsoft.com/clr/ns/System.Runtime.Remoting.Channels">
<_ContextID>1362648</_ContextID>
<_DomainID>1</_DomainID>
<_processGuid id="ref-11">faf88595_9b2e_4f23_99ee_1d0046915a98</_processGuid>
</a3:CrossAppDomainData>
...
<item id="ref-13">http://213.36.58.1:65100</item>
...
```

## *Intercepteur de messages (message sink)*

Pour résumer, en .NET Remoting un appel de méthode est en fait un échange de deux messages, un qui contient les informations que va consommer l'exécution de la méthode et un qui contient les informations produites par l'exécution de la méthode. Chacun de ces deux messages subit plusieurs traitements entre le proxy réel, situé dans le domaine d'application client, et le constructeur de pile, situé dans le domaine d'application serveur. Par exemple, le message contenant les arguments entrants de la méthode est sérialisé du côté client, puis désérialisé du côté serveur. Chacun de ces traitements est effectué par un objet appelé *intercepteur de messages* (*message sink* en anglais). Concrètement, un intercepteur de messages est une instance d'une classe qui implémente l'interface `System.Runtime.Remoting.Messaging.IMessageSink` et qui est marquée avec l'attribut .NET `Serializable` :

```
public interface System.Runtime.Remoting.Messaging.IMessageSink{
   IMessageSink   NextSink{get;}
   IMessage       SyncProcessMessage(  IMessage request ) ;
   IMessageCtrl   AsyncProcessMessage( IMessage request ,
```

```
                                        IMessageSink  replySink) ;
}
```

Grâce à la propriété `NextSink`, les intercepteurs de messages peuvent être chaînés. Pour cette raison, on les appelle parfois des *chaîneurs de messages*. Comme vous vous en doutez, le traitement d'un message représentant un appel synchrone se fait dans la méthode `SyncProcessMessage()` et le traitement d'un message représentant un appel asynchrone se fait dans la méthode `AsyncProcessMessage()`. Voici une implémentation où nous indiquons où vous pouvez placer le code de vos traitements :

Il faut toujours marquer une classe d'intercepteurs de messages avec l'attribut .NET `Serializable`.

```
[Serializable]
public class MonMsgSink : IMessageSink{
   private IMessageSink m_NextSink ;
   public  IMessageSink NextSink{ get { return m_NextSink;} }
   public  MonMsgSink(IMessageSink NextSink){m_NextSink = NextSink;}

   IMessage SyncProcessMessage(IMessage MsgIn){
      // Ici vous pouvez faire un traitement sur MsgIn.
      IMessage MsgOut = m_NextSink.SyncProcessMessage(MsgIn) ;
      // Ici vous pouvez faire un traitement sur MsgOut.
      return MsgOut ;
   }
   IMessageCtrl AsyncProcessMessage(   IMessage MsgIn,
                  IMessageSink  replySink){
      // Ici vous pouvez faire un traitement sur MsgIn.
      // Vous pouvez aussi ajouter un intercepteur de msgs pour la
      // chaine d'intercepteurs de msgs qui sera utilisée pour
      // traiter le msg représentant le retour de l'appel.
      IMessageCtrl MsgCtrl =
          m_NextSink.AsyncProcessMessage(MsgIn,replySink) ;
      // Ici vous pouvez indiquer au CLR de ne plus attendre de retour
      // pour cet appel asynchrone aprés une durée de
      // 1000 millisecondes en écrivant : 'MsgCtrl.Cancel(1000);'
      return MsgCtrl ;
   }
}
```

Remarquez que la méthode `SyncProcessMessage()` peut traiter le message entrant et le message sortant alors que la méthode `AsyncProcessMessage()` ne traite que le message entrant. En revanche, la méthode `AsyncProcessMessage()` a la possibilité de participer à la construction de la chaîne d'intercepteurs de messages qui sera utilisée pour traiter le message contenant les informations retournées par l'appel de la méthode. Naturellement, les traitements s'enchaîneront dans l'ordre inverse dans lesquels ils ont été chaînés.

Vous savez maintenant comment concevoir des intercepteurs de messages. Nous allons vous expliquer, dans la suite de ce chapitre, comment injecter vos propres intercepteurs de message

dans la chaîne d'un appel de méthode, et surtout, quels types de bénéfices vous pouvez tirer de cette pratique. Auparavant, nous présentons une implémentation d'un proxy réel propriétaire. Vous allez vous apercevoir que nous allons pour cela avoir accès au premier intercepteur de messages de la chaîne d'un appel synchrone de méthode.

## *Pourquoi utiliser un proxy réel propriétaire ?*

Avant de montrer des exemples de création de classes de proxys réels propriétaires, il est utile de donner quelques exemples d'utilisation d'un proxy réel propriétaire.

Vous pouvez vous servir d'un proxy réel propriétaire simplement pour tracer les appels à un objet distant.

Vous pouvez vous servir d'un proxy réel propriétaire pour empêcher certains appels distants, qui peuvent être réalisés localement. Vous pouvez ainsi réaliser un système de cache d'informations. Lorsqu'une information est demandée à l'objet distant, le proxy réel propriétaire va d'abord vérifier si l'information n'est pas disponible localement.

Vous pouvez utiliser un proxy réel propriétaire pour modifier les arguments d'un appel de méthode d'une manière transparente. Par exemple vous pourriez traduire les chaînes de caractères, passées en argument d'une méthode, d'une langue source vers une langue destination.

Vous pouvez vous servir d'un proxy réel propriétaire pour permettre le transtypage d'un proxy transparent en un type qui n'est pas forcément le type détenu par le proxy réel. Nous ne décrivons pas en détail cette possibilité, mais sachez que pour la réaliser, votre classe de proxy réel propriétaire doit implémenter l'interface `System.Runtime.Remoting. IRemotingTypeInfo`. La description à l'article **IRemotingTypeInfo members** des **MSDN** des deux membres de cette interface, vous permettra de comprendre comment l'utiliser.

Vous pouvez vous servir d'un proxy réel propriétaire pour faire de la *répartition de charge* entre les serveurs (*load balancing* en anglais). Il vous suffit d'obtenir régulièrement la charge de chaque serveur et de router les appels à un objet distant, vers le serveur le moins surchargé. Vous pouvez aussi tenir compte de la puissance de chaque serveur lors de l'évaluation de sa charge. Nous vous conseillons aussi d'essayer de router les appels aléatoirement vers les serveurs, avec des densités de probabilité proportionnelles aux charges que chaque serveur peut supporter. Cette technique donne souvent des résultats équivalents à l'application d'un algorithme basé sur la charge de chaque serveur. Nous précisons que la répartition de charge entre plusieurs serveurs ne peut se faire que lorsque vous travaillez avec des objets distants WKO sans état.

## *Implémenter et utiliser une classe de proxys réels propriétaires*

Il est très facile d'implémenter une classe de proxys réels propriétaires. Il vous suffit de faire dériver une classe de la classe `RealProxy`, et d'implémenter la méthode `IMessage Invoke(IMessage)`. Cette méthode est automatiquement appelée par le CLR lors d'un appel synchrone sur le proxy transparent associé. Le message en entrée représente le message fabriqué par le proxy transparent. Le message en sortie est le message représentant le retour de l'appel, qui sera automatiquement passé au proxy transparent. Pour transmettre un appel synchrone au sein de la méthode `Invoke()` il suffit d'appeler la méthode `IMessage.SyncProcessMessage(IMessage)` sur le premier intercepteur de messages de la chaîne. Nous allons voir comment récupérer une référence vers cet intercepteur de message.

Voici un exemple de classe de proxy réel propriétaire. Dans cet exemple, l'intercepteur de messages que la méthode `Invoke()` utilise est fourni par le canal. Cependant, on verra qu'il peut y

avoir d'autres intercepteurs de messages intercalés entre le proxy réel et le canal. Dans la méthode `Invoke()` on ne fait qu'afficher les arguments en entrée de la méthode, puis la valeur de retour. Notez que l'on place l'URI du service d'activation d'objets par le serveur, dans le message d'entrée. Cette étape pourrait être très facilement modifiée pour faire de la répartition de charge entre plusieurs serveurs.

*Exemple 22-28 :* *Client.cs*

```
using System ;
using System.Runtime.Remoting ;
using System.Runtime.Remoting.Channels ;
using System.Runtime.Remoting.Channels.Http ;
using System.Runtime.Remoting.Proxies ;
using System.Runtime.Remoting.Messaging ;
using System.Collections ;
using NommageInterface ;
// Il faut utiliser soapsuds.exe
// pour que le client connaisse la classe CAdditionneur.
using NommageServer;

public class MonProxyReel : RealProxy {
   // URI du service d'activation d'objet par le serveur.
   String m_Uri ;
   // Correspond au premier intercepteur de msg du canal HttpSender.
   IMessageSink m_MsgSink ;

   public MonProxyReel(Type type, String uri,
                       IChannelSender CanalEnvoi) : base(type) {
      m_Uri = uri ;
      string unused ;
      // Obtient du canal, une chaine d'intercepteurs de messages.
      m_MsgSink = CanalEnvoi.CreateMessageSink(m_Uri, null, out unused);
   }

   // Méthode appelée par le CLR avant chaque appel du client.
   public override IMessage Invoke(IMessage msgIn) {
      // Place l'URI de l'objet distant dans MsgIn.
      IDictionary d = msgIn.Properties;
      d["__Uri"] = m_Uri;

      // Affiche les arguments en entrée contenus dans MsgIn.
      IMethodCallMessage msgAppel = (IMethodCallMessage)msgIn ;
      Console.Write("MonProxyReel : Avant l'appel de:{0}(",
                    msgAppel.MethodName) ;
      for (int i = 0 ; i < msgAppel.InArgCount ; i++)
         Console.Write(" {0}={1} ",
                       msgAppel.GetArgName(i), msgAppel.GetArg(i)) ;
      Console.WriteLine(")") ;

      // Réalise l'appel distant.
```

```
      IMethodReturnMessage msgOut =
         (IMethodReturnMessage)m_MsgSink.SyncProcessMessage(msgIn);

      // Affiche la valeur de sortie contenue dans msgOut.
      Console.WriteLine("MonProxyReel : Après appel de:{0}() RetVal={1}",
               MsgAppel.MethodName, msgOut.ReturnValue) ;
      return msgOut;
   }
}

namespace NommageClient {
   class Program {
      static void Main() {
         HttpChannel canal = new HttpChannel(0) ;
         ChannelServices.RegisterChannel(canal, false) ;

         // Fabrication de notre propre proxy réel.
         // On lui fournit le type de l'objet, l'URI du service WKO
         // et le canal d'envoi que l'on souhaite utiliser.
         MonProxyReel proxy = new MonProxyReel(
            typeof(CAdditionneur),
            "http://localhost:65100/ServiceAjout",
            (IChannelSender)canal);
         IAdditionneur obj = (IAdditionneur)
                  proxy.GetTransparentProxy();

         // À ce stade, il n'y a toujours pas eu de contact réseau.
         double d = obj.Add(3.0, 4.0) ;
      }
   }
}
```

Voici l'affichage du client, lorsqu'on l'utilise avec un serveur adéquat :

```
MonProxyReel : Avant l'appel de:Add( d1=3  d2=4 )
MonProxyReel : Après appel de:Add() RetVal=7
```

## *Affectation automatique d'un proxy réel propriétaire à toutes les instances d'une classe*

Dans le programme précédent, nous avons explicitement construit le proxy réel et nous avons explicitement demandé le proxy transparent. Vous pouvez faire en sorte que toutes les instances de la classe `CAdditionneur` aient chacune un proxy réel propriétaire. Dans ce cas, une instance aura son proxy réel propriétaire même si elle est construite avec l'opérateur `new` et même si elle est utilisée d'une manière non distante. En fait, la technique que nous vous proposons ici est surtout utilisée pour pouvoir obtenir les bénéfices d'un proxy réel propriétaire sur des objets utilisés d'une manière non distante.

Pour cela, il faut d'abord que la classe de nos instances, la classe `CAdditionneur` en l'occurrence, dérive de la classe `System.ContextBoundObject`. Nous donnons la signification précise de la classe `ContextBoundObject` un peu plus loin dans ce chapitre. Ensuite il faut définir une classe d'attribut .NET qui dérive de la classe `System.Runtime.Remoting.Proxies.ProxyAttribute`. Dans cette classe, vous réécrivez la méthode virtuelle `MarshalByRefObject ProxyAttribute.CreateInstance(Type t)` de façon qu'elle intercale un proxy réel entre l'objet créé et sa référence. Il faut marquer la classe `CAdditionneur` avec cet attribut .NET. Voici un programme illustrant tout ceci :

*Exemple 22-29 :*

```
using System ;
using System.Runtime.Remoting ;
using System.Runtime.Remoting.Services ;
using System.Runtime.Remoting.Messaging ;
using System.Runtime.Remoting.Proxies ;
using System.Runtime.Remoting.Activation ;

public class MonProxyReel : RealProxy {
   readonly bool m_bAffiche ;
   readonly MarshalByRefObject m_ObjCible ;

   public MonProxyReel(
      MarshalByRefObject objCible, Type type, bool bAffiche):base(type){
      m_bAffiche = bAffiche ;
      m_ObjCible = objCible ;
   }

   public override IMessage Invoke(IMessage msgIn) {
      IMessage msgOut ;
      if (msgIn is IConstructionCallMessage) {
         IConstructionCallMessage appelCtor =
                              (IConstructionCallMessage)msgIn;

         // Obtient le proxy par défaut.
         RealProxy proxyParDefaut =
                           RemotingServices.GetRealProxy(m_ObjCible);

         // Invoque le constructeur sur ce proxy réel.
         proxyParDefaut.InitializeServerObject(appelCtor);

         // Retourne notre Proxy transparent sur le nouvel objet.
        msgOut=EnterpriseServicesHelper.CreateConstructionReturnMessage(
                  appelCtor, (MarshalByRefObject)GetTransparentProxy());

         if (m_bAffiche)
            Console.WriteLine("MonProxyReel : appel du constructeur") ;
      }
      else {
         IMethodCallMessage appel = (IMethodCallMessage)msgIn ;
```

```
         if (m_bAffiche)
            Console.WriteLine("MonProxyReel : Avant l'appel de:{0}",
               appel.MethodName) ;

         msgOut = RemotingServices.ExecuteMessage(m_ObjCible, appel) ;

         if (m_bAffiche)
            Console.WriteLine("MonProxyReel : Aprés l'appel de:{0}",
                              appel.MethodName) ;
      }
      return msgOut ;
   }
}

[AttributeUsage(AttributeTargets.Class)]
public class MonProxyAttribute : ProxyAttribute {
   bool m_bAffiche ;
   public MonProxyAttribute(bool bAffiche) {
      m_bAffiche = bAffiche ;
   }
   public override MarshalByRefObject CreateInstance(Type T) {
      // Création d'une nouvelle instance.
      MarshalByRefObject objCible = base.CreateInstance(T);
      // Intercale un proxy réel entre la nouvelle instance
      // et le client.
      RealProxy realProxy = new MonProxyReel(objCible, T, m_bAffiche);

      return (MarshalByRefObject)realProxy.GetTransparentProxy();
   }
}

// Le paramètre true indique que l'on souhaite que le proxy réel
// signale sa présence en affichant des messages sur la console.
[MonProxyAttribute(true)]
public class CAdditionneur : ContextBoundObject {
   public int Add(int a, int b) { return a + b ; }
}

public class Program {
   static void Main() {
      CAdditionneur obj = new CAdditionneur() ;
      obj.Add(5, 6) ;
   }
}
```

Ce programme affiche ceci sur la console :

```
MonProxyReel : Appel du constructeur
MonProxyReel : Avant l'appel de:Add
MonProxyReel : Aprés l'appel de:Add
```

## *Consultation et modification des arguments d'un appel*

Nous avons vu comment lire les arguments d'un appel, à partir d'un message représentant cet appel. On peut aussi utiliser un proxy réel propriétaire ou un intercepteur de messages propriétaire pour modifier les arguments d'un appel. Par exemple vous pourriez utiliser cette possibilité pour traduire les chaînes de caractères passées en argument d'u ne langue source vers une langue destination. L'accès en écriture aux arguments contenus dans un message est possible mais elle n'est pas aussi directe que l'accès en lecture, qui rappelons-le, peut être effectué ainsi :

```
...
public override IMessage Invoke(IMessage msgIn){
   IMethodCallMessage msgAppel = (IMethodCallMessage)msgIn ;
   Console.Write("MonProxyReel : Avant l'appel de:{0}(",
                 msgAppel.MethodName) ;
   for(int i=0 ; i< msgAppel.InArgCount ; i++)
      Console.Write(" {0}={1} ", msgAppel.GetArgName(i),
                                 msgAppel.GetArg(i) ) ;
   Console.WriteLine(")") ;
...
```

En effet, quelle que soit l'interface que l'on utilise pour manipuler un message, on ne peut avoir accès en écriture aux arguments contenus dans le message. Autrement dit, chacune de ces interfaces supporte un accesseur `get` sur le tableau d'argument mais pas d'accesseur `set`. Pour arriver à nos fins, nous vous proposons d'utiliser la classe `System.Runtime.Remoting.Messaging.MethodCallMessageWrapper` qui a un accesseur `set` sur le tableau des arguments. Voici un extrait de code illustrant cette manipulation :

```
...
public override IMessage Invoke(IMessage msgIn){
   IMethodCallMessage msgAppel = (IMethodCallMessage)msgIn ;
   MethodCallMessageWrapper newAppel = new
                                   MethodCallMessageWrapper(appel);
   object[] tmpArgs = newAppel.Args;
   tmpArgs[0] = 1; // Affecte une valeur au premier argument.
   tmpArgs[1] = 2; // Affecte une valeur au second argument.
   newAppel.Args = methodArgs;
   appel = newAppel;

   msgOut = RemotingServices.ExecuteMessage(m_ObjCible,appel) ;
...
```

# Canaux (channels)

## Introduction

Les *canaux* sont les entités qui transmettent les messages représentant les appels de méthodes inter domaines d'application. De ce fait, il existe au moins un canal dans le domaine d'application du client et un canal dans le domaine d'application du serveur. Cependant, un domaine d'application peut contenir plusieurs canaux, et une implémentation d'un canal peut être utilisée par un client ou par un serveur.

Un canal est une instance d'une classe implémentant l'interface `System.Runtime.Remoting.Channels.IChannel`. Un canal qui peut être utilisé par un client est appelé *canal émetteur*. Par définition, un canal émetteur implémente l'interface `System.Runtime.Remoting.Channels.IChannelSender`. Un canal qui peut être utilisé par un serveur est appelé *canal récepteur*. Par définition, un canal récepteur implémente l'interface `System.Runtime.Remoting.Channels.IChannelReceiver`.

Le *framework* .NET expose trois implémentations pour les canaux : la classe `System.Runtime.Remoting.Channels.Http.HttpChannel`, la classe `System.Runtime.Remoting.Channels.Tcp.TcpChannel` et la classe `System.Runtime.Remoting.Channels.Ipc.Ipc\-Chan\-nel`. Chacune de ces implémentations peut à la fois servir de canal émetteur et de canal récepteur.

Les protocoles HTTP et TCP sont bien évidemment supportés respectivement par les classes `HttpChannel` et `TcpChannel`. La classe `IpcChannel` supporte quant à elle la notion de *pipe nommé*. Un pipe nommé est un objet *Windows* permettant à deux processus *Windows* hébergés sur la même machine de communiquer. Vous pouvez aussi faire communiquer deux processus *Windows* hébergés sur la même machine avec un protocole réseau tel que HTTP ou TCP. L'avantage des pipes nommés réside dans le fait qu'ils sont implémentés au niveau du système d'exploitation et qu'ils n'utilisent donc pas d'API réseau. En conséquence ils sont plus performants lorsqu'il s'agit de faire communiquer deux processus hébergés sur la même machine. L'acronyme *IPC* signifie *Inter Processus Communication*. La documentation anglo saxonne parle aussi parfois de *same-box communication*.

Pour enregistrer un canal dans un domaine d'application, vous pouvez soit utiliser la méthode statique `ChannelServices.RegisterChannel()` soit charger un fichier de configuration .NET Remoting avec la méthode statique `RemotingConfiguration.Configure()`. Ces deux techniques ont déjà été présentées dans les pages précédentes.

Chaque canal a un nom. Deux canaux dans le même domaine d'application ne peuvent pas avoir le même nom. Par défaut le nom d'un canal instance de `HttpChannel` est `"http"`, le nom d'un canal instance de `TcpChannel` est `"tcp"` et le nom d'un canal instance de `IpcChannel` est `"ipc"`. Voici comment affecter un nom particulier à un canal :

```
...
static void Main() {
   IDictionary prop = new Hashtable() ;
   prop["name"] = "tcp2";
   prop["port"] = "65101" ;
   ChannelServices.RegisterChannel(new TcpChannel(prop, null, null)) ;
...
```

Chaque canal de type HTTP ou TCP a besoin d'un numéro de port de la machine. Un numéro de port ne peut être utilisé par plusieurs canaux existant sur une même machine. Pour éviter

toute collision dans le choix d'un numéro de port, vous pouvez affecter le numéro de port 0 durant la création d'un tel canal. Le CLR fera en sorte de trouver un numéro de port libre et d'affecter ce numéro au nouveau canal. Comme on peut s'en douter, les implémentations *Microsoft* des canaux HTTP et TCP se basent sur une socket en interne.

Lorsque deux domaines d'applications se trouvent dans le même processus, il serait dommage d'utiliser ces mécanismes lourds prévus pour la communication interprocessus. Pour cette raison, l'assemblage `mscorlib.dll` contient la classe interne `CrossAppDomain` qui est prévue spécialement pour le cas où le domaine d'application contenant le client et le domaine d'application contenant les objets serveurs sont dans le même processus. Cette classe est utilisée automatiquement par le CLR. Vous n'avez donc rien à faire de particulier pour bénéficier de cette optimisation.

## *Relations entre les canaux émetteurs et les proxys d'un domaine d'application client*

La relation entre les canaux émetteurs et les proxys situés dans un même domaine d'application, est souvent mal comprise. Cela mène souvent à des aberrations, comme la création d'un canal émetteur pour chaque objet distant !

Un canal émetteur dans un domaine d'application est une fabrique à chaînes d'intercepteurs de messages pour les proxys. Revenons un instant au programme illustrant la conception d'un proxy réel propriétaire (l'Exemple 22-28). Le constructeur de la classe de proxy réel `MonProxyReel` avait besoin de connaître le canal émetteur à utiliser pour contacter l'objet distant :

```
...
   public MonProxyReel(Type type, String Uri, IChannelSender CanalEnvoi):
             base(type){
      m_Uri = Uri ;
      string Unused ;
      // Obtient une chaine d'intercepteurs de messages du canal.
      m_MsgSink = CanalEnvoi.CreateMessageSink(m_Uri, null,out Unused);
   }
...
```

La méthode `IChannelSender.CreateMessageSink()` demande au canal émetteur sous-jacent de fabriquer une chaîne d'intercepteurs de messages. Cette chaîne est paramétrée notamment par l'URI du service d'activation d'objet distant et par le port du canal.

## *Relations entre les canaux récepteurs et les objets d'un domaine d'application serveur*

La relation entre les canaux récepteurs et les objets d'un serveur, situés dans un même domaine d'application, est également mal comprise. Cela mène souvent à des aberrations comme la création d'un canal récepteur pour chaque objet du serveur.

Cette relation est mal comprise en partie parce que ni la méthode `RemotingConfiguration.RegisterWellKnownServiceType()` qui sert à exposer un service d'activation d'objet WKO, ni la méthode `RemotingConfiguration.RegisterActivatedServiceType()` qui sert à exposer

une classe à partir du serveur, ne prend de canal récepteur en argument. La raison est simple : chaque canal récepteur d'un domaine d'application peut recevoir des appels pour chaque objet de ce domaine utilisable d'une manière distante.

L'appel à la méthode `RemotingConfiguration.RegisterWellKnownServiceType()` crée une association interne et globale au domaine d'application entre l'URI du service WKO et le service WKO lui-même. Ce service est paramétré par le mode d'appel et une classe. L'appel à la méthode `RemotingConfiguration.RegisterActivatedServiceType()` enregistre dans le domaine d'application le fait qu'une certaine classe dérivant de la classe `MarshalByRefObject` peut être instanciée par un client distant. Nous rappelons que chaque fois qu'une instance de la classe `ObjRef` est créée pour référencer un objet du domaine d'application, un URI unique basé sur un GUID est créé pour cet objet. Une association interne et globale au domaine d'application, entre cet URI et l'objet réel, est alors créée.

Lorsqu'un canal récepteur reçoit un message représentant un appel, trois cas peuvent se présenter :

- L'appel se fait sur un service WKO. Dans ce cas un objet est activé si le mode d'appel est simple appel. Si le mode d'appel est singleton, soit un objet existe déjà pour l'URI spécifiée, soit un nouvel objet est activé.
- L'appel se fait sur un objet du domaine d'application, associé avec un URI fourni. Si un objet est effectivement associé avec cet URI, l'appel est effectué sur cet objet, sinon un message est retourné au client indiquant que l'objet n'existe pas.
- L'appel est un appel à un constructeur sur une classe CAO. Dans ce cas, un nouvel objet est créé. Cet objet est associé avec un nouvel URI basé sur un nouveau GUID. Ce nouvel URI est retourné au client à l'aide d'une instance de la classe `ObjRef`.

## *Intercepteurs de messages, formateurs et canaux*

À l'instar de ce qui se fait du côté client, du côté serveur un message représentant un appel est traité par des intercepteurs de messages, chaînés les uns aux autres. Cependant une différence conceptuelle fondamentale existe entre un canal émetteur et un canal récepteur. Un canal émetteur fabrique une chaîne d'intercepteurs de messages pour chaque proxy réel vers un objet distant. Un canal récepteur fabrique une chaîne d'intercepteurs de messages dès qu'il est créé. Cette chaîne sera utilisée pour tous les appels transitant par ce canal récepteur, quel que soit l'objet concerné par l'appel. Cette différence est illustrée par la Figure 22-6 :

Sur la Figure 22-6, chaque message est traité par quatre intercepteurs de messages, deux du côté client et deux du côté serveur. Ceci est une simplification et ne reflète pas la réalité. Il existe en fait quatre catégories d'intercepteurs de messages qui agissent au niveau du canal :

- Les intercepteurs de messages qui travaillent avec le message non sérialisé.
- Les intercepteurs de messages qui sérialisent/désérialisent le message, pour qu'il puisse transiter sur le réseau. On les appelle formateurs (*formater sink* en anglais). Il y a toujours un formateur qui sérialise dans une chaîne d'intercepteurs de messages côté client, et un formateur qui désérialise dans une chaîne d'intercepteurs de messages côté serveur. Le *framework* .NET fourni deux types de formateurs : les formateurs qui sérialisent/désérialisent un message dans un *stream* binaire et les formateurs qui sérialisent/désérialisent un message dans un document SOAP. Vous avez aussi la possibilité de créer vos propres formateurs.
- Les intercepteurs de messages qui travaillent avec le message sérialisé.

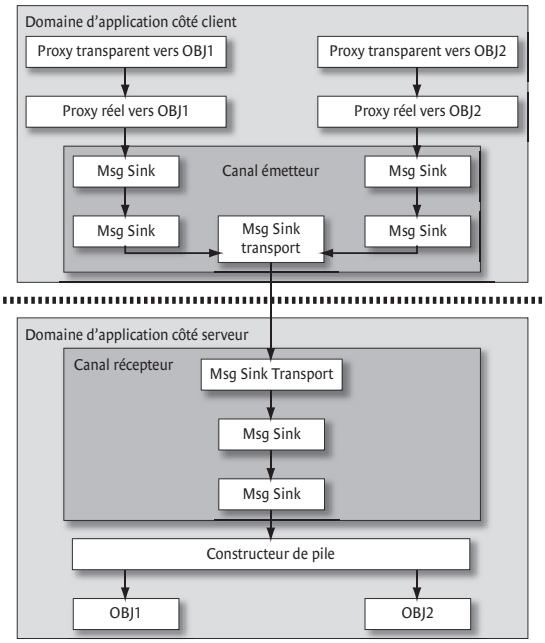


*Figure 22-6 : Intercepteurs de messages et canaux*

- Les intercepteurs de messages transporteurs, qui s'occupent de faire transiter le message sérialisé sur le réseau. Il est possible de créer votre propre niveau de transport mais ce sujet dépasse le cadre de cet ouvrage.

Toutes les classes d'intercepteurs de messages implémentent l'interface `IMessageSink`, vue précédemment, indépendamment de leur catégorie. Les classes d'intercepteurs de messages destinées à être instanciées par un canal émetteur implémentent l'interface `System.Runtime.Remoting.Channels.IClientChannelSink`. Les classes d'intercepteurs de messages destinées à être instanciées par un canal récepteur implémentent l'interface `System.Runtime.Remoting.Channels.IServerChannelSink`. Les classes de formateurs destinées à être instanciées par un canal émetteur implémentent l'interface `System.Runtime.Remoting.Channels.IClientFormatterSink` qui étend à la fois les interfaces `IMessageSink` et `IClientChannelSink`. Les classes de formateurs destinées à être instanciées par un canal récepteur implémentent l'interface `System.Runtime.Remoting.Channels.IServeurFormatterSink` qui étend à la fois les interfaces `IMessageSink` et `IServerChannelSink`.

Avant de pouvoir exposer tout ceci dans un exemple concret, il faut expliquer comment définir la production de chaînes d'intercepteurs de messages d'un canal.

## *Fournisseurs d'intercepteurs de messages (channel sink providers)*

Pour définir la production de chaînes d'intercepteurs de messages d'un canal, il faut agir au niveau de sa chaîne de fournisseurs d'intercepteurs de messages. Un *fournisseur d'intercepteurs de messages* (*channel sink providers* en anglais) d'un canal émetteur (respectivement d'un canal

récepteur) est une instance d'une classe qui implémente l'interface `System.Runtime.Remoting.Channels.IClientChannelSinkProvider` (respectivement `IServerChannelSinkProvider`). Rappelons que dans un canal émetteur, cette production est sollicitée pour chaque proxy réel alors que dans un canal récepteur, cette production est sollicitée une seule fois, à la création du canal.

Chacune de ces deux interfaces présente naturellement une méthode `CreateSink()`. Cette méthode est appelée par l'implémentation d'un canal pour obtenir un nouvel intercepteur de messages du fournisseur :

```
IServerChannelSink IServerChannelSinkProvider.CreateSink(
      IChannelReceiver  canal) ;
IClientChannelSink IClientChannelSinkProvider.CreateSink(
      IChannelSender    canal,
      String            url,
      Object            remoteChannelData) ;
```

Pour que l'on puisse créer une chaîne de tels fournisseurs, chacune de ces deux interfaces présente la propriété `Next`, du type de l'interface concernée. Vous pouvez donc facilement créer une telle chaîne dans votre code et la communiquer à un canal lors de sa construction, en utilisant les constructeurs des classes de canaux prévus à cet effet. Néanmoins on préfère toujours configurer la chaîne de fournisseurs d'un canal par l'intermédiaire du fichier de configuration Remoting de l'application. Ceci est illustré dans la section suivante.

## *Un exemple récapitulatif : Afficher le nombre d'octets passant sur le réseau*

Nous allons construire nos propres intercepteurs de messages pour canaux et nos propres fournisseurs pour ces intercepteurs. Le but est d'afficher sur la console, du côté client et du côté serveur, le nombre d'octets envoyés et reçus.

### *Intercepteurs de messages et fournisseurs*

Nous avons donc besoin de développer quatre classes pour ce projet :

- Une classe nommée `CustomClientSink`, dont les instances sont des intercepteurs de messages. Les instances de cette classe doivent être placées dans le canal émetteur, après le formateur.
- Une classe nommée `CustomServerSink`, dont les instances sont des intercepteurs de messages. Les instances de cette classe doivent être placées dans le canal récepteur, avant le formateur.
- Une classe nommée `CustomClientSinkProvider`, dont les instances sont des fournisseurs d'instances de `CustomClientSink`. Les instances de cette classe doivent être chaînées après le fournisseur de formateurs dans le canal émetteur.
- Une classe nommée `CustomServerSinkProvider`, dont les instances sont des fournisseurs d'instances de `CustomServerSink`. Les instances de cette classe doivent être chaînées avant le fournisseur de formateurs dans le canal récepteur.

Nous développons aussi une classe `Helper`, qui fournit la méthode statique `GetStreamLength()` qui retourne la taille d'un stream. Cette méthode est capable de retourner cette taille, que le

stream ait la possibilité d'accès aléatoire (*seek*) ou non. Voici le code du nouvel assemblage qui contient toutes ces classes :

*Exemple 22-30 :* *CustomChannelSink.cs*

```
using System ;
using System.Runtime.Remoting.Channels ;
using System.Runtime.Remoting.Messaging ;
using System.Collections ;
using System.IO ;

namespace CustomChannelSink {
   internal class Helper {
      static public Stream GetStreamLength(Stream inStream,
                                           out long length) {
         // Les accès aléatoires sont-ils autorisés sur InStream ?
         if (inStream.CanSeek) {
            length = inStream.Length ;
            return inStream ;
         }
         // Les accès aléatoires ne sont pas autorisés sur InStream.
         // Copie de InStream dans OutStream pour obtenir la taille.
         Stream outStream = new MemoryStream() ;
         byte[] tampon = new Byte[1024] ;
         int tmp, nBytesRead = 0 ;
         while ((tmp = inStream.Read(tampon, 0, 1024)) > 0) {
            outStream.Write(tampon, nBytesRead, tmp) ;
            nBytesRead += tmp ;
         }
         outStream.Seek(0, SeekOrigin.Begin) ;
         length = nBytesRead ;
         return outStream ;
      }
   }

   //
   // Intercepteur propriétaire de messages, pour un canal émetteur.
   //
   public class CustomClientSink : BaseChannelSinkWithProperties,
                                   IClientChannelSink {
      private IClientChannelSink m_NextSink ;

      public CustomClientSink(IClientChannelSink nextSink) {
         m_NextSink = nextSink ; }

      public IClientChannelSink NextChannelSink {
         get { return m_NextSink ; }
      }

      public void AsyncProcessRequest(
```

```
                        IClientChannelSinkStack sinkStack,
                        IMessage msgIn,
                        ITransportHeaders headers,
                        Stream msgStream) {
      long length ;
      msgStream = Helper.GetStreamLength(msgStream, out length) ;
      Console.WriteLine(
      "CustomClientSink:Async, taille du stream d'envoi {0}",length);
      // Auto-chaînage du présent intercepteur pour le retour
      // d'un appel asynchrone.
      sinkStack.Push(this, null) ;
      m_NextSink.AsyncProcessRequest(sinkStack,
                                     msgIn, headers, msgStream) ;
   }

   public void AsyncProcessResponse(
                        IClientResponseChannelSinkStack sinkStack,
                        Object state,
                        ITransportHeaders headers,
                        Stream msgStream) {
      long length ;
      msgStream = Helper.GetStreamLength(msgStream, out length) ;
      Console.WriteLine(
           "CustomClientSink:Async, taille du stream de retour {0}",
           length);

      m_NextSink.AsyncProcessResponse(
              sinkStack, state, headers, msgStream) ;
   }

   public Stream GetRequestStream(
                        IMessage msg,
                        ITransportHeaders headers) {
      return m_NextSink.GetRequestStream(msg, headers) ;
   }

   public void ProcessMessage(
                        IMessage msg,
                        ITransportHeaders headersIn,
                        Stream msgInStream,
                        out ITransportHeaders headersOut,
                        out Stream msgOutStream) {
      long length ;
      msgInStream = Helper.GetStreamLength(msgInStream, out length) ;
      Console.WriteLine(
       "CustomClientSink:Sync, taille du stream d'envoi {0}",length);
      m_NextSink.ProcessMessage(msg, headersIn, msgInStream,
                                out headersOut, out msgOutStream) ;
      msgOutStream = Helper.GetStreamLength(msgOutStream,out length) ;
```

```
            Console.WriteLine(
               "CustomClientSink:Sync, taille du stream de retour {0}",
               length);
         }
      }

      //
      // Intercepteur propriétaire de messages, pour un canal récepteur.
      //
      public class CustomServerSink : BaseChannelSinkWithProperties,
                                      IServerChannelSink {
         private IServerChannelSink m_NextSink ;

         public CustomServerSink(IServerChannelSink nextSink) {
            m_NextSink = nextSink ;
         }
         public IServerChannelSink NextChannelSink {
            get { return m_NextSink ; }
         }
         public void AsyncProcessResponse(
                           IServerResponseChannelSinkStack sinkStack,
                           object state,
                           IMessage msg,
                           ITransportHeaders headers,
                           Stream msgStream) {
            long length ;
            msgStream = Helper.GetStreamLength(msgStream, out length) ;
            Console.WriteLine(
               "CustomServerSink:Async, taille du stream de retour {0}",
               length);
            m_NextSink.AsyncProcessResponse(
                              sinkStack, state, msg, headers, msgStream) ;
         }

         public Stream GetResponseStream(
                           IServerResponseChannelSinkStack sinkStack,
                           object state,
                           IMessage msg,
                           ITransportHeaders headers) {
            return null ;
         }

         public ServerProcessing ProcessMessage(
                           IServerChannelSinkStack sinkStack,
                           IMessage msgIn,
                           ITransportHeaders headersIn,
                           Stream msgInStream,
                           out IMessage msgOut,
                           out ITransportHeaders headersOut,
```

```
                        out Stream msgOutStream) {
        long length ;
        msgInStream = Helper.GetStreamLength(msgInStream, out length) ;
        Console.WriteLine(
         "CustomServerSink:Sync, taille du stream d'envoi {0}",length);
        // Auto-chaînage du présent intercepteur pour le retour
        // pour prévoir le cas où l'appel est asynchrone.
        sinkStack.Push(this, null) ;
        ServerProcessing svrProc = m_NextSink.ProcessMessage(
                        sinkStack, msgIn, headersIn, msgInStream,
                        out msgOut, out headersOut, out msgOutStream) ;
        msgOutStream = Helper.GetStreamLength(msgOutStream,out length) ;
        Console.WriteLine(
           "CustomServerSink:Sync, taille du stream de retour {0}",
           length);
        return svrProc ;
     }
  }

  //
  // Fournisseur propriétaire d'intercepteurs de messages,
  // pour un canal émetteur.
  //
  public class CustomClientSinkProvider : IClientChannelSinkProvider {
     private IClientChannelSinkProvider m_NextProvider ;
     public CustomClientSinkProvider(IDictionary prop,
        ICollection providerData) { }

     public IClientChannelSinkProvider Next {
        get { return m_NextProvider ; }
        set { m_NextProvider = value ; }
     }

     public IClientChannelSink CreateSink(
                    IChannelSender canal,
                    string url,
                    object remoteChannelData) {
        IClientChannelSink next =
            m_NextProvider.CreateSink(canal, url, remoteChannelData) ;
        Console.WriteLine(
        "CustomClientSinkProvider:Creation d'un intercepteur de msg.");
        return new CustomClientSink(next) ;
     }
  }

  //
  // Fournisseur propriétaire d'intercepteurs de messages,
  // pour un canal récepteur.
  //
```

```
   public class CustomServerSinkProvider : IServerChannelSinkProvider {
      private IServerChannelSinkProvider m_NextProvider ;
      public CustomServerSinkProvider(IDictionary prop,
         ICollection providerData) { }

      public IServerChannelSinkProvider Next {
         get { return m_NextProvider ; }
         set { m_NextProvider = value ; }
      }
      public IServerChannelSink CreateSink(IChannelReceiver canal) {
         IServerChannelSink next = m_NextProvider.CreateSink(canal) ;
         Console.WriteLine(
         "CustomServerSinkProvider:Creation d'un intercepteur de msg.");
         return new CustomServerSink(next) ;
      }

      public void GetChannelData(IChannelDataStore channelData) { }
   }
}
```

Quelques précisions s'imposent :

- Dans les constructeurs des fournisseurs, le dictionnaire passé en argument correspond aux propriétés que vous souhaitez affecter aux fournisseurs. Ici nous n'utilisons pas cette possibilité. Il suffirait d'écrire dans le fichier de configuration :

  ```
  <provider type = "CustomChannelSink.CustomClientSinkProvider,ChannelSink"
      Prop1="hello"/>
  ```

  Vous pouvez obtenir la valeur de la propriété comme ceci :

  ```
  public CustomClientSinkProvider(IDictionary Prop,
           ICollection ProviderData) {
     string s = (string) Prop["Prop1"];
     ...
  ```

  Vous pouvez ainsi configurer votre fournisseur. Par exemple vous pouvez spécifier le nom d'un algorithme de compression ou de cryptage des *streams*.
- Le type `IServerChannelSinkStack` utilisé dans les méthodes dans l'intercepteur de messages côté serveur, concerne le retour des appels asynchrone. Cette pile correspond en fait à la chaîne d'intercepteurs de messages qui sera utilisée du côté serveur, pour faire transiter le message de retour.
- Le type `ITransportHeaders` utilisé dans les méthodes dans les intercepteurs de messages côté client et côté serveur, permet de passer des informations concernant le message. Par exemple, si la fonction des intercepteurs de message est de crypter/décrypter le *stream*, vous pouvez spécifier dans cet en-tête si le message est effectivement crypté :

  ```
  class CustomClientSink{ ...
     public void ProcessMessage(
             IMessage              Msg,
  ```

```
            ITransportHeaders      HeadersIn, ...){
        HeadersIn["Crypté"] ="Affirmatif";
        ...
```

Cela permet à l'intercepteur dans le canal récepteur de tester que le message a été effectivement crypté, avant d'essayer de le décrypter.

```
class CustomServerSink{ ...
   public ServerProcessing ProcessMessage(
            IServerChannelSinkStack   SinkStack,
            IMessage                  MsgIn,
            ITransportHeaders         HeadersIn,...){
      String sCrypté = (string) HeadersIn["Crypté"] ;
      if( sCrypté != null &&   sCrypté == "Affirmatif"){ ...
```

Ce canal récepteur devient ainsi polyvalent, et peut traiter à la fois les messages cryptés et non cryptés.

### *Côté serveur*

Du côté serveur, toutes les informations relatives aux canaux se trouvent dans le fichier de configuration :

Serveur.cs

```
...
static void Main() {
   RemotingConfiguration.Configure("Serveur.config") ;
   Console.WriteLine("Pressez une touche pour stopper le serveur...") ;
   Console.Read() ;
}
...
```

Nous affectons ici un formateur binaire avec un canal récepteur HTTP.

*Exemple 22-31 :* *Serveur.config*

```
<configuration>
   <system.runtime.remoting>
      <application name = "Serveur">
         <service>
            <wellknown type="NommageInterface.CAdditionneur,Interface"
             mode ="Singleton" objectUri="Service1.rem" />
         </service>
         <channels>
            <channel port="65100" ref ="http">
               <serverProviders>
                  <provider type=
              "CustomChannelSink.CustomServerSinkProvider,ChannelSink" />
                  <formatter ref="binary"/>
               </serverProviders>
            </channel>
         </channels>
```

```
      </application>
   </system.runtime.remoting>
</configuration>
```

### *Côté client*

Du côté client, toutes les informations pertinentes sont également dans le fichier de configuration :

Client.cs

```
...
static void Main() {
   RemotingConfiguration.Configure("Client.config",false) ;

   CAdditionneur objA = new CAdditionneur() ;
   double dA = objA.Add( 3.0 , 4.0 ) ;
   Console.WriteLine("Valeur retournée:"+dA) ;
   CAdditionneur objB = new CAdditionneur() ;
   double dB = objB.Add( 3.0 , 4.0 ) ;
   Console.WriteLine("Valeur retournée:"+dB) ;
}
...
```

Nous affectons ici un formateur binaire avec un canal émetteur HTTP.

*Exemple 22-32 :* *Client.config*

```
<configuration>
   <system.runtime.remoting>
      <application name = "Client">
         <client>
            <wellknown type="NommageInterface.CAdditionneur,Interface"
             url="http://localhost:65100/Service1.rem" />
         </client>
         <channels>
            <channel ref ="http">
               <clientProviders>
                  <formatter ref="binary"/>
                  <provider type =
             "CustomChannelSink.CustomClientSinkProvider,ChannelSink"/>
               </clientProviders>
            </channel>
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

### *Exécution du projet*

Voici ce qu'affiche le serveur sur sa console :

```
CustomServerSinkProvider:Creation d'un intercepteur de msg.
Pressez une touche pour quitter le serveur...
CustomServerSink:Sync, taille du stream d'envoi 155
CAdditionneur ctor
CAdditionneur Add( 3 + 4 )
CustomServerSink:Sync, taille du stream de retour 32
CustomServerSink:Sync, taille du stream d'envoi 155
CAdditionneur Add( 3 + 4 )
CustomServerSink:Sync, taille du stream de retour 32
```

Voici ce qu'affiche le client sur sa console :

```
CustomClientSinkProvider:Creation d'un intercepteur de msg.
CustomClientSink:Sync, taille du stream d'envoi 155
CustomClientSink:Sync, taille du stream de retour 32
Valeur retournée:7
CustomClientSinkProvider:Creation d'un intercepteur de msg.
CustomClientSink:Sync, taille du stream d'envoi 155
CustomClientSink:Sync, taille du stream de retour 32
Valeur retournée:7
```

Si l'on avait choisi un formateur SOAP et non pas un formateur binaire, la taille du stream d'envoi aurait été de 574 octets et la taille du stream de retour aurait été de 586 octets.

Si vous disposez d'une bibliothèque de compression ou de cryptage, cet exemple pourrait être très facilement modifié pour compresser ou crypter les données binaires passants sur le réseau.

# *Contexte .NET*

## *La place de la notion de contexte dans l'architecture .NET*

Nous avons vu que les domaines d'application permettent l'isolation à l'exécution au niveau des types, de la sécurité et de la gestion des exceptions. En revanche, il existe une entité plus fine que le domaine d'application pour stocker les objets. Un domaine d'application .NET peut contenir plusieurs de ces entités, appelées *contexte .NET*. Pour simplifier la lecture, on les appellera contextes tant qu'il n'y a pas de confusion avec la notion de contexte COM+, qui est différente (cette notion est présentée page 300). Certains auteurs utilisent les termes *contexte géré* ou *contexte d'exécution* pour un contexte .NET et le terme de *contexte non géré* pour les contextes COM+. Tous les objets .NET « vivent » dans un contexte, et il existe au moins un contexte par domaine d'application. Ce contexte, appelé *contexte par défaut* du domaine d'application, est créé en même temps que son domaine. La Figure 22-7 résume ces relations d'inclusions :

La notion de contexte offre aux développeurs la fonctionnalité très pratique d'interception des appels à un objet. Nous rappelons qu'intercepter un appel signifie que l'on peut effectuer un ou plusieurs traitements à chaque appel entrant ou sortant vers un objet d'un contexte. Ces traitements sont matérialisés par des intercepteurs de messages. L'idée centrale est que lorsqu'un client appelle une méthode d'un objet, il n'a pas conscience que son appel est intercepté et que des traitements sont effectués avant et après l'exécution de l'appel.

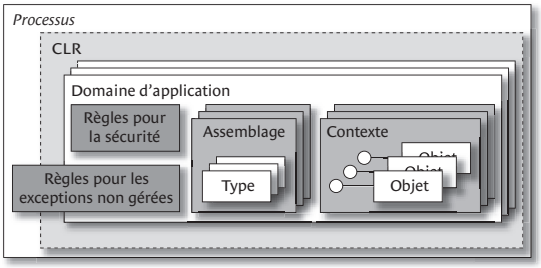


*Figure 22-7 : Processus, domaines d'application et contextes*

## *Context-bound et context-agile*

D'après la section précédente, un contexte peut être vu comme une zone d'un domaine d'application, contenant des objets et des intercepteurs de messages. Les appels vers les objets d'un contexte sont transformés en messages qui sont interceptés et traités par les intercepteurs du message. Nous savons maintenant que pour transformer un appel en un message, il faut passer par l'intermédiaire d'un proxy transparent. Or, nous savons aussi que le CLR ne fabrique un proxy transparent vers un objet que si ce dernier est une instance d'une classe dérivée de `MarshalByRefObject` appelée par une entité située hors de son domaine d'application. Ici, nous souhaiterions bénéficier du mécanisme d'interception de messages pour tous les appels, même ceux effectués entre entités situées dans le même domaine d'application. C'est exactement pour cela qu'existe la classe `System.ContextBoundObject`. Une instance d'une classe qui dérive de `ContextBoundObject` est accessible seulement à partir de proxys transparents. Dans ce cas, même la référence `this` utilisée dans les méthodes de la classe est un proxy transparent et non une référence directe. Il est logique que la classe `ContextBoundObject` dérive de la classe `MarshalByRefObject`, puisqu'elle ne fait que renforcer le comportement de cette classe qui est d'indiquer au CLR qu'une classe est potentiellement utilisée au travers d'un proxy transparent.

Une instance d'une classe dérivant de `ContextBoundObject` est qualifiée de *context-bound* (*liée au contexte* en français). Une instance d'une classe ne dérivant pas de `ContextBoundObject` est qualifiée de *context-agile* (*désolidarisés du contexte* en français). Un objet context-bound est toujours exécuté au sein de son contexte. Dans le cas d'un appel non distant, un objet context-agile est toujours exécuté au sein du contexte de celui qui l'appelle. La Figure 22-8 illustre ceci.

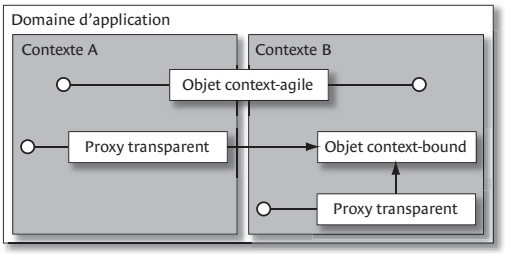


*Figure 22-8 : Objets context-bound vs. objets context-agile*

Illustrons maintenant ces notions de context-bound et context-agile avec un programme qui utilise des objets context-bound et des objets context-agile.

Nous allons utiliser dans ce programme la classe `System.Runtime.Remoting.Contexts.Context` dont une instance représente un contexte géré, un peu comme une instance de la classe `Thread` représente un thread géré. Nous allons aussi utiliser dans ce programme l'attribut `System.Runtime.Remoting.Contexts.Context.Synchronization` appliqué à une classe dérivant de `ContextBoundObject`. Cet attribut oblige les instances de cette classe à être dans un contexte spécialement créé pour supporter ce service de synchronisation. Le service de synchronisation proposé par cet attribut est décrit page 160. Ce service de synchronisation est assuré par un mécanisme d'interception de messages complètement transparent aussi bien pour l'appelant que pour l'appelé.

Nous profitons aussi de cet exemple pour montrer l'utilisation de la méthode statique `bool IsOutOfContext(object)` de la classe `System.Runtime.Remoting.RemotingServices` qui permet de savoir si une référence pointe vers un objet hors du contexte de l'appelant de cette méthode.

*Exemple 22-33 :*

```
using System ;
using System.Runtime.Remoting.Contexts ;
using System.Runtime.Remoting ;
using System.Threading ;

[Synchronization(SynchronizationAttribute.REQUIRED)]
public class Foo1 : ContextBoundObject {
   public void AfficheContexte() {
      Console.WriteLine("Foo1: ContextID:" +
                        Thread.CurrentContext.ContextID) ;
   }
   public Foo2 GetFoo2() {
      Foo2 obj = new Foo2() ;
      obj.AfficheContexte();
      return obj ;
   }
}

public class Foo2 {
   public void AfficheContexte() {
      Console.WriteLine("Foo2: ContextID:" +
                        Thread.CurrentContext.ContextID) ;
   }
}

public class Program {
   static void Main() {
      Console.WriteLine("Main: ContextID:" +
                        Thread.CurrentContext.ContextID) ;
      Foo1 obj1 = new Foo1() ;
```

```
      obj1.AfficheContexte();
      Console.WriteLine("IsObjectOutOfContext(obj1):" +
                        RemotingServices.IsObjectOutOfContext(obj1)) ;

      Foo2 obj2 = obj1.GetFoo2() ;
      obj2.AfficheContexte();
      Console.WriteLine("IsObjectOutOfContext(obj2):" +
                        RemotingServices.IsObjectOutOfContext(obj2)) ;
   }
}
```

Ce programme affiche :

```
Main: ContextID:0
Foo1: ContextID:1
IsObjectOutOfContext(obj1):True
Foo2: ContextID:1
Foo2: ContextID:0
IsObjectOutOfContext(obj2):False
```

On voit bien que l'objet `obj2` context-agile peut être exécuté hors du contexte dans lequel il a été créé. On observe aussi que l'objet `obj1` context-bound s'exécute au sein de son contexte lorsqu'on l'appelle à partir d'un autre contexte.

Dans un même domaine d'application, les `ContextID` sont tous différents. Cependant il est possible que deux `ContextID` de deux contextes appartenant à deux domaines d'application différents soient égaux.

D'après nos propres tests, dans une méthode statique, la propriété `Thread.CurrentContext.ContextID` est égale au `ContextID` du contexte par défaut du domaine d'application. Ce fait est logique étant donné que le comportement du CLR est de créer un objet dans le contexte du client, si cela est possible. Or, une méthode statique peut constituer un client d'un objet.

## *Attribut de contexte et propriété de contexte*

Nous allons exposer ici la technique permettant d'injecter des intercepteurs de messages au niveau du contexte. Commençons par introduire les notions d'attribut de contexte et de propriété de contexte.

### *Attribut de contexte*

Un *attribut de contexte* est un attribut .NET qui agit sur une classe context-bound. Une classe attribut de contexte implémente l'interface `System.Runtime.Remoting.Contexts.IContextAttribute`. Une classe context-bound peut avoir plusieurs attributs de contexte. Lors de la création d'un objet de cette classe, chaque attribut de contexte de la classe vérifie si le contexte de celui qui construit l'objet lui convient. Cette opération s'effectue dans la méthode :

```
public bool IContextAttribute.IsContextOK(
                            Context                   ctxDuClient,
                            IConstructionCallMessage  ctorMsg)
```

Si au moins un attribut de contexte retourne `false`, le CLR doit créer un nouveau contexte pour accueillir le nouvel objet. Dans ce cas chaque attribut de contexte peut injecter une ou plusieurs propriétés de contexte dans le nouveau contexte. Ces injections ont lieu dans la méthode suivante :

```
public void IContextAttribute.GetPropertiesForNewContext(
                              IConstructionCallMessage    ctorMsg)
```

### *Propriété de contexte*

Une propriété de contexte est une instance d'une classe qui implémente l'interface `System.Runtime.Remoting.Contexts.IContextProperty`. Chaque contexte peut avoir plusieurs propriétés. Les propriétés d'un contexte sont injectées par les attributs de contexte dans le contexte, lorsque le contexte est créé. Une fois que chaque attribut de contexte a injecté ses propriétés, la méthode suivante est appelée sur chaque propriété. Il n'est alors plus possible d'injecter une propriété dans ce contexte :

```
public void IContextProperty.Freeze(Context ctx)
```

Le CLR demande ensuite à chaque propriété si elle est satisfaite par ce nouveau contexte en appelant la méthode :

```
public bool IContextProperty.IsNewContextOK(Context ctx)
```

Chaque propriété d'un contexte a un nom défini par la propriété `Name` :

```
public string IContextProperty.Name{ get }
```

Les méthodes des objets hébergés dans le contexte peuvent avoir accès aux propriétés du contexte en appelant la méthode :

```
IContextProperty Context.GetProperty(string sPropertyName)
```

Cette possibilité peut être intéressante, puisque les objets du contexte peuvent ainsi partager des informations et accéder à des services grâce aux propriétés de leur contexte. Cependant le rôle principal des propriétés d'un contexte n'est pas de fournir cette possibilité :

> Le rôle principal des propriétés d'un contexte est d'injecter des intercepteurs de messages, dans les régions d'interception de messages des contextes concernés.

La description de ces régions d'interceptions fait l'objet de la prochaine section. Familiarisons-nous d'abord avec ces concepts d'attributs et de propriétés d'un contexte à l'aide d'un exemple.

Pour ceux qui ont assimilé la section sur les canaux, un attribut de contexte joue un peu le rôle d'un fichier de configuration qui injecte des fournisseurs dans les canaux. De même une propriété de contexte joue un peu le rôle du fournisseur d'intercepteur de message.

### *Exemple utilisant un attribut et une propriété de contexte*

Le programme suivant définit la classe d'attribut de contexte `LogContextAttribute` et la classe de propriété de contexte `LogContextProperty`. Toute instance d'une classe ayant pour attribut `LogContextAttribute` est hébergée dans un contexte qui a une propriété de type `LogContextProperty`. Une telle instance peut ainsi avoir accès aux services présentés par cette

propriété. Ces services sont en l'occurrence la possibilité d'écrire une chaîne de caractères dans un fichier en appelant la méthode `LogContextProperty.Log(string)`. Le nom de ce fichier est un paramètre de l'attribut `LogContextAttribute`. On peut ainsi avoir un fichier pour chaque classe. Lorsqu'une nouvelle instance d'une classe ayant pour attribut `LogContextAttribute` est créée, la méthode `bool LogContextAttribute.IsContextOK(Context)` permet de vérifier si le contexte dans lequel réside l'entité qui appelle le constructeur, contient déjà une instance de `LogContextAttribute` avec le même nom de fichier. Si ce n'est pas le cas, un nouveau contexte doit alors être créé. La méthode `LogContextAttribute.GetPropertiesForNewContext(IConstructionCallMessage ctor)` crée une instance de `LogContextProperty`. Au retour de cette méthode, la nouvelle propriété est automatiquement injectée dans le nouveau contexte par le CLR. Voici le programme :

*Exemple 22-34 :*

```
using System ;
using System.Runtime.Remoting.Contexts ;
using System.Runtime.Remoting.Activation ;
using System.Threading ;

public class LogContextProperty : IContextProperty {
   public LogContextProperty(string sFileName) {
      m_sFileName = sFileName ; }
   string m_sFileName ;
   public string sFileName { get { return m_sFileName ; } }
   public string Name { get { return "Log" ; } }
   public bool IsNewContextOK(Context ctx) { return true ; }
   public void Freeze(Context ctx) { }
   public void Log( string sLog ) {
      // Nous nous contentons d'écrire les données sur la console.
      Console.WriteLine(
              "ContextID={0} Ecrire '{1}' dans le fichier {2}",
              Thread.CurrentContext.ContextID,
              sLog,
              m_sFileName) ;
   }
}

[AttributeUsage(AttributeTargets.Class)]
public class LogContextAttribute : Attribute, IContextAttribute {
   string m_sFileName ;
   public LogContextAttribute(string sFileName) {
      m_sFileName = sFileName ; }
   // Pas la peine de créer un nouveau contexte si celui a déjà
   // la propriété de log dans le bon fichier.
   public bool IsContextOK(Context ctx_courant,
                           IConstructionCallMessage ctor) {
      LogContextProperty prop = ctx_courant.GetProperty("Log")
                  as LogContextProperty;
      if ( prop == null ) return false;
      return ( prop.sFileName == m_sFileName );
```

```
    }
    public void GetPropertiesForNewContext(
                            IConstructionCallMessage ctor) {
       IContextProperty prop = new LogContextProperty(m_sFileName);
       ctor.ContextProperties.Add(prop);
    }
}

[LogContextAttribute("Log_Foo.txt")]
public class Foo : ContextBoundObject {
   public Foo CreateNewInst() {
      return new Foo() ;
   }
   public int Ajout(int a, int b) {
      string s = string.Format("Ajout de {0}+{1}", a, b) ;
      Context ctx = Thread.CurrentContext ;
      LogContextProperty Logger =
          ctx.GetProperty("Log") as LogContextProperty;
      Logger.Log(s);
      return a + b ;
   }
}

public class Program {
   static void Main() {
      Foo obj1 = new Foo() ;
      obj1.Ajout(4, 5) ;
      Foo obj2 = new Foo() ;
      obj2.Ajout(6, 7) ;
      Foo obj3 = obj1.CreateNewInst() ;
      obj3.Ajout(8, 9) ;
   }
}
```

Ce programme affiche :

```
ContextID=1 Ecrire 'Ajout de 4+5' dans le fichier LogPourFoo.txt
ContextID=2 Ecrire 'Ajout de 6+7' dans le fichier LogPourFoo.txt
ContextID=1 Ecrire 'Ajout de 8+9' dans le fichier LogPourFoo.txt
```

Les objets `obj1` et `obj3` sont hébergés dans le même contexte puisque `obj3` a été construit à partir du contexte d'`obj1`.

## *Notion de région d'interception de messages*

Il existe quatre *régions d'interception de messages*, la région « serveur », la région « objet », la région « envoi » et la région « client ». Pour comprendre ces notions de régions, il faut se placer dans le cas où un objet context-bound est appelé par une entité située dans un autre contexte. Cette entité peut être une méthode statique ou un autre objet. Dans notre discussion sur ces régions, on appelle le contexte de cette entité le *contexte appelant* et le contexte de l'objet appelé

le *contexte cible*. Chacune des propriétés de contexte du contexte cible a la possibilité d'injecter des intercepteurs de messages dans chacune de ces régions.

- Les intercepteurs de messages injectés dans la région « serveur » interceptent tous les messages d'appels venant d'un autre contexte vers tous les objets du contexte cible. Il existe donc une région « serveur » par contexte cible.
- Les intercepteurs de messages injectés dans la région « objet » interceptent tous les messages d'appels venant d'un autre contexte vers un objet particulier du contexte cible. Il y a donc une région « objet » pour chaque objet d'un contexte. Les régions « objet » se situent donc dans le contexte cible.
- Les intercepteurs de messages injectés dans la région « envoi » interceptent tous les messages d'appels venant d'un autre contexte vers un objet particulier du contexte cible. Il y a donc une région « envoi » pour chaque objet du contexte cible. La différence entre une région « envoi » et une région « objet » est qu'une région « envoi » se situe dans le contexte appelant l'objet et non pas dans le contexte cible contenant l'objet. On utilise les régions « envoi » pour transmettre aux intercepteurs de messages du contexte cible des informations sur le contexte appelant.
- Les intercepteurs de messages injectés dans la région « client » interceptent tous les messages d'appels du contexte cible vers des objets situés dans d'autres contextes. Il y a donc une région « client » pour chaque contexte cible.

La Figure 22-9 illustre ces notions de régions. Le contexte cible contient deux objets OBJ1 et OBJ2. Nous avons choisi de placer deux objets dans le contexte cible et pas un, pour bien montrer que les types de région « objet » et « envoi » sont concernées par l'interception des messages au niveau d'un objet alors que les types de région « serveur » et « client » sont concernées par l'interception des messages au niveau d'un contexte.

Nous avons placé deux intercepteurs propriétaires de messages (*Msg sink*) par région pour bien montrer que chaque région peut avoir zéro, un ou plusieurs intercepteurs de messages. Concrètement, tous ces intercepteurs propriétaires de messages sont injectés dans les régions par les propriétés du contexte cible, même lorsque la région n'appartient pas au contexte cible. Comme vous pouvez définir vos propres classes de propriétés de contexte vous pouvez choisir quels sont les intercepteurs de messages qu'il faut injecter.

Vous remarquez que chaque région contient un intercepteur système de messages dit de terminaison, qui permet d'indiquer au CLR la sortie d'une région. Concrètement, vous n'aurez pas à vous préoccuper de ces intercepteurs systèmes de messages de terminaison.

Sachez que lorsque le contexte appelant et le contexte cible sont dans le même domaine d'application, le CLR utilise une instance de la classe `CrossContextChannel` interne à `mscorlib.dll`. Cette instance fait basculer la propriété `Context` du thread courant. Notre figure représente ces instances.

## *Exemple d'utilisation des régions d'interception*

Voici tous ce que nous exposons avec le programme de l'Exemple 22-35 :

- Le code d'un attribut de contexte propriétaire (classe `MyAfficheContextAttribute`) injectant une propriété de contexte propriétaire (classe `MyAfficheContextProperty`) dans

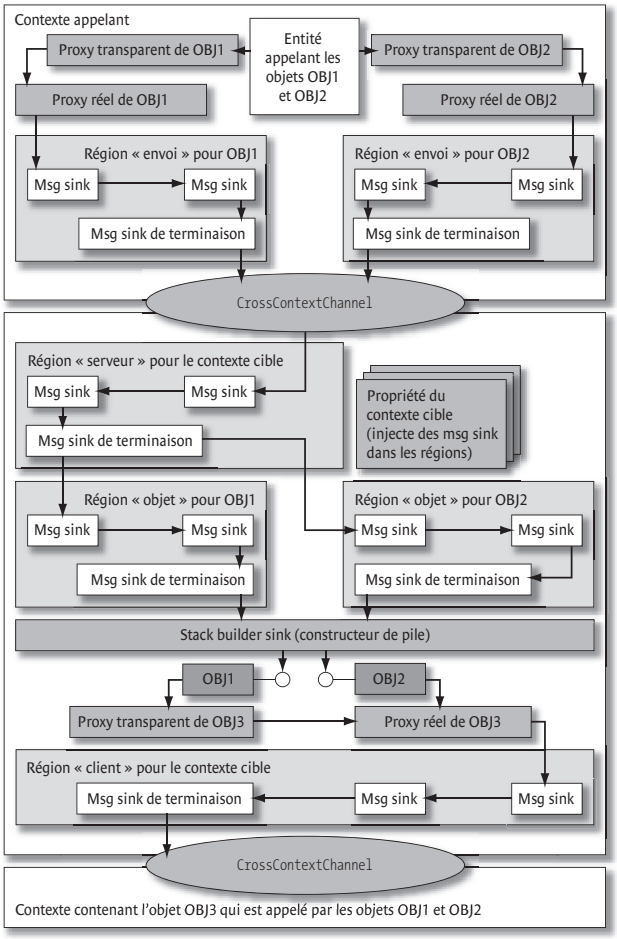


*Figure 22-9 : Interceptions des appels au niveau du contexte*

un contexte. Cette propriété de contexte injecte des intercepteurs propriétaires de messages (classe `MyAfficheMessageSink`) dans les régions « objet » « serveur » et « client » du contexte cible et dans la région « envoi » du contexte appelant.

- La modification du comportement des intercepteurs de messages en fonction d'un paramètre passé dans l'attribut du contexte. En l'occurrence, ce paramètre est un booléen qui indique aux intercepteurs de messages s'ils doivent ou non afficher quelque chose sur la console.
- L'injection d'intercepteurs de messages dans chacune des quatre régions par une propriété de contexte. L'attribut de contexte `MyAfficheContextAttribute` fait en sorte de créer un contexte pour chaque objet de la classe `Foo`. Nous créons une première instance de `Foo` et nous appelons une méthode sur cette instance pour passer par les intercepteurs de messages des régions « envoi », « serveur » et « objet ». Pour passer par les intercepteurs de messages

de la région « client », nous créons une seconde instance de `Foo` et nous l'appelons à partir de la première instance. Il y a donc trois contextes en jeu : le contexte dans lequel s'exécute la méthode `Main()` (`ContextID=0`), le contexte qui contient la première instance de `Foo` (`ContextID=1`) et le contexte qui contient la seconde instance de `Foo` (`ContextID=2`).

- Le fait qu'un appel « intra contexte » ne déclenche pas l'appel de tous ces intercepteurs de messages. Ce comportement est exposé lorsque la première instance de `Foo` s'auto appelle.
- L'injection de plusieurs intercepteurs de messages dans une région (en l'occurrence la région « client »).
- Le moment où le CLR injecte effectivement des intercepteurs de messages dans les régions. On voit clairement que ce moment dépend du type de la région.

Voici le programme :

*Exemple 22-35 :*

```
using System ;
using System.Runtime.Remoting.Contexts ;
using System.Runtime.Remoting.Messaging ;
using System.Runtime.Remoting.Activation ;
using System.Threading ;
using System.Collections ;

//
// Les instances de cette classe de propriété de contexte injectent
// des intercepteurs de messages dans les quatre régions d'interception.
//
public class MyAfficheContextProperty :
      IContextProperty,
      IContributeEnvoySink,
      IContributeObjectSink,
      IContributeServerContextSink,
      IContributeClientContextSink {
   public MyAfficheContextProperty(bool bAffiche) {
      m_bAffiche = bAffiche ;
   }
   bool m_bAffiche ;
   public bool bAffiche { get { return m_bAffiche ; } }

   // IContextProperty
   public string Name { get { return "PropAffiche" ; } }
   public bool IsNewContextOK(Context ctx) { return true ; }
   public void Freeze(Context ctx) {
      Console.WriteLine("   Freeze ContextID={0}", ctx.ContextID) ;
   }

   // Injection de deux intercepteurs de messages
   // dans la région 'client'.
   public IMessageSink GetClientContextSink( IMessageSink nextSink) {
      Console.WriteLine("   GetClientContextSink()") ;
```

```
      IMessageSink nextnextSink = new MyAfficheMessageSink(
                              nextSink, "Région Client1  ", m_bAffiche) ;
      return new MyAfficheMessageSink(
            nextnextSink, "Région Client2  ", m_bAffiche) ;
   }

   // Injection d'un intercepteur de messages dans la région 'serveur'.
   public IMessageSink GetServerContextSink( IMessageSink nextSink) {
      Console.WriteLine("   GetServerContextSink()") ;
      return new MyAfficheMessageSink(
            nextSink, "Région Serveur  ", m_bAffiche) ;
   }

   // Injection d'un intercepteur de messages dans la région 'envoi'
   // NOTE:Vous pouvez vous servir de 'mbro' pour obtenir une référence
   // vers l'objet et ainsi modifier l'injection d'intercepteurs de
   // messages en fonction de l'objet.
   public IMessageSink GetEnvoySink( MarshalByRefObject mbro,
                                     IMessageSink nextSink) {
      Console.WriteLine("   GetEnvoySink()") ;
      return new MyAfficheMessageSink(
                              nextSink, "Région Envoi    ", m_bAffiche) ;
   }

   // Injection d'un intercepteur de messages dans la région 'objet'
   // NOTE : même remarque sur 'mbro' que ci-dessus.
   public IMessageSink GetObjectSink( MarshalByRefObject mbro,
                                      IMessageSink nextSink) {
      Console.WriteLine("   GetObjectSink()") ;
      return new MyAfficheMessageSink(
                              nextSink, "Région Objet    ", m_bAffiche) ;
   }
}

//---------------------------------------------------------------------
//
// Attribut de contexte qui force la création d'un contexte par objet
// et qui injecte une instance de MyAfficheContextProperty dans chaque
// nouveau contexte créé.
//
[AttributeUsage(AttributeTargets.Class)]
public class MyAfficheContextAttribute : Attribute, IContextAttribute {
   bool m_bAffiche ;
   public MyAfficheContextAttribute(bool bAffiche) {
     m_bAffiche = bAffiche ;
   }
   // Force à créer un nouveau context pour chaque nouvel objet.
   public bool IsContextOK(Context ctx_courant,
                           IConstructionCallMessage ctor) {
```

```
      return false ;
   }

   // Injection d'une nouvelle instance de MyAfficheContextProperty
   // dans chaque contexte créé.
   public void GetPropertiesForNewContext(
                                 IConstructionCallMessage ctor){
      IContextProperty prop = new MyAfficheContextProperty(m_bAffiche) ;
      ctor.ContextProperties.Add(prop) ;
   }
}

//------------------------------------------------------------------------
//
// Les instances de MyAfficheMessageSink sont des intercepteurs de
// messages qui ne font que signaler leur présence en affichant deux
// lignes sur la console (ils n'affichent rien si (m_bAffiche==false) ).
//
[Serializable]
public class MyAfficheMessageSink : IMessageSink {
   // Prochain intercepteur de messages à qui transmettre le message.
   IMessageSink m_NextSink ;
   // Message à afficher.
   string m_sAffiche ;
   // Doit-on afficher un message sur la console ?
   bool m_bAffiche ;

   public IMessageSink NextSink { get { return m_NextSink ; } }
   public MyAfficheMessageSink(IMessageSink nextSink,
                               string sAffiche,
                               bool bAffiche) {
      m_NextSink = nextSink ;
      m_sAffiche = sAffiche ;
      m_bAffiche = bAffiche ;
   }
   public IMessage SyncProcessMessage(IMessage msg) {
      if (m_bAffiche)
         Console.WriteLine("   Deb MsgSink:{0} ContextID={1}",
                           m_sAffiche, Thread.CurrentContext.ContextID);
      // Transmet le msg au prochain intercepteur de message...
      IMessage retMsg = m_NextSink.SyncProcessMessage(msg) ;
      if (m_bAffiche)
         Console.WriteLine("   Fin MsgSink:{0} ContextID={1}",
                           m_sAffiche, Thread.CurrentContext.ContextID);
      return retMsg ;
   }
   public IMessageCtrl AsyncProcessMessage(IMessage msg,
                                           IMessageSink replySink) {
      return m_NextSink.AsyncProcessMessage(msg, replySink) ;
```

```
   }
}

//---------------------------------------------------------------------
//
// La classe Foo supporte l'attribut de contexte
// MyAfficheContextAttribute.
// L'argument 'true' signifie que les intercepteurs de messages en
// présence d'un message concernant un appel vers une instance de
// Foo, peuvent effectuer leurs affichages sur la console.
//
[MyAfficheContextAttribute(true)]
public class Foo : ContextBoundObject {
   public Foo() {
      Console.WriteLine("   Constructeur de Foo") ;
   }
   public int Ajout(int a, int b) {
      Console.WriteLine("   Ajout de {0}+{1}", a, b) ;
      return a + b ;
   }
   public int AjoutCross(Foo tmp, int a, int b) {
      Console.WriteLine("   Ajout cross de {0}+{1}", a, b) ;
      return tmp.Ajout(a, b) ;
   }
}

//---------------------------------------------------------------------
//
public class Program {
   static void Main() {
      Console.WriteLine( "Avant la construction de obj1..." ) ;
      Foo obj1 = new Foo() ;
      Console.WriteLine( "...avant l'appel à obj1..." ) ;
      obj1.Ajout(4, 5) ;
      Console.WriteLine( "...avant la construction de obj2..." ) ;
      Foo obj2 = new Foo() ;
      Console.WriteLine( "...avant qu'obj1 appelle obj2..." ) ;
      obj1.AjoutCross(obj2, 6, 7) ;
      Console.WriteLine( "...avant qu'obj1 appelle obj1..." ) ;
      obj1.AjoutCross(obj1, 8, 9) ;
   }
}
```

L'exécution de ce programme affiche :

```
Avant la construction de obj1...
   Freeze ContextID=1
   GetServerContextSink()
   Deb MsgSink:Région Serveur   ContextID=1
   Constructeur de Foo
```

```
   GetEnvoySink()
   Fin MsgSink:Région Serveur   ContextID=1
...avant l'appel à obj1...
   Deb MsgSink:Région Envoi     ContextID=0
   Deb MsgSink:Région Serveur   ContextID=1
   GetObjectSink()
   Deb MsgSink:Région Objet     ContextID=1
   Ajout de 4+5
   Fin MsgSink:Région Objet     ContextID=1
   Fin MsgSink:Région Serveur   ContextID=1
   Fin MsgSink:Région Envoi     ContextID=0
...avant la construction de obj2...
   Freeze ContextID=2
   GetServerContextSink()
   Deb MsgSink:Région Serveur   ContextID=2
   Constructeur de Foo
   GetEnvoySink()
   Fin MsgSink:Région Serveur   ContextID=2
...avant qu'obj1 appelle obj2...
   Deb MsgSink:Région Envoi     ContextID=0
   Deb MsgSink:Région Serveur   ContextID=1
   Deb MsgSink:Région Objet     ContextID=1
   Ajout cross de 6+7
   Deb MsgSink:Région Envoi     ContextID=1
   GetClientContextSink()
   Deb MsgSink:Région Client2   ContextID=1
   Deb MsgSink:Région Client1   ContextID=1
   Deb MsgSink:Région Serveur   ContextID=2
   GetObjectSink()
   Deb MsgSink:Région Objet     ContextID=2
   Ajout de 6+7
   Fin MsgSink:Région Objet     ContextID=2
   Fin MsgSink:Région Serveur   ContextID=2
   Fin MsgSink:Région Client1   ContextID=1
   Fin MsgSink:Région Client2   ContextID=1
   Fin MsgSink:Région Envoi     ContextID=1
   Fin MsgSink:Région Objet     ContextID=1
   Fin MsgSink:Région Serveur   ContextID=1
   Fin MsgSink:Région Envoi     ContextID=0
...avant qu'obj1 appelle obj1...
   Deb MsgSink:Région Envoi     ContextID=0
   Deb MsgSink:Région Serveur   ContextID=1
   Deb MsgSink:Région Objet     ContextID=1
   Ajout cross de 8+9
   Ajout de 8+9
   Fin MsgSink:Région Objet     ContextID=1
   Fin MsgSink:Région Serveur   ContextID=1
   Fin MsgSink:Région Envoi     ContextID=0
```

## *Passage d'information entre le contexte appelant et le contexte cible (call context)*

Vous avez la possibilité de passer des informations entre un intercepteur de messages s'exécutant dans le contexte appelant, et un intercepteur de messages s'exécutant dans le contexte cible. On utilise cette technique principalement pour faire passer une information concernant le contexte appelant au contexte cible (par exemple l'information : le contexte appelant supporte-t-il certaines propriétés ?). Cette fonctionnalité porte le nom de *call context*. Malgré son nom, cette fonctionnalité peut cependant être utilisée dans n'importe quel intercepteur de messages, y compris ceux qui ne font pas partie d'un contexte.

Pour cela il faut d'abord définir une classe implémentant l'interface `System.Runtime.Remoting.Messaging.ILogicalThreadAffinative` pour décrire les informations à passer. Dans le code de l'intercepteur de messages s'exécutant dans le contexte appelant (dans une région « envoi » ou « client ») il faut « accrocher » une instance de cette classe au message représentant l'appel. Dans le code de l'intercepteur de messages s'exécutant dans le contexte cible (dans une région « objet » ou « serveur ») il faut « décrocher » du message représentant l'appel, l'instance de cette classe. Ces opérations s'effectuent en utilisant la propriété `IMethodMessage.LogicalCallContext`.

Voici du code montrant comment implémenter cette technique.

*Exemple 22-36 :*

```
...
public class DonneDeContexte : ILogicalThreadAffinative{
   public int Donnée ;
   public DonneDeContexte(int i){Donnée=i;}
}

[Serializable]
public class MyEnvoiMessageSink : IMessageSink{
   public IMessage SyncProcessMessage(IMessage msg){
       DonneDeContexte dc = new DonneDeContexte(691);
      // 'accroche' la donnée au message représentant l'appel.
      ((IMethodMessage)Msg).LogicalCallContext.SetData("UneDC",dc);
       return m_NextSink.SyncProcessMessage(msg) ;
}...}

[Serializable]
public class MyServeurMessageSink : IMessageSink{
   public IMessage SyncProcessMessage(IMessage msg){
      // 'décroche' la donnée du message représentant l'appel.
      DonneDeContexte dc = (DonneDeContexte)
         ((IMethodCallMessage)Msg).LogicalCallContext.GetData("UneDC");
      Console.WriteLine("   DonneDeContexte:"+dc.Donnée) ;
      IMessage retMsg = m_NextSink.SyncProcessMessage(msg) ;
      return retMsg ;
}...}...
```

On a vu dans la section précédente que l'injection d'intercepteurs de messages dans une région « client » ou « envoi » se fait après l'appel au constructeur d'un objet. Un intercepteur de messages dans une région « client » ou « envoi » ne peut donc pas « accrocher » d'information à un message représentant l'appel au constructeur. Or, un intercepteur de messages dans une région « serveur » peut potentiellement chercher à « décrocher » une information d'un message représentant l'appel au constructeur. Dans ce cas vous pouvez accrocher l'information dans la méthode `GetPropertiesForNewContext()` de l'attribut du contexte :

*Exemple 22-37 :*

```
...
public class MyAfficheContextAttribute : Attribute, IContextAttribute{
   public void GetPropertiesForNewContext(IConstructionCallMessage ctor){

      DonneDeContexte dc = new DonneDeContexte(10);
      ctor.LogicalCallContext.SetData("UneDC",dc);
      IContextProperty prop = new MyAfficheContextProperty(m_bAffiche) ;
      ctor.ContextProperties.Add(prop) ;
}...}
```

# *Récapitulatif*

## *Les modes d'activations d'un objet*

Nous avons vu quatre modes d'activation d'un objet, WKO mode d'appel simple, WKO mode d'appel singleton, CAO et publication d'un objet. Voici un tableau récapitulatif des principales différences de ces modes :

| | **WKO mode simple appel** | **WKO mode singleton** | **CAO** | **Publication** |
|---|---|---|---|---|
| Quand l'objet est-il activé ? | À chaque appel. | Au premier appel d'un client. | Lors de l'appel du constructeur par le client. | Lors de l'appel du constructeur par le serveur. |
| Quelle information détient le client pour accéder à l'objet ? | Un URI contenant le point terminal. | Un URI contenant le point terminal. | Un ObjRef obtenu implicitement lors de l'appel au constructeur. | Un ObjRef obtenu explicitement. |
| Un objet est-il partageable entre plusieurs clients ? | Non | Oui | Non | Oui |
| Le client a-t-il une instance de ObjRef sur l'objet ? | Non | Non | Oui | Oui |

| | | | | |
|---|---|---|---|---|
| Si l'objet est détruit par l'administrateur de baux, un appel retourne- t-il une exception ? | Non | Non | Oui, il n'y a pas de reconstruction automatique d'un objet. | Oui, il n'y a pas de reconstruction automatique d'un objet. |
| Le client est-il obligé de connaître les méta- données de la classe ? | Non, le client peut se satisfaire des métadonnées d'une interface | Non, le client peut se satisfaire des métadonnées d'une interface | Oui, à moins d'utiliser le *design pattern factory.* | Oui |
| Peut-on configurer ce mode dans un fichier de configuration ? | Oui | Oui | Oui | Non |
| Peut-on utiliser un constructeur avec arguments pour l'activation de l'objet ? | Non | Non | Ca dépend. Oui si on utilise le *design pattern factory*. | Oui |

## *Les niveaux d'interception d'un appel*

Nous avons consacré la moitié de ce chapitre à exposer comment les messages représentant les appels, peuvent être interceptés, et pour quelles raisons. La Figure 22-10 présente un bref récapitulatif des niveaux d'interceptions possibles :

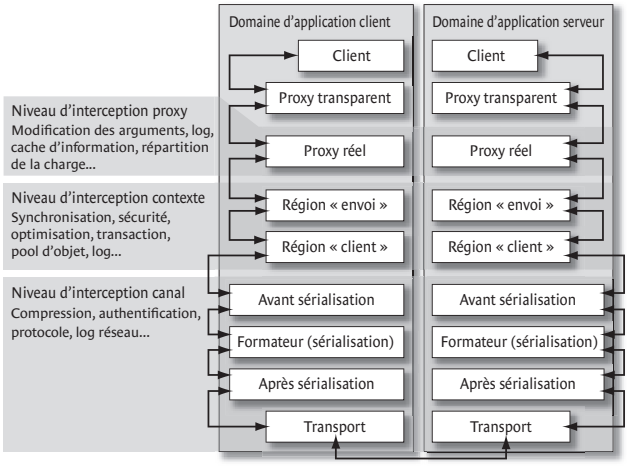


*Figure 22-10 : Les niveaux d'interception d'un appel*

# 23

# *ASP.NET 2.0*

## *Introduction*

Une application web est une application qui renvoie des pages rédigées avec le langage HTML, en réponse à des requêtes HTTP. Ces pages HTML sont exploitables par le client à partir d'une application appelée *navigateur* (*browser*, tel que Internet Explorer). Les navigateurs sont spécialisés dans la visualisation des pages HTML. En 1995, la société *Netscape* fournit le premier navigateur exploitable par le grand public. Ce type de logiciel a rapidement bouleversé le quotidien de centaines de millions de personnes.

Les avantages des applications web sur les applications graphiques encapsulées dans un exécutable (les clients riches) sont nombreux :

- Le déploiement de nouvelles versions de l'application est automatique puisque tout est centralisé du côté serveur.
- Les navigateurs prennent en compte une grande partie de la complexité de la gestion des contrôles.
- La réalisation d'applications distribuées est simplifiée. Les outils de développement sont de plus en plus puissants et les protocoles sous-jacents du web (HTML, HTTP etc) sont bien connus et très largement adoptés.

La principale différence est qu'une application graphique produit des bitmaps affichés sur l'écran à raison de plusieurs dizaines de rafraîchissements par seconde alors qu'une application web produit des pages HTML renvoyées aux clients à la demande. Grâce à l'architecture .NET qui permet et encourage une programmation de haut niveau basé sur les composants et les objets, le développement d'applications graphiques web et le développement d'applications graphiques riches n'ont jamais étaient aussi proche. On utilise parfois le terme de « modèle de programmation unifié ».

## Historique

Au cours des années 90, le développement d'applications web s'est simplifié. Durant la même période les applications web étaient de plus en plus interactives grâce à l'introduction de contrôles. Les informations saisies par l'utilisateur sont incluses dans les requêtes HTTP. HTTP 1.1 présente sept méthodes pour invoquer une ressource. Les deux méthodes les plus usitées sont HTTP-GET et HTTP-POST. La méthode HTTP-GET permet de passer les paramètres d'une requête en ajoutant des informations à l'URL. La méthode HTTP-POST permet de passer les paramètres d'une requête dans le corps de cette requête. Dans ce cas, les paramètres ne sont pas visibles dans l'URL.

Les pages HTML renvoyées en réponses aux requêtes HTTP sont de moins en moins statiques. Elles sont fabriquées dynamiquement en fonction des informations saisies par l'utilisateur. Cette fabrication est réalisée par du code appelé par le serveur HTTP.

Au début des années 90, on utilisait des programmes *CGI* (*Common Gateway Interface*) pour construire dynamiquement des pages HTML. Avec de tels programmes, vous pouviez accéder à des bases de données et réaliser des traitements. Cependant le développement de programmes CGI était fastidieux et souvent les performances n'étaient pas au rendez-vous. Déjà, des techniques d'utilisation de scripts rédigés en langages interprétés tels que *Perl* émergeaient.

En 1996, *Microsoft* intègre les filtres ISAPI à son serveur d'application *IIS* (*Internet Information Server*) pour faciliter la redirection des flux HTTP. Mais surtout, la même année, l'entreprise de Redmond crée les *ASP* (*Active Serveur Page*). Une nouveauté apportée par ASP était que le code était inséré dans l'HTML alors qu'en CGI, c'est l'HTML qui était inséré dans le code. De nombreuses améliorations sont apportées à cette technologie de la version 1 en 1996 à la version 3 en 2000. Malgré son succès les principaux concurrents, JSP et PHP, continuaient à gagner des parts de marché.

## ASP vs. ASP.NET

Aussi, lors de la conception de la plateforme .NET, *Microsoft* a décidé de fournir une technologie nommée ASP.NET pour succéder à ASP. Les applications ASP.NET peuvent être rédigées dans n'importe quel langage .NET tel que C# ou VB.NET. Contrairement aux applications ASP, elles présentent la particularité d'être compilées en langage IL et exécutées par le CLR. Il y a donc un gain de performance conséquent puisqu'il n'y a plus d'interprétation du code.

ASP.NET a été conçu avec le souci de résoudre les problèmes majeurs de la technologie ASP tels que :

- En ASP.NET le code HTML et le code de fabrication dynamique des pages HTML ont la possibilité d'être stockés dans des fichiers différents pour permettre une meilleure collaboration entre les *designers* du site web et les développeurs.
- Les sessions d'ASP.NET peuvent être partagées entre plusieurs machines, par exemple, en les stockant dans une base de données.
- ASP.NET présente plusieurs méthodes pour garder à jour l'état des contrôles d'une page, d'un chargement à l'autre. Cela fait autant de code fastidieux en moins à écrire et à maintenir.
- Les paramètres de configuration d'une application ASP.NET sont stockés dans un fichier au format XML. Pour répliquer une configuration, il suffit de copier un tel fichier. Cela résout la difficulté inhérente à l'utilisation des méta bases IIS.

## *ASP.NET 1.x vs. ASP.NET 2.0*

Nous en sommes maintenant à la version 2.0 d'ASP.NET. Cette seconde version présente de très nombreuses améliorations sans bouleverser les acquis des développeurs ASP.NET 1.x. ASP.NET 2.0 fournit de nombreux composants *prêts à l'emploi*. Ces composants adressent la plupart des tâches récurrentes qui nécessitent des efforts de la part des développeurs pour être mises en œuvre avec ASP.NET 1.x. Aussi, *Microsoft* va jusqu'à dire qu'il y a 70% de code en moins dans les scénarios les plus favorables. Malgré des APIs plus étoffées et donc plus d'informations à assimiler de la part des développeurs, ces derniers gagnent en productivité du fait qu'ils n'ont pas à « réinventer la roue » pour la plupart des tâches classiques.

Vous n'avez pas besoin d'avoir déjà développé avec ASP.NET 1.x pour aborder ce chapitre. Si vous souhaitez ne vous intéresser qu'aux nouveautés ASP.NET 2.0, sachez qu'elles sont listées en annexe page 1029, avec pour chacune, une référence vers la page où elle est exposée.

# *Architecture générale d'ASP.NET*

## *Notion de Web Forms*

Une *web form* est une page d'une application web. Durant la phase de développement, le code source d'une web form est contenu dans un fichier texte d'extension `.aspx` auquel s'ajoute éventuellement un fichier source .NET (C# ou VB.NET par exemple) voire aussi un fichier ressource. À l'exécution, une web form est une classe .NET dérivée de la classe `System.Web.UI.Page`. Une instance de cette classe est créée pour chaque requête de la page concernée. Une telle instance est responsable de la création d'une page HTML. ASP.NET collabore avec un serveur web sous-jacent pour faire en sorte que cette page HTML soit retournée à l'utilisateur responsable de la requête.

## *ASP.NET, IIS et l'exécution des applications web*

Une *application web* est l'ensemble des web forms et des assemblages correspondant, situés dans l'arborescence d'un répertoire virtuel. À l'exécution d'une ou plusieurs applications web sur une machine, il y a deux processus sur la machine :

- Le processus `INETINFO.EXE` représente le processus IIS. Toutes les requêtes HTTP destinées aux applications web et aux services web sont réceptionnées dans ce processus par le code du filtre ISAPI `aspnet_isapi.dll`. Les règles de sécurité IIS sont appliquées puis les requêtes sont transmises au processus `aspnet_wp.exe` par l'intermédiaire d'un pipe nommé. Si à la réception d'une requête le processus `aspnet_wp.exe` n'existe pas, le filtre ISAPI `aspnet_isapi.dll` s'occupe de le démarrer et de créer un pipe nommé dédié.
- Le processus `aspnet_wp.exe` (`wp` pour *Worker Process*) contient un domaine d'application par application web ou par service web tournant sur la machine. Quel que soit le nombre d'applications web ou de services web s'exécutant sur la machine, ils seront tous exécutés dans ce processus, à moins d'une configuration multi processeurs (présentées dans les prochaines pages). L'isolation entre applications web résulte ainsi de l'isolation garantie par le CLR entre les domaines d'applications.

Une autre architecture est mise en œuvre avec IIS 6.0 sous *Windows Server 2003*. Dans ce cas IIS et ASP.NET s'exécute au sein d'un même processus nommé `w3wp.exe`.

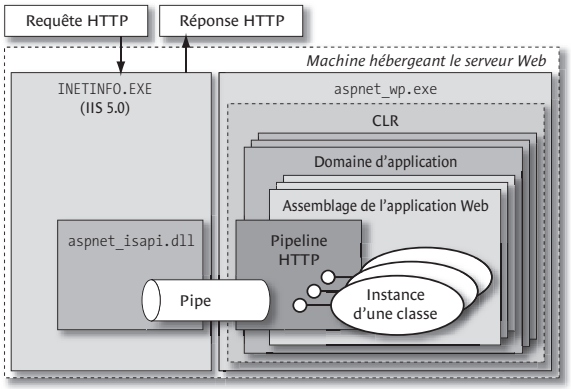


*Figure 23-1 : Interaction des processus de IIS et de ASP.NET*

ASP.NET a été conçu pour être indépendant du serveur web sous-jacent. Concrètement, cela implique qu'ASP.NET peut s'interfacer avec d'autres serveurs web qu'IIS. Par exemple, *Visual Studio .NET 2005* est fourni avec le serveur web `WebDev.WebServer.EXE` permettant de tester et de déboguer vos applications web en cours de développement. Ce serveur web est basé sur l'ancien serveur web nommé *Cassini* qui était fourni gratuitement par *Microsoft*. Ceci est très pratique puisque IIS n'est disponible que sur les versions professionnelles de *Windows*. Il est donc maintenant possible de développer une application web sur une machine n'ayant qu'une édition familiale de *Windows*. Notez que vous pouvez configurer *Visual Studio 2005* pour qu'il s'interface avec IIS en modifiant l'option *Propriété du projets web* ► *Options de démarrage* ► *Utiliser un serveur de votre choix.*

ASP.NET effectue un certain nombre de tâches pour traiter les requêtes et les réponses HTTP dans ce que nous appelons *pipeline HTTP* dans la Figure 23-1. Parmi ces tâches nous pouvons citer l'authentification ASP.NET de l'utilisateur qui a initié une requête ou la gestion des informations concernant la session courante pour cet utilisateur. Tout au long de ce chapitre, nous aurons l'occasion de décrire ces tâches. Nous expliquerons aussi comment vous pouvez étendre ce pipeline en injectant vos propres traitements à certains points précis.

L'exécution d'une requête au sein du processus `aspnet_wp.exe` se fait sur le même thread de bout en bout. Pour cela, ASP.NET exploite les threads I/O du pool de threads du CLR et peut donc exécuter plusieurs requêtes simultanément. En plus d'éviter *by-design* un certains nombre de problèmes de concurrence cette façon de faire est efficace car la réception d'une requête à partir du pipe nommé débute directement sur un thread I/O. On évite donc une transition d'information entre threads. Puisqu'il n'y a pas de pipe nommé avec IIS 6.0 les choses se passent différemment et dans cette version, ASP.NET utilise les threads ouvriers du CLR.

Il est très important de remarquer que le processus `aspnet_wp.exe` s'exécute par défaut sous un compte utilisateur *Windows* nommé *ASPNET* (ou *Network Service* sous IIS 6.0) aux privilèges restreints. En effet, durant le développement il se peut que vous ne vous rendiez pas compte de certains problèmes du fait que `WebDev.WebServer.EXE` s'exécute par défaut sous votre compte utilisateur. Nous verrons comment éventuellement configurer ASP.NET pour que ce processus s'exécute sous un autre utilisateur. Notez enfin que si l'utilisateur initiateur d'une requête a pu

être authentifié comme un utilisateur *Windows*, il est possible que le thread qui traite la requête s'exécute sous le compte de cet utilisateur.

Vous pouvez héberger simultanément plusieurs applications web avec ASP.NET. Par exemple un serveur IIS avec les adresses IP mappées suivantes et les sites logiques suivants, peut exposer les applications web suivantes :

```
Adresses IP mappées :
   www.xyz.com
   www.smacchia.com

Sites logiques :
   http://www.xyz.com
   http://www.smacchia.com

Applications web :
   http://www.xyz.com/holidays        Saisie des jours de congés
   http://www.xyz.com/holidays/cadre  Saisie des jours de congés des cadres
   http://www.smacchia.com/Documents  Publication de documents
   http://www.smacchia.com            Présentation d'activité
```

Les répertoires physiques contenant les applications web n'ont pas de contraintes quant à leur localisation. Ils peuvent être sur le système de gestion de fichiers local ou sur un autre système distant. Par exemple le répertoire de l'application web `http://www.xyz.com/holidays/cadre` n'est pas tenu d'être physiquement imbriqué dans le répertoire de l'application web `http://www.xyz.com/holidays`.

Enfin, il est possible de faire cohabiter une version 1.x et une version 2.0 d'ASP.NET sur la même machine. Dans ce cas, vous devez préciser au filtre ISAPI `aspnet_isapi.dll` quelle version utiliser en vous servant de l'outil `aspnet_regiis.exe`. Il suffit de taper la ligne de commande suivante avec la version souhaitée de cet outil :

```
aspnet_regiis.exe -r
```

## *Héberger ASP.NET dans vos applications .NET*

L'espace de nom `System.Web.Hosting` présente des classes permettant d'héberger ASP.NET dans une application .NET quelconque. Cette possibilité est illustrée par l'exemple suivant :

*Exemple 23-1 :* *AspnetHosting.cs*

```
using System ;
using System.Web ;
using System.Web.Hosting ;
class Program {
   static void Main() {
      Console.WriteLine("Main Appdomain:" +
                        AppDomain.CurrentDomain.FriendlyName) ;
      CustomSimpleHost host = (CustomSimpleHost)
            ApplicationHost.CreateApplicationHost(
               typeof(CustomSimpleHost), @"/",
               System.IO.Directory.GetCurrentDirectory());
```

```
      // Passe en argument la page web et les paramètres de la requête.
      host.ProcessRequest("Default.aspx", string.Empty) ;
   }
}
public class CustomSimpleHost : MarshalByRefObject {
   public void ProcessRequest(string file, string query) {
      Console.WriteLine("ASP.NET AppDomain:" +
                        AppDomain.CurrentDomain.FriendlyName) ;
      SimpleWorkerRequest aspnetWorker =
               new SimpleWorkerRequest(file, query, Console.Out) ;
      HttpRuntime.ProcessRequest(aspnetWorker);
   }
}
```

On déclare d'abord vouloir héberger ASP.NET dans le processus courant grâce à l'appel de la méthode statique `ApplicationHost.CreateApplicationHost()`. Cette méthode crée un nouveau domaine d'application et y charge ASP.NET. ASP.NET y charge alors l'assemblage contenant la classe `CustomSimpleHost`. En l'occurrence cet assemblage est celui de notre application, à savoir `AspnetHosting.exe`. Une instance de cette classe est ensuite créée dans ce nouveau domaine d'application. La méthode `CreateApplicationHost()` retourne une référence vers cette instance dans le domaine d'application initial. Ceci est possible car la classe `CustomSimpleHost` dérive de la classe `MarshalByRefObject`. Ainsi grâce à la technologie *.NET Remoting*, une instance de `CustomSimpleHost` peut être référencée et utilisée à partir d'un domaine d'application différent de celui dans lequel elle réside.

Le code de la méthode `ProcessRequest` se sert d'une instance de la classe `System.Web.Hosting.SimpleWorkerProcess` pour traiter une requête HTTP GET. Ici, nous créons artificiellement cette requête en fournissant le nom de la web form demandée (`Default.aspx`), les paramètres de la requête GET (ici il n'y en a pas) et un flot de données vers lequel ASP.NET va pouvoir diriger la réponse HTTP (en l'occurrence ce flot est la console). La page HTML produite par ASP.NET à partir de la web form `Default.aspx` est affichée sur la console. Ce programme affiche donc ceci :

```
Main Appdomain:ConsoleApplication6.exe
ASP.NET AppDomain:c6ed2272-1-127654065004140000

<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <center>
      <form name="Form1" method="post" action="Default.aspx" id="Form1">
...
```

Pour exécuter cet exemple il faut que vous prévoyiez de stocker une web form nommée `Default.aspx` dans le même répertoire que l'assemblage `AspnetHosting.exe`. Vous pouvez par exemple vous servir de la page `Default.apsx` présentée un peu plus loin dans l'Exemple 23-3. En outre il faut prévoir de dupliquer l'assemblage `AspnetHosting.exe` dans le sous répertoire `/bin` du répertoire contenant la version originale de `AspnetHosting.exe`. En effet, ASP.NET va automatiquement rechercher dans ce répertoire l'assemblage contenant la classe `CustomSimpleHost` pour le charger dans le nouveau domaine d'application.

## *Utiliser conjointement ASP.NET avec la couche HTTP.SYS*

Dans la section précédente nous avons bien précisé que nous simulions la requête GET transmise à ASP.NET. En utilisant un serveur web basé sur les services de la couche HTTP.SYS il est aisé de construire un serveur web complet hébergeant ASP.NET en se passant complètement d'IIS. Cette couche HTTP.SYS fait l'objet de la section page 654. Nous y exposons notamment des exemples de programmes .NET qui exploitent cette couche grâce aux services de la classe `HttpListener`. Le programme suivant utilise conjointement les services d'hébergement d'ASP.NET et de HTTP.SYS :

*Exemple 23-2 :*

```
using System ;
using System.Web ;
using System.Web.Hosting ;
using System.IO ;
using System.Net ;
class Program {
   static void Main() {
      CustomSimpleHost host = (CustomSimpleHost)
            ApplicationHost.CreateApplicationHost(
               typeof(CustomSimpleHost), @"/",
               System.IO.Directory.GetCurrentDirectory()) ;
      HttpListener httpListener = new HttpListener() ;
      string uri = string.Format("http://localhost:8008/") ;
      httpListener.Prefixes.Add(uri) ;
      httpListener.Start() ;
      Console.WriteLine("En attente sur " + uri) ;
      while (true) {
         HttpListenerContext ctx = httpListener.GetContext();
         string page = ctx.Request.Url.LocalPath.Replace("/", "");
         string query = ctx.Request.Url.Query;
         Console.WriteLine("Requête recue : {0}?{1}",page, query) ;
         StreamWriter writer = new
                 StreamWriter( ctx.Response.OutputStream,
                               System.Text.Encoding.Unicode) ;
         host.ProcessRequest(page, query, writer);
         writer.Flush() ;
         writer.Close() ;
         ctx.Response.Close() ;
      }
   }
}
public class CustomSimpleHost : MarshalByRefObject {
   public void ProcessRequest(string file, string query,
                              TextWriter writer) {
      SimpleWorkerRequest aspnetWorker =
               new SimpleWorkerRequest(file, query, writer) ;
      HttpRuntime.ProcessRequest(aspnetWorker) ;
```

```
    }
}
```

Pour tester ce serveur web, il suffit de demander l'URL suivante dans un navigateur s'exécutant sur la même machine :

```
http://localhost:8008/Default.aspx
```

# Stockage du code source

## Code-inline

Chaque page a un seul fichier d'extension `.aspx` et un tel fichier ne correspond qu'à une seule page. Voici le code d'une page web :

*Exemple 23-3 :* *`Default.aspx`*

```
<%@ Page Language="C#" Debug="false" Description="Ma première page !" %>
<script language=C# runat="server">
   void Btn_Click(Object sender, EventArgs e) {
      Msg.Text = "Vous avez sélectionné : "+Couleur.SelectedItem.Value;
   }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <center>
      <form id="Form1" action="Default.aspx" method="post" runat="server">
         Couleur : <asp:dropdownlist id="Couleur" runat="server">
                     <asp:listitem>blanc</asp:listitem>
                     <asp:listitem>noir</asp:listitem>
                   </asp:dropdownlist>
         <br/>
         <asp:button id="Button1" text="Soumettre" OnClick="Btn_Click"
          runat="server"/>
         <br/>
         <asp:label id="Msg" runat="server"/>
      </form>
      </center>
   </body>
</html>
```

Ce fichier commence par des directives qui peuvent fournir de multiples renseignements à ASP.NET tels que le langage .NET utilisé pour coder la page, des options de compilation, des options de débogage etc.

Hormis ces directives, le reste du contenu ressemble à du code HTML à ceci prés qu'il contient quelques balises non HTML telles que `<script>`, `<form>` ou `<asp:button>` qui sont reconnues par ASP.NET. Une balise `<script>` contient du code .NET écrit par exemple en C# ou en VB.NET. On parle alors de *code-inline*. ASP.NET fera en sorte que ce code fasse partie de la classe représentant la page à l'exécution.

Les autres balises non HTML contiennent des descriptions de contrôles ASP.NET. Pour l'instant, retenez qu'un contrôle ASP.NET est un champ d'instance de la classe de la web form qui à l'exécution, remplace sa description dans le fichier `.aspx` par du code HTML.

Notons que pour une même page `.apsx` *Visual Studio* fournit deux vues. La vue design plutôt utilisée par les graphistes et la vue qui présente le contenu du fichier `.aspx`. Comprenez bien qu'il ne s'agit que de vues des mêmes données et que tous changements à partir d'une vue est immédiatement répercutés sur l'autre vue :

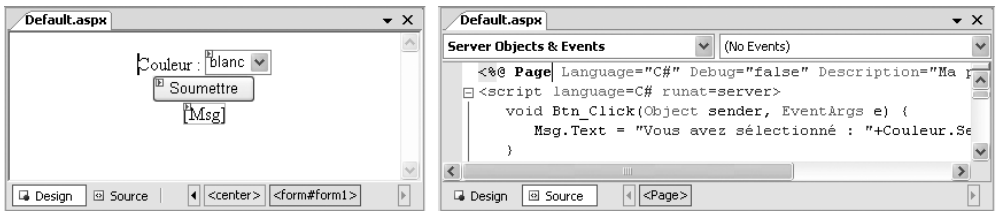


*Figure 23-2 : Vue design et vue source*

## *Bloc de restitution de code*

À l'instar d'ASP, en ASP.NET vous avez la possibilité d'écrire du code dans des *blocs de restitution de code* délimités par des balises <% et %>. Ce code doit être rédigé dans le langage .NET de la page, précisé par l'attribut `Language`. À l'exécution de la page, ASP.NET remplace un tel bloc par la chaîne de caractères produit par ce code. Par exemple, cette page `.aspx` produit le code HTML suivant :

*Exemple 23-4 :* *`Default.aspx`*

```
<%@ Page Language="C#" Debug="true" Description="Ma deuxième page !" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <form id="Form1" action="Default.aspx" method="post" runat="server">
         <% for(int i=4 ; i<8 ; i++) { %>
            <font size="<%=i%>"> Bonjour </font>
         <% } %>
      </form>
   </body>
</html>
```

Page HTML produite

```
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <form method="post" action="Default.aspx" id="Form1">
...
         <font size="4"> Bonjour </font>
         <font size="5"> Bonjour </font>
         <font size="6"> Bonjour </font>
         <font size="7"> Bonjour </font>
```

```
      </form>
   </body>
</html>
```

Si vous exécutez cet exemple, vous vous apercevrez qu'il y a du code HTML en plus là où nous avons placé des points de suspension. Il est pour l'instant trop tôt dans ce chapitre pour s'intéresser à ce code. Retenez juste pour l'instant que les blocs de restitutions de code ont été exécutés et ont produits les quatre lignes en gras dans la page HTML.

Il est intéressant de noter que le code compris dans les blocs de restitutions de code a été compilé en une méthode de la classe qui représente la page :

```
...
private void __RenderForm1( HtmlTextWriter __w,
                            Control parameterContainer) {
   __w.Write("\r\n           ");
   for (int num1 = 4 ; num1 < 8 ; num1++) {
      __w.Write("\r\n              <font size=\"");
      __w.Write(num1);
      __w.Write("\"> Bonjour </font> \r\n           ");
   }
}
...
```

Cette méthode est automatiquement appelée par ASP.NET au moment où la page est créée.

## *Code-Behind*

Si vous créez un nouveau site web C# avec *Visual Studio* (avec *Fichier* ▶ *Nouveau Site Web*...) vous vous apercevrez que lorsque vous créez une nouvelle page, vous pouvez cocher l'option *Placez le code dans un fichier séparé*. Dans ce cas, un fichier source C# nommé `[page].aspx.cs` sera associé à votre nouvelle page `[page].aspx`. Vous avez la possibilité de définir une classe dans ce fichier qui contiendra des membres exploitables directement à l'exécution dans la classe représentant la page. On nomme ce code *code-behind* de la page. Clairement, les avantages du *code-behind* sur le *code-inline* et les blocs de restitution de code sont d'alléger les fichiers `.aspx` et de permettre aux designers web et aux développeurs de collaborer plus efficacement puisque ces premiers travaillent avec des fichiers `.aspx` tandis que ces derniers travaillent avec des fichiers `.aspx.cs`.

Pour utilisez cette technique avec notre méthode `Btn_Click()`, il faut d'abord préciser dans le fichier `Default.aspx` quelle classe contient le *code-behind* avec la directive `Inherits` et quel fichier source contient cette classe avec la directive `CodeFile` :

*Exemple 23-5 :* *`Default.aspx`*

```
<%@ Page Language="C#"  Inherits="MyDefaultPage"
         CodeFile="Default.aspx.cs" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <center>
      <form id="Form1" action="Default.aspx" method="post" runat="server">
         Couleur : <asp:dropdownlist id="Couleur" runat="server">
                   <asp:listitem>blanc</asp:listitem>
```

```
                    <asp:listitem>noir</asp:listitem>
                  </asp:dropdownlist>
         <br/>

         <asp:button id="Button1" text="Soumettre" OnClick="Btn_Click"
          runat="server"/>
         <br/>
         <asp:label id="Msg" runat="server"/>
      </form>
      </center>
   </body>
</html>
```

Ensuite il faut déclarer la classe `MyDefaultPage` comme partielle avec le mot clé C♯2 `partial` :

*Exemple 23-6 :* *`Default.aspx.cs`*

```
using System ;
public partial class MyDefaultPage : System.Web.UI.Page {
   protected void Btn_Click(Object sender, EventArgs e) {
      Msg.Text = "Vous avez sélectionné : "+Couleur.SelectedItem.Value ;
   }
}
```

En effet, ASP.NET 2.0 présente un modèle de construction du *code-behind* différent de celui d'ASP.NET 1.x. Ces deux modèles sont illustrés par la figure suivante :

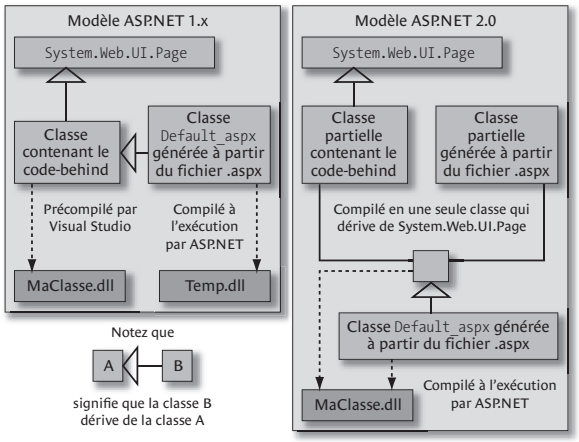


*Figure 23-3 : Modèles 1.x et 2.0 de construction des classes*

On voit qu'en ASP.NET 2.0, la classe de base contenant le *code-behind* a la particularité d'être la réunion de deux classes partielles : notre classe partielle `MyDefaultPage` et une classe partielle générée à partir du fichier `.aspx`. Ce modèle présente plusieurs avantages :

- Il allège la définition de la classe contenant le *code-behind* de nombreux détails tels que la définition des contrôles. Ces détails étaient anciennement générés par *Visual Studio*. Ils

sont maintenant invisibles pour le développeur puisqu'ils sont ajoutés automatiquement et implicitement par ASP.NET à la compilation (notez cependant que le mécanisme d'intellisense de *Visual Studio* tient compte de ces détails).

- Il diminue le risque d'erreur de synchronisation puisque les détails générés ne sont plus accessibles aux développeurs.
- Il facilite la migration des projets ASP.NET 1.x en préservant le fait que la classe `[pageName]_aspx` dérive de la classe contenant le *code-behind* qui elle-même dérive de la classe `Page`.
- Une même classe de base partielle peut être exploitée peut être réutilisée par plusieurs pages.

Par curiosité, voici à quoi ressemblent la classe partielle et la classe `Default_aspx` générées dans notre exemple par ASP.NET :

```
public partial class MyDefaultPage : System.Web.UI.Page,
        System.Web.SessionState.IRequiresSessionState {
   protected System.Web.HttpApplication ApplicationInstance {
      get{ return return this.Context.ApplicationInstance ; }
   }
   protected System.Web.Profile.DefaultProfile Profile {
      get { return (DefaultProfile)this.Context.Profile ; }
   }
   protected System.Web.UI.WebControls.Button Button1 ;
   protected System.Web.UI.WebControls.DropDownList Couleur ;
   protected System.Web.UI.HtmlControls.HtmlForm Form1 ;
   protected System.Web.UI.WebControls.Label Msg ;
}
namespace ASP {
   public class Default_aspx : MyDefaultPage {
      public Default_aspx() {
         base.AppRelativeVirtualPath = "~/Default.aspx" ;
         if (!Default_aspx.__initialized) {
            string[] textArray1 = new string[2]
               { "~/Default.aspx", "~/Default.aspx.cs" } ;
            Default_aspx.__fileDependencies =
               base.GetWrappedFileDependencies(textArray1) ;
            Default_aspx.__initialized = true ;
         }
      }
      private void __BuildControl__control2(ListItemCollection __ctrl) {
         ListItem item1 = this.__BuildControl__control3() ;
         __ctrl.Add(item1) ;
         ListItem item2 = this.__BuildControl__control4() ;
         __ctrl.Add(item2) ;
      }
      private ListItem __BuildControl__control3() { ... }
      private ListItem __BuildControl__control4() { ... }
      private Button __BuildControlButton1() { ... }
```

```
        private DropDownList __BuildControlCouleur() { ... }
        private HtmlForm __BuildControlForm1() { ... }
        private Label __BuildControlMsg() { ... }
        private void __BuildControlTree(Default_aspx __ctrl) { ... }
        protected override void FrameworkInitialize() {
           base.FrameworkInitialize() ;
           this.__BuildControlTree(this) ;
           base.AddWrappedFileDependencies(Default_aspx.__fileDependencies) ;
           base.Request.ValidateInput() ;
        }
        public override int GetTypeHashCode() {return 0x36b54869;}
        private static object __fileDependencies ;
        private static bool __initialized ;
     }
  }
```

Si vous utilisez la directive `Src` à la place de la directive `CodeFile`, ou si vous ne précisez ni l'une ni l'autre mais que vous déployez l'assemblage contenant la classe contenant le *code-behind* dans le répertoire `bin`, vous revenez au modèle ASP.NET 1.x et il vous incombe d'ajouter les détails tels que la définition des contrôles à votre classe de base. Enfin, sachez que la directive ASP.NET 1.x `CodeBehind` n'est plus utilisable.

# *Modèles de compilation et de déploiement*

ASP.NET 2.0 présente trois modèles de compilation : la compilation dynamique (parfois nommé *full runtime compilation* en anglais), la pré compilation sur place (*in-place pre-compilation* en anglais) et la pré compilation pour déploiement. (*deployment pre-compilation* en anglais).

## *Compilation dynamique*

Dans ce modèle, vous déployez les fichiers sources sur le serveur et ASP.NET s'occupe de les compiler à l'exécution. Chaque fichier source est compilé lors de sa première utilisation. Si un fichier source déjà compilé est modifié, ASP.NET le détecte et le recompile. L'avantage principal de ce modèle est de faciliter le travail des développeurs puisqu'ils n'ont pas à se soucier des mises à jours, du déploiement et de l'étape de la compilation. En revanche, ce modèle dégrade légèrement les performances de votre serveur puisque chaque compilation a un coût.

La notion de compilation dynamique existait déjà en ASP.NET 1.x puisque comme le montre la Figure 23-3, chaque page `aspx` était compilée à l'exécution lors de la première demande. En revanche, en ASP.NET 1.x vous deviez compiler manuellement (ou avec *Visual Studio*) les classes contenant le *code-behind* ainsi que toutes les autres classes utilisées par votre application. Si vous souhaitiez réutiliser des classes déjà compilées dans des assemblages, il fallait placer ces assemblages dans le répertoire `[AppRootDir]/bin` de votre application (ou dans le GAC) et les référencer à la compilation de votre application.

La compilation dynamique en ASP.NET 2.0 a considérablement évolué. Vous pouvez déployer vos fichiers sources dans le répertoire `[AppRootDir]/App_Code` de votre application (ou dans un de ses sous répertoires). Cette arborescence peut contenir des fichiers sources rédigés éventuellement avec dans plusieurs langages tels que C#, VB.NET, XSD (pour les datasets typés) etc. Les classes contenues dans ces fichiers peuvent être exploitées dans le code de n'importe quelle page

de votre application. Pour améliorer les performances de la compilation, l'élément `<configuration>/<system.web>/<compilation>` du fichier `web.config` présente plusieurs sous éléments permettant de paramétrer finement la quantité de code à compiler à chaque compilation (*batch compilation* en anglais).

Les assemblages contenus dans le répertoire `/bin` n'ont pas besoin d'être référencé d'aucune sorte. Les classes et autres ressources contenues dans ces assemblages peuvent être exploitées dans le code de n'importe quelle page de votre application. ASP.NET 2.0 se charge de les trouver à l'exécution.

En plus des répertoires `/App_Code` et `/bin`, ASP.NET sait aller chercher d'autres types de ressources que des classes dans les répertoires suivants (tous directement placés dans le répertoire racine de votre application) :

- `/App_GlobalResources` et `/App_LocalResources` pour les fichiers ressources globaux ou locaux à l'application web (voir page 953).
- `/App_WebReferences` pour les fichiers WSDL à compiler en proxy.
- `/App_Data` pour les fichiers contenant des données.
- `/App_Browsers` pour les fichiers `.browser` qui déclarent quelles possibilités doit supporter un navigateur (ce répertoire replace l'élément `<browserCaps>` in `machine.config`).
- `/App_Themes` pour les fichiers `.css` et `.skin`.

Toute mise à jour de n'importe quel fichier contenu dans un de ces répertoires sera automatiquement détectée et prise en compte par ASP.NET 2.0 à l'exécution. La mise à jour d'un fichier source provoque sa recompilation tandis que la mise à jour d'un assemblage dans le répertoire `/bin` provoque son rechargement par le CLR. Ce mécanisme de rechargement d'assemblage exploite la possibilité nommée *shadow copy* du CLR qui est décrite en page 108. Enfin, sachez que le filtre ISAPI `aspnet_filer.dll` fait en sorte qu'aucun des fichiers contenus dans ces répertoires n'est accessible par une requête HTTP.

## *Pré compilation sur place*

Ce modèle est similaire au modèle précédant mis à part que vous pouvez déclencher la compilation complète de votre application avec une requête HTTP sur le répertoire racine de votre application suivie de `precompile.axd`. Par exemple :

```
http://localhost/MonSiteWeb/precompile.axd
```

En plus de maîtriser le moment de la compilation, la pré compilation sur place permet de rencontrer les éventuelles erreurs de compilation avant qu'elles ne soient découvertes par les clients.

## *Pré compilation pour déploiement*

Le nouvel outil `aspnet_compiler.exe` permet de compiler complètement une application web. Ainsi, il vous permet de ne déployer que des DLLs sans aucun fichier de code source. En plus des avantages évidents au niveau des performances, ce modèle de compilation et de déploiement est adapté aux projets de grande envergure. En effet, il permet la mise en place de processus évolués de compilation (avec une batterie de tests unitaires ou le recours à la technologie MSI par exemple) et éventuellement le renforcement de la propriété intellectuelle en permettant

l'obfuscation des assemblages. Bien évidemment, ceci se paye par une souplesse amoindrie dans le processus de mise à jour.

L'outil `aspnet_compiler.exe` s'utilise très simplement. Vous précisez en entrée le répertoire virtuel contenant le code source de votre application web avec l'option `/m` (ou le répertoire *Windows* racine avec l'option `/p`), le nom de votre application avec l'option `/v` et le répertoire de sortie contenant le résultat de la compilation.

```
aspnet_compiler.exe /m /LM/W3SVC/1/ROOT/MonSiteWeb D:/TestDeploy
aspnet_compiler.exe /v WebSite /p D:/Site/MonSiteWeb  D:/TestDeploy
```

Il vous suffit ensuite de copier/coller le contenu de ce répertoire de sortie dans le répertoire virtuel adéquat sur le serveur. Il est intéressant de remarquer que ce répertoire de sortie présente toujours vos fichiers d'extensions `.aspx` mais vidés de leur contenu, votre fichier `web.config` ainsi que des assemblages aux noms générés dans le répertoire `/bin`. Vous remarquez aussi la présence d'un fichier `PrecompiledApp.config`. Les éléments XML contenus dans ce fichier indique à ASP.NET si il est autorisé ou pas à compiler des pages `.aspx`. Ainsi, les paramètres de ce fichier peuvent empêcher la prise en compte de nouvelles pages `.aspx` ajoutées à un site.

# *Web forms et contrôles*

## *Contrôle serveur*

Un contrôle serveur est un objet dont la classe dérive de `System.Web.UI.Control`. Tout l'intérêt d'un contrôle serveur réside dans sa capacité à fabriquer un fragment de code HTML à la demande. Pour cela, la classe `Control` présente des méthodes virtuelles telles que la méthode `void Render(HtmlTextControl)`. Les classes qui dérivent de `Control` ont la possibilité de redéfinir une ou plusieurs de ces méthodes. Notez que le fragment de code HTML produit par un contrôle serveur ASP.NET peut correspondre à un ou plusieurs contrôles HTML. Dans la suite, nous préciserons *contrôle HTML* ou *contrôle serveur* lorsqu'il peut y avoir ambiguïté.

Grâce à la propriété `ControlCollection Controls{get;set;}` de la classe `Control`, chaque contrôle serveur a la possibilité de se comporter comme un containeur de contrôle serveur fils. Notamment, la classe `Page` dérive de la classe de `Control`. Une page web est donc un contrôle serveur qui a la particularité de n'être fils d'aucun autre contrôle. En outre, la classe `Control` présente la méthode `void RenderChildren(HtmlTextControl)` qui appelle tour à tour les méthode `Render()` de chaque contrôle fils en respectant l'ordre dans lequel ils sont stockés. Reprenons la page de l'Exemple 23-5 :

```
<%@ Page Language="C#" ... %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <center>
      <form id="Form1" action="Default.aspx" method="post" runat="server">
         Couleur : <asp:dropdownlist id="Couleur" runat="server">
                     <asp:listitem>blanc</asp:listitem>
                     <asp:listitem>noir</asp:listitem>
                   </asp:dropdownlist>
         <br/>
         <asp:button id="Button1" text="Soumettre" OnClick="Btn_Click"
```

```
          runat="server"/>
         <br/>
         <asp:label id="Msg" runat="server"/>
      </form>
      </center>
   </body>
</html>
```

Lorsque ASP.NET construit la classe qui représente cette page, il y insère du code qui construira une arborescence de contrôle serveur. Cette arborescence est illustrée par la figure suivante :

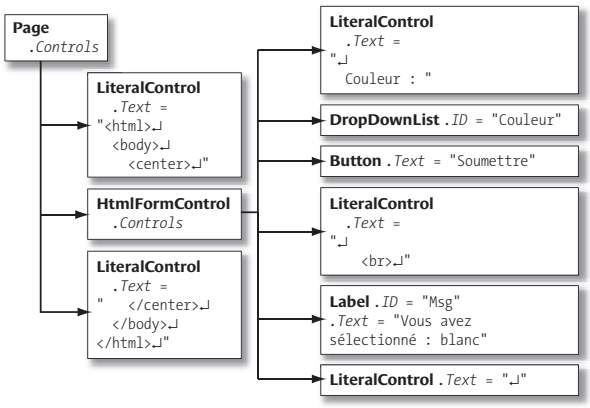


*Figure 23-4 : Arborescence de contrôles serveur*

On voit que les classes de contrôle `Page`, `LiteralControl`, `HtmlFormControl`, `Label`, `DropDownList` et `Button` sont impliquées. Le comportement d'une instance de la classe `LiteralControl` est très simple. Lors de l'appel de sa méthode `Render()` un tel objet ne fait qu'ajouter à la page HTML en construction le texte qu'il contient dans sa propriété `string Text{get;set;}`. Nous aurons dans la suite l'occasion de détailler les comportements plus actifs des autres types de contrôle.

Reprenons le code C# de cette page :

```
using System ;
public partial class MyDefaultPage : System.Web.UI.Page {
   protected void Btn_Click(Object sender, EventArgs e) {
      Msg.Text = "Vous avez sélectionné : "+Couleur.SelectedItem.Value ;
   }
}
```

Nous attirons votre attention sur le fait que l'on se sert des deux champs `Label Msg` et `DropDownList Couleur`. Ces deux champs sont deux références vers les deux objets de type `Label` et `DropDownList` de notre arborescence de contrôles. On remarque que leurs noms sont les mêmes que les chaînes de caractères précisés dans les attributs `id`. Lors de la discussion sur le *code-behind*, nous avons déjà énoncé que la déclaration de ces champs est automatiquement faite par ASP.NET dans la deuxième moitié partielle de notre classe `MyDefaultPage`. Nous

pouvons maintenant préciser qu'ASP.NET génère aussi du code responsable de l'initialisation de ces champs. Ceci souligne l'importance de la compréhension de l'enchaînement des actions lors de la vie d'une page. En effet, si vous essayez d'utiliser les champs `Msg` ou `Couleur` avant qu'ils n'aient été initialisés par ce code, vous obtiendrez une exception de type `NullReferenceException`. Nous détaillons un peu plus tard cet enchaînement crucial mais avant, intéressons nous à la logique de notre page `Default.aspx`.

## *Interaction client/serveur*

La Figure 23-5 illustre la logique de notre page `Default.aspx` en précisant les principales étapes de l'interaction client/serveur web. Le client initie l'interaction en demandant au serveur la page `/WebSite1/Default.aspx` au moyen d'une requête HTTP GET. Le serveur lui renvoie la page HTML demandée. Cette dernière est fabriquée par une nouvelle instance de la classe `Default_aspx`. Lors de la première demande d'un client, les contrôles serveur ont leurs états initiaux. Pour le `DropDownList` l'état initial est blanc et pour le `Label` l'état initial est une chaîne de caractères vide. Une fois la page reçue, le client sélectionne `noir` sur le contrôle HTML correspondant à notre `DropDownList` et clique sur le bouton. Cette dernière action entraîne l'envoi d'une requête HTTP POST au serveur. Cette requête contient notamment l'état du contrôle HTML correspondant à notre `DropDownList`. Le traitement interne des pages `.aspx` de ASP.NET (dans le pipeline HTTP) est capable de détecter que la requête POST est due à un click du bouton `Button1`. Aussi, il invoque la méthode `Btn_Click()` sur une nouvelle instance de la classe `Default_aspx`. En effet, cette méthode a été associée à l'évènement « click sur Button1 » dans notre page `Default.aspx` grâce à la ligne `<...id="Button1"...OnClick="Btn_Click"...>`. L'exécution de cette méthode positionne l'état de notre contrôle `Label` à `"Vous avez sélectionné : noir"`. Le fragment HTML ajouté par ce contrôle est alors `"<span id="Msg">Vous avez sélectionné : noir</span>"`.

Dans l'interaction client/serveur de la section précédente, vous avez peut être remarqué que la sélection `noir` du contrôle `Couleur` a été retenue lorsque le client charge la page pour la seconde fois. Cela résulte du fait qu'ASP.NET initialise automatiquement et implicitement les valeurs des contrôles avec les valeurs trouvées dans une requête POST. Certains types de contrôles HTML comme le contrôle `<span>` (correspondant à un contrôle serveur de type `Label`) n'ont pas leurs valeurs sauvées explicitement dans la requête POST. Cela se voit en analysant la requête POST. Seules les contrôles nommés `Couleur`, `Button1` et `__VIEWSTATE` (dont nous allons parler) ont leurs valeurs sauvées dans le corps de la requête POST :

```
POST /WebSite1/Default.aspx HTTP/1.1
...
Content-Length: 104
__VIEWSTATE=%2FwEPDwUJOTE3ODUwMjE3ZGS%2ByRcRG2v6mOv5xTATxgcXeOGIOA%
3D%3D&Couleur=blanc&Button1=Soumettre
```

Et pourtant, en déboguant la méthode `Btn_Click()` on s'aperçoit lors d'un deuxième click du bouton que le serveur connaît le contenu du contrôle `Msg` de type `Label` :

Il est très important que vous soyez convaincu que dans cet exemple le serveur ne retient aucun état. La valeur du contrôle `Label Msg` est donc forcément encodée d'une manière implicite dans la requête POST.

En effet, lorsque le navigateur client fabrique une requête HTTP, il rassemble dans une chaîne de caractères toutes les valeurs de tous les contrôles dont les valeurs n'ont pas été placées d'une

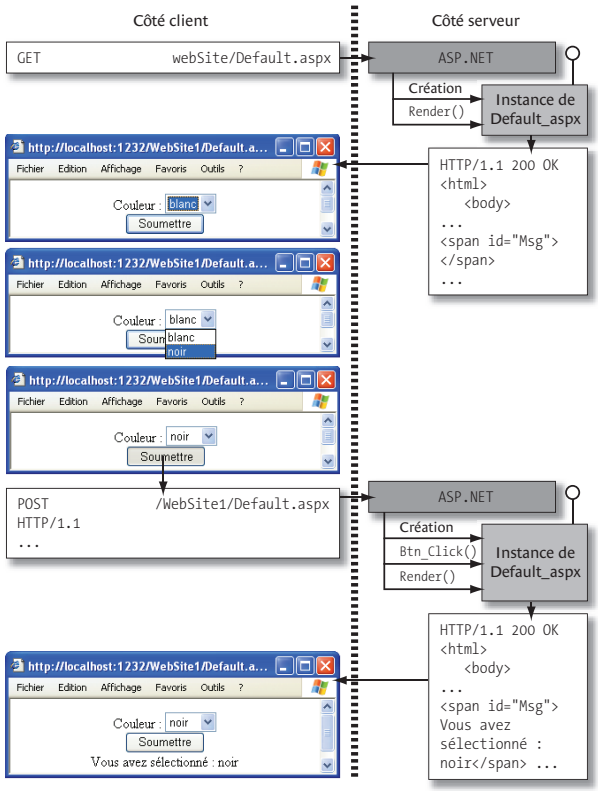


*Figure 23-5 : Interaction client/serveur*

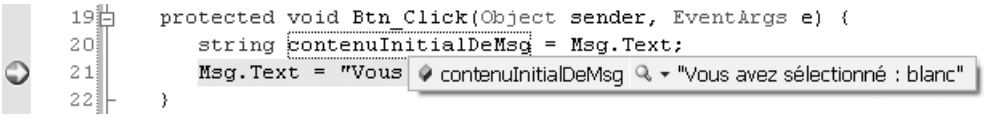


*Figure 23-6 : Débogage pour mettre en évidence le rôle du ViewState*

manière standard dans la requête POST en construction. Cette chaîne est ensuite encodée dans un tableau binaire qui est lui-même encodé dans une chaîne de caractères base64.

Il est maintenant utile de remarquer que toutes les pages HTML produites par une page ASP.NET incluent un contrôle HTML invisible de type `Input` nommé `__VIEWSTATE`. Pour s'en convaincre, voici un extrait d'une telle page HTML produite par une instance de notre classe `Default_aspx` :

```
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <center>
      <form method="post" action="Default.aspx" id="Form1">
```

```
<div>
...
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUJOTE3ODUwMjE3D2QWAgIBD2QWAgIFDw8WAh4EVGV4dAUfVm91cyBhdm
V6IHPDqWxlY3Rpb25uw6kgOiBibGFuY2RkZGsF3jbEtiYP6owubOheigPWF2Fq" />
</div>
...
```

Lorsque le navigateur fabrique une requête HTTP, il affecte la chaîne de caractères base64 à la valeur du contrôle HTML `__VIEWSTATE`. On retrouve notamment cette valeur dans notre extrait précédent d'une requête POST (en gras). Ainsi, en décodant ces informations le traitement interne des pages `.aspx` de ASP.NET est capable de retrouver tous les états de tous les contrôles.

La première fois que l'on rencontre cette astuce elle ne nous parait pas bien « propre ». Comprenez bien que cette technique est là pour pallier l'absence d'état dans un environnement web. Un tel comportement se révèle fort utile puisqu'en naviguant d'une page à une autre, les utilisateurs s'attendent à ce que les valeurs des contrôles ne changent pas. Il n'y avait pas de technique équivalente dans la technologie ASP et les développeurs étaient obligés de prévoir du code pour chaque contrôle sensé « se souvenir » de son état entre les requêtes.

## *Evénements postback et non-postback*

Lorsqu'un utilisateur travaille sur une page HTML dans un navigateur, il effectue des actions susceptibles de déclencher deux types d'évènements : Les *événements postback* et les événements *non-postback*.

Les événements *postback* provoquent l'envoi d'une requête POST au serveur pour lui fournir l'ensemble des états des contrôles de la page. Fort de ces informations, le serveur peut régénérer dynamiquement la page et la renvoyer au navigateur. Typiquement, notre exemple montre que le clic d'un bouton est un événement *postback*.

Les événements *non-postback* ne provoquent pas d'appel serveur. Un événement *non-postback* est sauvegardé et sera envoyé au serveur lors du prochain événement *postback*. Lorsque le serveur doit exécuter plusieurs événements *non-postback*, vous ne devez pas compter sur un quelconque ordre d'exécution. Typiquement les événements *non-postback* sont les changements d'états de contrôles. Notre exemple montre que le changement de la sélection dans un combo box est un évènement *non-postback*. On peut aussi citer l'écriture d'un texte dans une boite d'édition de texte. Vous pouvez forcer un événement *non-postback* à se comporter comme un événement *postback* en positionnant à `true` la propriété `AutoPostBack` du contrôle serveur sous-jacent. Par exemple, si vous ajoutez l'attribut `autopostback="true"` à notre contrôle serveur `Couleur`, une requête POST sera envoyée au serveur à chaque modification de la sélection de notre combo box :

```
...
  <form id="Form1" action="Default.aspx" method="post" runat="server">
  Couleur : <asp:dropdownlist id="Couleur" runat="server" autopostback="true">
              <asp:listitem>blanc</asp:listitem>
              <asp:listitem>noir</asp:listitem>
            </asp:dropdownlist>
            <br/>
...
```

Dans ce contexte, nous pouvons maintenant préciser que l'élément `<form>` définit un contrôle serveur de type `System.Web.UI.HtmlControls.HtmlForm`. Tous les contrôles serveur qui peuvent provoquer un événement *postback* doivent être déclarés entre les balises d'un élément `<form>`.

Nous avons déjà mentionné qu'ASP.NET sait associer la méthode `Btn_Click()` de notre classe `MyDefaultPage` à l'évènement *postback* « click sur Button1 » du fait que l'on affecte la valeur `Btn_Click` à l'attribut `OnClick` de `Button1` dans notre page `Default.aspx` :

```
...
 <asp:button id="Button1" text="Soumettre" OnClick="Btn_Click" runat="server"/>
...
```

En fait, la classe `System.Web.UI.WebControls.Button` présente un évènement `Click` de type la délégation `EventHandler`. Lors de la compilation d'une page `aspx`, ASP.NET sait que la valeur d'un attribut `OnXXX` correspond au nom d'une méthode qu'il faut associer à l'évènement `XXX` du contrôle serveur sous-jacent. On peut se passer de cette facilité du moment que l'on effectue cette association nous-même lors de l'initialisation de notre instance de `Default_aspx` déclenchée par une requête POST. Ainsi la page suivante est équivalente à notre page (nous détaillons un peu plus tard l'évènement `Page_Load()` déclenché par ASP.NET lors du chargement d'une page).

*Exemple 23-7 :* *Default.aspx*

```
...
 <asp:button id="Button1" text="Soumettre" runat="server"/>
...
```

*Exemple 23-8 :* *Default.aspx.cs*

```
using System ;
public partial class MyDefaultPage : System.Web.UI.Page {
   protected void Page_Load(object sender, EventArgs e) {
      if (IsPostBack)
         Button1.Click += Btn_Click;
   }
   protected void Btn_Click(Object sender, EventArgs e) {
      Msg.Text = "Vous avez sélectionné : "+Couleur.SelectedItem.Value ;
   }
}
```

Notez l'accès à la propriété `bool Page.IsPostBack{get;}`. Cette propriété vaut `true` si la requête courante est une requête POST. Nous l'utilisons car il n'y a pas lieu de s'abonner à l'évènement `Button1.Click` si l'on n'est pas dans le cas d'une requête *postback*.

Une question se pose : Comment la plomberie ASP.NET détermine quel évènement de quel contrôle serveur il faut déclencher lors de la réception d'une requête POST d'un client ? À l'instar de ce que l'on a vu pour le *viewstate* ASP.NET ajoute deux contrôles cachés nommés `__EVENTTARGET` et `__EVENTARGUMENT` à chaque page HTML générée, ainsi que du code *javascript* pour les initialiser. Par exemple voici un extrait d'une telle page HTML produite par une instance de notre classe `Default_aspx` :

```
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
```

```
      <center>
      <form method="post" action="Default.aspx" id="Form1">
<div>
<input type="hidden" name="__EVENTTARGET" id="__EVENTTARGET" value="" />
<input type="hidden" name="__EVENTARGUMENT" id="__EVENTARGUMENT"
 value="" />
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="..." />
</div>
<script type="text/javascript">
<!--
var theForm = document.forms['Form1'];
if (!theForm) {
    theForm = document.Form1;
}
function __doPostBack(eventTarget, eventArgument) {
    if (!theForm.onsubmit || (theForm.onsubmit() != false)) {
        theForm.__EVENTTARGET.value = eventTarget;
        theForm.__EVENTARGUMENT.value = eventArgument;
        theForm.submit();
    }
}
// -->
</script>
...
```

Ainsi, toute requête POST contient dans les paramètres `__EVENTTARGET` et `__EVENTARGUMENT` l'information permettant à la plomberie ASP.NET d'identifier le contrôle HTML responsable du *postback*.

## *ContolState*

Un problème courant rencontré en ASP.NET 1.x provient du fait que la chaîne base64 du *viewstate* devient énorme (>10.000 caractères) dés que l'on essaye d'y stocker des informations telles que le contenu d'une table par exemple pour un contrôle serveur de type `DataGrid`. Lorsque ce problème est rencontré, on peut choisir de désactiver le *viewstate* pour ce contrôle et de gérer son état autrement, par exemple en stockant le contenu de la table dans un cache du côté serveur ou en rechargeant le contenu à chaque requête. Il se pose alors un second problème : pour certains contrôles tels que le `DataGrid` les navigateurs utilisent le contenu du *viewstate* pour des raisons fonctionnelles telles que le déclenchement d'un évènement *postback* lors du changement des données. Ainsi, en ASP.NET 1.x on peut être tenté de désactiver le *viewstate* d'un contrôle pour des raisons de bande passante et de l'activer pour des raisons fonctionnelles.

En ASP.NET 2.0 le problème du *viewstate* énorme est amoindri du fait que les informations sont stockées d'une manière plus efficace dans la chaîne base64. Mais surtout, ASP.NET 2.0 introduit la notion de *controlstate*. Un contrôle serveur peut profiter du *controlstate* pour stocker les informations nécessaires à son bon fonctionnement. Les contrôles serveur suivants peuvent ainsi profiter du *controlstate* pour des raisons fonctionnelles et désactiver le *viewstate* pour des raisons de bande passante : `CheckBoxList`, `DetailsView`, `FormView`, `GridView`, `ListControl` et ses classes dérivées, `LoginView`, `MultiView` et `Table`.

Sachez que vous ne verrez pas de contrôles cachés spéciaux dans vos pages HTML pour stocker les *controlstate*. Les *controlstate* sont stockés comme une sous-section du *viewstate*.

## *Changer de page lors d'une requête*

Par défaut, toute requête *postback* réalisée par une page se fait sur elle-même. Cependant, il est souvent nécessaire de changer de page lors d'une requête *postback*. Cela est possible en spécifiant la nouvelle page cible avec la propriété `PostBackUrl` des contrôles serveurs de type boutons de la page source. Réécrivons notre page source `Default.aspx` de façon à ce qu'elle communique le choix de couleur à une autre page cible `DisplayColor.aspx` spécialisée dans l'affichage d'une couleur :

*Exemple 23-9 :* *`Default.aspx`*

```
<%@ Page Language="C#" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<script language=C# runat="server">
   private string m_SelectedColor;
   public string SelectedColor { get { return m_SelectedColor ; } }
   void Btn_Click(Object sender, EventArgs e) {
      m_SelectedColor = Couleur.SelectedItem.Value;
   }
</script>
<body><center>
   <form id="Form1" action="Default.aspx" method="post" runat="server">
      Couleur : <asp:dropdownlist id="Couleur" runat="server">
                  <asp:listitem>blanc</asp:listitem>
                  <asp:listitem>noir</asp:listitem>
                </asp:dropdownlist><br/>
      <asp:button id="Button1" text="Soumettre" OnClick="Btn_Click"
          postBackUrl ="~/DisplayColor.aspx" runat="server"/><br/>
   </form>
</center></body></html>
```

Il faut bien comprendre qu'étant donné que la page cible `DisplayColor.aspx` n'a pas les mêmes contrôles serveurs que la page source `Default.aspx`, les données contenues dans notre requête *postback* ne peuvent être directement exploitées. En fait, tout se passe comme si l'on effectué une requête GET sur la page cible. En interne, les données de la requête *postback* ont été exploitées par une instance de la classe représentant la page source. Cette instance est accessible par la propriété `PreviousPage` de la page cible. Aussi, si vous souhaitez exploiter les données de la requête *postback* à partir de la page cible, il est conseillé de les rendre accessible à l'aide de propriétés de la page source. Cette technique est illustrée par la propriété `SelectedColor` de notre page source `Default.aspx` et par la page cible `DisplayColor.aspx` :

*Exemple 23-10 :* *`DisplayColor.aspx`*

```
<%@ Page Language="C#" %>
<%@ previousPageType virtualpath ="~/Default.aspx" %>
<script runat="server">
void Page_Load(object sender, EventArgs e) {
  if (PreviousPage != null && PreviousPage.IsCrossPagePostBack) {
```

```
    Msg.Text = "Vous avez sélectionné : " + PreviousPage.SelectedColor ;
  } else { Msg.Text = "Pas de couleur sélectionnée." ; }
}
</script>
<html xmlns="http://www.w3.org/1999/xhtml" ><body>
<form id="form1" runat="server"><asp:label id="Msg" runat="server"/></form>
</body></html>
```

Il faut remarquer que la référence `PreviousPage` prend le type de notre classe `Default_aspx` puisque nous pouvons invoquer directement notre propriété `SelectedColor{get;}`. ASP.NET a pu typer cette référence grâce à la directive `<%@ PreviousPageType>` qui lui indique la page source. Dans certains cas, cette facilité peut se révéler être une limitation puisqu'elle empêche une page cible d'avoir plusieurs types de page source. Aussi, il est possible de se passer de cette directive à condition d'en payer le prix : avoir une page source non typée. Réécrivons notre page cible `DisplayColor.aspx` sans directive `<%@ PreviousPageType>` :

*Exemple 23-11 : DisplayColor.aspx*

```
<%@ Page Language="C#" %>
<script runat="server">
void Page_Load(object sender, EventArgs e) {
   if (PreviousPage != null ) {
      DropDownList Couleur =
         PreviousPage.FindControl("Couleur") as DropDownList;
      if (Couleur != null)
         Msg.Text = "Vous avez sélectionné : " +
            Couleur.SelectedItem.Value ;
   }
   if( Msg.Text.Length==0 ) Msg.Text = "Pas de couleur sélectionnée." ;
}
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<html xmlns="http://www.w3.org/1999/xhtml" ><body>
<form id="form1" runat="server"><asp:label id="Msg" runat="server"/></form>
</body></html>
```

Cette page peut ainsi afficher la couleur sélectionnée à partir de n'importe quelle page source contenant un contrôle `DropDownList` nommé `Couleur`.

Une autre technique disponible depuis ASP.NET 1.x permet aussi de réaliser un changement de page. Pour cela, il suffit d'utiliser la méthode `HttpServerUtility.Transfert( string pageCible)` comme ceci :

*Exemple 23-12 : Default.aspx*

```
...
   void Btn_Click(Object sender, EventArgs e) {
      m_SelectedColor = Couleur.SelectedItem.Value ;
      Server.Transfer("~/DisplayColor.aspx");
   }
...
```

```
<asp:button id="Button1" text="Soumettre" OnClick="Btn_Click" runat="server"/>
...
```

Cette technique présente un désavantage par rapport à l'utilisation d'un évènement `PostBackUrl` : l'URL ne change pas dans le navigateur.

Enfin, signalons la présence d'une nouvelle propriété `bool IsCrossPagePostBack{get;}`. Voici un tableau qui illustre la valeur de cette propriété selon le contexte :

```
PageSource réalise un postback sur elle-même :
PageSource.IsPostBack            true
PageSource.IsCrossPagePostBack   false
PageSource.PreviousPage          null
PageSource réalise un postBackUrl sur PageCible :
PageSource.IsPostBack            true
PageSource.IsCrossPagePostBack   true
PageSource.PreviousPage          null
PageCible.IsPostBack             false
PageCible.IsCrossPagePostBack    false
PageCible.PreviousPage           référence vers PageSource
PageSource réalise un transfert sur PageCible :
PageSource.IsPostBack            false
PageSource.IsCrossPagePostBack   false
PageSource.PreviousPage          null
PageCible.IsPostBack             false
PageCible.IsCrossPagePostBack    false
PageCible.PreviousPage           référence vers PageSource
```

## *Notion de contrôles serveur HTML et de contrôles serveur Web*

Jusqu'ici nous n'avons parlé que de contrôle serveur et de contrôles HTML. Les contrôles serveur sont exploités par ASP.NET et n'existent que du côté serveur. Les contrôles HTML sont inclus dans les pages HTML et sont exploités par les navigateurs, côté client.

Il existe deux catégories bien distinctes de contrôles serveurs : les *contrôles serveurs HTML* et les *contrôles serveur web*. À chaque type de contrôle HTML correspond un type de contrôle serveur HTML. La réciproque est fausse puisqu'il existe des contrôles serveur HTML tels que `HtmlGenericControl`. Un contrôle serveur HTML ne fait que produire le fragment de code HTML correspondant à son contrôle HTML associé. En revanche, les contrôles serveurs web sont plus évolués et en plus de présenter une API plus complète, ils produisent en général des fragments de code HTML contenant plusieurs contrôles HTML. Ainsi les contrôles serveur web ont souvent la préférence des développeurs. D'ailleurs notre page `Default.aspx` exemple exhibe trois contrôles serveur web (`<asp:dropdownlist>`, `<asp:button>` et `<asp:label>`) pour un contrôle serveur HTML (`<form>`). L'avantage des contrôles serveur HTML réside dans la simplicité de leur déclaration pour celui qui connaît le langage HTML. En effet, il suffit d'ajouter l'attribut `runat="server"` à n'importe quel contrôle HTML d'une page `.aspx` pour qu'ASP.NET crée un contrôle serveur HTML associé. De ce fait, les contrôles serveur HTML sont adaptés à un processus de migration de la technologie ASP vers ASP.NET. Voici notre page `Default.aspx` réécrite qu'avec des contrôles serveur HTML :

*Exemple 23-13 :* *Default.aspx*

```
<%@ Page Language="C#" %>
<script language=C# runat="server">
   void Btn_Click(Object sender, EventArgs e) {
      Msg.InnerText = "Vous avez sélectionné : " + Couleur.Value ;
   }
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <center>
      <form id="Form1" action="Default.aspx" method="post" runat="server" >
         Couleur : <select id="Couleur" runat="server" >
            <option value="blanc" />
            <option value="noir" />
         </select>
         <br/>
         <input id="Button1" type="Submit" value="Soumettre"
                OnServerClick="Btn_Click" runat="server" />
         <br/>
         <span id="Msg" runat="server" />
      </form>
      </center>
   </body>
</html>
```

La liste des contrôles serveur HTML est disponible dans l'article **HTML Server Controls Hierarchy** des **MSDN**. Ces contrôles font tous partie de l'espace de noms `System.Web.UI.HtmlControls` et dérivent tous de la classe `System.Web.UI.HtmlControls.HtmlControl` (qui elle-même dérive de la classe `System.Web.UI.Control`).

La liste des contrôles serveur web est disponible dans l'article **Web Server Controls Hierarchy** des **MSDN**. Ces contrôles font pratiquement tous partie de l'espace de noms `System.Web.UI.WebControls` et dérivent pratiquement tous de la classe `System.Web.UI.WebControls.WebControl` (qui elle-même dérive de la classe `System.Web.UI.Control`).

# *Cycle de vie d'une page*

Maintenant que nous avons défini les notions d'évènements *postback*, de *viewstate/controlstate* et d'arborescence de contrôles, il est temps d'exposer la chronologie des évènements qui ponctuent le traitement de toutes requêtes. Une bonne compréhension d'ASP.NET ne peut se faire sans la connaissance de ce cycle de vie. Nous rappelons que le modèle de threading d'ASP.NET consiste à associer un thread du pool pour traiter entièrement une requête. Aussi, les traitements abonnés aux évènements sont exécutés les uns après les autres par le même thread.

Dans le tableau suivant, chaque évènement est indexé par le nom de sa méthode abonnée par défaut. Nous précisons si les méthodes sont invoquées récursivement sur tous les contrôles ou bien uniquement sur la page. La plupart de ces méthodes sont protégées et virtuelles, aussi vous pouvez les réécrire avec vos propres implémentations. Le cas échéant, il est souvent conseillé

d'invoquer l'implémentation de la classe de base à partir de votre implémentation sous peine de perturber gravement le traitement de votre requête. Certains de ces évènements peuvent être interceptés autrement qu'en réécrivant une méthode virtuelle. Par exemple vous pouvez vous abonner à l'évènement `Control.Load` au niveau d'une page autrement qu'en réécrivant la méthode `OnLoad()`. Pour cela, il suffit de fournir une méthode nommé `Page_Load()`. Si vous positionnez la sous directive `AutoEventWireUp` à `true`, ASP.NET sait retrouver cette méthode par réflexion et l'invoquer au bon moment. Si vous ne souhaitez pas subir la coût de la réflexion, vous pouvez positionner cette sous directive à `false` et soit assigner vous-même les délégué aux évènements, soit utiliser la réécriture des méthodes virtuelles.

Pour vous aider à retenir cet enchaînement d'évènement, nous les avons classés par étapes. Les évènements qui ne sont déclenchés que lors d'une requête de type POST sont représentés par des cellules grisées. Quant aux nouveaux évènements introduits avec la version 2.0 d'ASP.NET, ils sont écrits en gras :

| Étape | Événements et méthodes | Contrôl | Commentaires |
|---|---|---|---|
| Construction | Appel des constructeurs | Tous | Possibilité d'initialiser les champs. |
| Initialisation | **`InitializeCulture`** | Page | Permet de personnaliser l'initialisation de la culture. |
| | `DeterminePostBackMode` | Page | Détermine si c'est une requête *postback* ou non. Possibilité de simuler une requête *postback*. |
| | **`OnPreInit`** | Page | Permet de spécifier une master page ou un thème. |
| | `OnInit` | Tous | Initialisation des états avec les valeurs spécifiées dans la page. |
| | `OnInitComplete` | Page | Possibilité d'altérer les états initiaux des contrôles et de créer des contrôles dynamiquement. |
| Restaure les états | `LoadPageStateFrom-PersistenceMedium` | Page | Possibilité de charger le *viewstate* à partir d'un autre endroit que dans la page (par exemple à partir d'une sauvegarde effectuée lors de la dernière requête). |
| | `LoadViewState` | Tous | Restaure les états à partir du *viewstate* et du *controlstate*. |
| | **`LoadControlState`** | Tous | |
| | `ProcessPostData` | Page | Restaure les états des contrôles à partir des valeurs spécifiées dans les paramètres *postback*. |

| | | | |
|---|---|---|---|
| Chargement de la page | **`OnPreLoad`** | Page | C'est ici que vous effectuez la plupart des actions personnalisées telles que l'abonnement à un évènement ou l'obtention d'une connexion BD. |
| | `OnLoad` | Tous | |
| | `OnDataBinding` | Tous | Charge les données à l'intérieur des contrôles qui supportent une liaison vers une source de données. |
| | `ProcessPostData` | Page | Appelée une deuxième fois au cas où des contrôles sont créés dynamiquement dans `OnLoad()`. Ainsi, leurs états peuvent être restaurés à partir des valeurs spécifiées dans les paramètres *postback*. |
| | `Validate` | Page | Déclenche la validation par les contrôles spécialisés. |
| | `RaiseChangedEvent` | Page | Déclenche les évènements dus à des changements d'états (suite logique de `ProcessPostData`). |
| | `RaisePostBackEvent` | Page | Déclenche les évènements *postback*. |
| | **`OnLoadComplete`** | Page | |
| Avant la fabrication | `OnPreRender` | Tous | Dernière chance d'agir sur l'arborescence des contrôles ainsi que leur états. |
| | **`OnPreRenderComplete`** | Page | |
| Sauve les états | `Page.SaveViewState` | Tous | Sauve les états dans le *viewstate* et le *controlstate*. |
| | **`SaveControlState`** | Tous | |
| | `SavePageState\ -To-Persistence\ -Medium` | Page | Possibilité de sauver le *viewstate* autre part que dans la page. Très utile pour ne pas surcharger la bande passante. |
| | **`OnSaveStateComplete`** | Page | |
| Rendu | `Render` | Tous | Fabrication de la page HTML puis envoie immédiat au client. |

| | | | |
|---|---|---|---|
| Finalisation | UnLoad | Tous | Dernière chance de libérer des ressources non gérées tels qu'une connexion à une BD. Ces méthodes sont appelées après l'envoi de la page HTML. |
| | Dispose | Tous | |

## *Traitement d'une page sur plusieurs threads*

Il arrive que le traitement d'une requête implique une longue attente du thread. Typiquement, cela se produit lorsque l'on accède à un service web lent ou lorsque l'on effectue une requête coûteuse sur une base de donnée. Dans ce cas, le modèle de threading d'ASP.NET est inefficace. En effet, l'attente d'un thread n'est pas sans conséquence puisque le nombre de threads du pool pouvant servir une requête est limité. Si la plupart des requêtes impliquent une attente de la part des threads, le serveur sera très vite surchargé et ce, quelque soit sa puissance. On est face à un goulot d'étranglement *by design*.

ASP.NET 2.0 propose une infrastructure pour régler ce problème. L'idée est que la finalisation du traitement d'une requête puisse se faire avec un thread différent de celui qui a servi à amorcer le traitement. Entre temps, l'accès à une ressource tel qu'un service web ou une base de données s'est fait sans aucune monopolisation d'un thread du pool. Dans le cas de l'accès à une base de données, cela est possible grâce aux nouvelles requêtes asynchrones d'ADO.NET 2.0 décrites en page 738. Cette possibilité permet d'effectuer une requête à une base de données sans mobiliser aucun thread du pool pendant l'attente des données. L'exemple suivant illustre une page qui exploite à la fois la possibilité ASP.NET 2.0 de traiter une page sur plusieurs threads et la possibilité ADO.NET 2.0 d'effectuer une requête asynchrone. Les données récupérées de la base sont alors présentées dans une `GridView` :

*Exemple 23-14 :* *Default.aspx*

```
<%@ Page Language="C#" Async="true" CodeFile="Default.aspx.cs"
                                    Inherits="_Default" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<body><form id="form1" runat="server">
   <asp:GridView ID="MyGridView" runat="server"
                 AutoGenerateColumns="true" />
</form></body></html>
```

*Exemple 23-15 :* *Default.aspx.cs*

```
using System ;
using System.Data ;
using System.Data.SqlClient ;
using System.Web ;
using System.Web.UI ;
using System.Threading ;
public partial class _Default : System.Web.UI.Page {
   private SqlCommand cmd ;
```

```
    private SqlConnection cnx ;
    protected void Page_Load(object sender, EventArgs e) {
       Response.Write("PageLoad thread:"+
                      Thread.CurrentThread.ManagedThreadId +"<br/>") ;
       PageAsyncTask task = new PageAsyncTask(BeginInvoke, EndInvoke,
                                              EndTimeOutInvoke, null);
       Page.RegisterAsyncTask(task);
    }
    public IAsyncResult BeginInvoke(object sender,
                                    EventArgs e,
                                    AsyncCallback callBack,
                                    object extraData) {
       Response.Write("BeginInvoke thread:" +
                      Thread.CurrentThread.ManagedThreadId + "<br/>") ;
       cnx =new SqlConnection("async=true ; server = localhost ; " +
                              "uid=sa ; pwd = ; database = ORGANISATION") ;
       cmd =new SqlCommand("SELECT * FROM EMPLOYES", cnx) ;
       cnx.Open() ;
       return cmd.BeginExecuteReader(callBack, extraData,
                                     CommandBehavior.CloseConnection) ;
    }
    public void EndInvoke(IAsyncResult result) {
       Response.Write("EndInvoke thread:" +
                      Thread.CurrentThread.ManagedThreadId + "<br/>") ;
       SqlDataReader rdr = cmd.EndExecuteReader(result) ;
       if (rdr != null && rdr.HasRows) {
          MyGridView.DataSource = rdr ;
          MyGridView.DataBind() ;
       }
       rdr.Close() ;
    }
    public void EndTimeOutInvoke(IAsyncResult result) {
       if (cnx != null && cnx.State != ConnectionState.Closed)
          cnx.Close() ;
       Response.Write("TimeOut") ;
       Response.End() ;
    }
    protected override void OnPreRender(EventArgs e) {
       Response.Write("OnPreRender thread:" +
                      Thread.CurrentThread.ManagedThreadId + "<br/>") ;
    }
    protected override void OnPreRenderComplete(EventArgs e) {
       Response.Write("OnPreRenderComplete thread:" +
                       Thread.CurrentThread.ManagedThreadId + "<br/>") ;
    }
}
```

Pour supporter ce modèle de traitement sur plusieurs threads, il faut positionner à `true` la sous-directive `Async` de la directive `<%@ Page>`. Lors de l'évènement `Load` on peut alors fabriquer des

instances de la classe System.Web.UI.PageAsyncTask. Chacune de ces instances représente un traitement qui va être effectué en dehors du processus aspnet_wp.exe. Aucun thread du pool n'est utilisé pour attendre la terminaison de ce traitement. Un tel traitement est représenté par trois délégués :

- Un délégué vers la méthode d'amorcage du traitement qui est exécutée par le thread qui a initié le traitement de la requête (i.e celui qui exécute Page_Load()). En l'occurrence, notre délégué référence la méthode BeginInvoke(). Elle est exécutée juste après l'évènement PreRender().
- Un délégué vers la méthode de finalisation du traitement qui est exécutée par un thread du pool lorsque le traitement a été signalé comme fini. En l'occurrence, notre délégué référence la méthode BeginInvoke. Elle est exécutée juste avant l'évènement PreRenderComplete.
- Un délégué vers une méthode exécutée par un thread du pool lorsque le traitement ne s'est pas effectué dans les temps impartis. En l'occurrence, notre délégué référence la méthode EndTimeOutInvoke().

La page HTML générée par cet exemple commence par ceci :

```
PageLoad thread:4
OnPreRender thread:4
BeginInvoke thread:4
EndInvoke thread:8
OnPreRenderComplete thread:8
...
```

Si vous souhaitez utiliser cette technique pour traiter d'une manière asynchrone un appel vers un service web, vous pouvez utiliser les méthodes IAsyncResult BeginXXX() et EndXXX(IAsyncResult) de la classe proxy générée par l'outil wsdl.exe (voir page 994).

# *Configuration d'une application ASP.NET*

Il est préférable d'avoir assimilé la section relative au fichier de configuration d'une application .NET page 58 avant d'aborder celle-ci.

## *Organisation des fichiers de configuration*

La configuration d'une application ASP.NET se fait au moyen de fichiers XML d'extension .Config. Dans ces fichiers, la plupart des paramètres relatifs à ASP.NET se trouvent dans la section <configuration>/<system.web>. Les paramètres de configuration d'une même application ASP.NET peuvent être disséminés dans les fichiers .Config suivant :

- Le fichier Machine.Config qui se trouve dans le répertoire d'installation du *framework* C:\WINDOWS\Microsoft.NET\Framework\v2.0.XXXXX\CONFIG\machine.config. C'est le seul fichier de configuration obligatoire.
- Le fichier Web.Config qui se trouve dans le répertoire racine des sites hébergés sur la machine, en général C:\Inetpub\wwwroot.
- Le fichier Web.Config qui se trouve dans le répertoire racine de l'application concernée.
- Des fichiers Web.Config qui peuvent se trouver dans n'importe quels sous répertoire de l'application concernée.

Pour une page stockée dans un sous répertoire [racine]\Foo de votre application web, les valeurs des paramètres définis dans le fichier [racine]\Foo\Web.Config masquent les valeurs des paramètres définis dans le fichier [racine]\Web.Config qui elles-mêmes masquent les valeurs des paramètres définis dans le fichier C:\textbackslashInetpub\textbackslashwwwroot\textbackslashWeb.Config qui elles-mêmes masquent les valeurs des paramètres définis dans le fichier Machine.Config. Avec ce modèle, il est par exemple aisé de définir une stratégie de sécurité par défaut commune à toutes les applications web hébergées sur la machine dans le fichier Machine.Config tout en se laissant la liberté d'appliquer une stratégie de sécurité exceptionnelle pour certaines pages confidentielles stockées dans un sous répertoire d'une application. En plus de cette souplesse, ce modèle permet aussi le déploiement d'une application ainsi que de ses fichiers de configuration par simple copie d'une arborescence de répertoires et de fichiers (déploiement xcopy). Notez enfin qu'ASP.NET rend inaccessible de l'extérieur les accès aux fichiers de configuration.

## *Les sections de configuration ASP.NET*

Voici la liste exhaustive des éléments de configuration ASP.NET 2.0 contenus dans l'élément <system.web>. Ceux qui sont en gras ont été introduits avec la version 2.0 d'ASP.NET. Nous donnons les références vers les explications dans le présent ouvrage lorsqu'elles sont disponibles :

```
<anonymousIdentification/> voir page <pageref id="ANONYMOUS_IDENTIFICATION"/>
<authentication/> voir page <pageref id="ASPNET_CONFIG_AUTHENTICATION"/>
<authorization/> voir page <pageref id="ASPNET_CONFIG_AUTHORIZATION"/>
<browserCaps/>
<caching/> voir page <pageref id="ASPNET_CACHING"/>
<clientTarget/>
<compilation/>
<customErrors/> voir page <pageref id="ASPNET_CUSTOM_ERRORS"/>
<deployment/>
<deviceFilters/>
<globalization/> voir page <pageref id="LOCALIZATION_WEBFORM"/>
<healthMonitoring/> voir page
<hostingEnvironment/>
<httpCookies/>
<httpHandlers/> voir page <pageref id="HTTP_MODULES"/>
<httpModules/> voir page <pageref id="HTTP_HANDLERS"/>
<httpRuntime/>
<identity/> voir page <pageref id="ASPNET_IMPERSONATION"/>
<machineKey/>
<membership/> voir page <pageref id="SECURITY_MEMBERSHIP"/>
<mobileControls/>
<pages/> voir page <pageref id="ASPNET_CONFIG_PAGES"/>
<processModel/> voir page <pageref id="ASPNET_PROCESS_MODEL"/>
<profile/> voir page
<protocols/>
<roleManager/> voir page <pageref id="SECURITY_ROLES"/>
<securityPolicy/>
<sessionPageState/>
<sessionState/> voir page <pageref id="ASPNET_SESSION_STATE"/>
```

```
<siteMap/> voir page <pageref id="ASPNET_CONFIG_SITEMAP"/>
<trace/> voir page <pageref id="ASPNET_TRACE"/>
<trust/>
<urlMappings/>
<webControls/>
<webParts/> voir page <pageref id="ASPNET_WEBPARTS"/>
<webServices/>
<xhtml11Conformance/>
```

Pour plus de détails sur la configuration d'une application web, nous vous conseillons de consulter l'article **ASP.NET Configuration** des **MSDN**.

Enfin, ajoutons qu'un élément `<location>`, enfant direct de l'élément `<configuration>`, permet de redéfinir pour chaque page les paramètres de configuration.

## *Configuration du modèle de processus*

La fiabilité fait partie des contraintes prépondérantes imposées à une application web. Lors du déploiement d'une application web, vous devez porter une attention particulière à l'élément de configuration `<processModel>` qui ne peut se trouver que dans le fichier `Machine.config`. Cet élément permet de paramétrer les possibilités suivantes destinées à consolider la fiabilité d'une application web :

- Vous pouvez indiquer à ASP.NET quand redémarrer un processus qui a un comportement anormal. L'attribut `requestQueueLimit` précise le nombre maximal de demandes en attentes. Lorsque ce nombre est atteint, ASP.NET considère que le processus a un comportement anormal. L'attribut `memoryLimit` précise le pourcentage maximal de mémoire système pouvant être utilisée. Au-delà de cette limite, ASP.NET considère que le processus a un comportement anormal. Dans ce cas, il lui donne la durée spécifiée par l'attribut `shutdownTimeout` pour s'arrêter automatiquement. Passée cette limite, ASP.NET arrête ce processus si ce n'est déjà fait et lance un nouveau processus. Notez que les demandes en attentes sont automatiquement réassignées à ce nouveau processus.
- Vous pouvez forcer ASP.NET à redémarrer un processus périodiquement, même si aucune condition anormale n'a été détectée. L'attribut `timeout` indique la durée au-delà de laquelle le processus doit être redémarré. L'attribut `requestLimit` indique le nombre de demandes traitées au-delà duquel le processus doit être redémarré. L'attribut `idleTimeout` indique la durée d'inactivité au-delà de laquelle le processus doit être arrêté.
- Les attributs `webGarden` et `cpuMask` indique à ASP.NET comment se comporter sur un serveur multi processeurs. En effet, si un serveur possède plusieurs processeurs, il est très efficace de pouvoir faire en sorte que chaque processeur gère seulement certains processus. On dit que l'on crée des affinités entre processus et processeurs. Cette technique est nommée *web garden*.
- Les attributs `userName` et `Password` indiquent l'identité de l'utilisateur *Windows* sous lequel doit s'exécuter le processus `aspnet_wp.exe`.

## *Mise à jour de la configuration*

En ASP.NET v1.x pour mettre à jour la configuration d'une application vous aviez le choix entre mettre à jour les fichiers XML `.Config` à la main ou manipuler programmatiquement le XML

contenu d'une manière non typée. ASP.NET 2.0 présente plusieurs facilités pour réaliser cette opération :

- *Visual Studio 2005* vous assiste avec l'intellisense lors de l'édition manuelle d'un fichier de configuration.
- Une nouvelle interface graphique web vous permet de mettre à jour la configuration directement à partir d'un navigateur exécuté en local (pour des raisons de sécurité). Vous pouvez y avoir accès à partir de *Visual Studio* avec l'onglet *Site Web* ▶ *ASP.NET Configuration*.
- Une nouvelle interface graphique de mise à jour de la configuration vient s'insérer dans la console de configuration d'IIS.
- Des nouvelles classes de base sont fournies pour manipuler programmatiquement et surtout, d'une manière fortement typée, le XML contenu (voir page 58).

### *Prise en compte des mises à jour*

Grâce au mécanisme de *shadow copy*, toutes modifications sur un fichier de configuration est prise en compte par ASP.NET. En conséquence, vous n'avez pas besoin d'arrêter puis de redémarrer l'application web pour que les changements soient pris en compte. Cela entraîne néanmoins la perte des états maintenus en mémoire de l'application (session et application). En conséquence, il faut éviter les mises à jour des fichiers de configuration d'une application web en production puisqu'elles peuvent gêner les utilisateurs. En outre, il faut savoir que les mises à jour concernant l'élément `<processModel>` ne sont prises en compte qu'au redémarrage d'IIS.

## *Pipeline HTTP*

### *Introduction*

Jusqu'ici, nous avons expliqué comment ASP.NET traite les demandes de pages `.aspx`. Lors de l'acheminement d'une requête HTTP entre la sortie du pipeline nommé (provenant en général du processus d'IIS) et le traitement de la page, nous avons eu l'occasion de mentionner le *pipeline http* ainsi que du code pour le traitement en interne des pages `.aspx`. Le pipeline HTTP est un mécanisme d'ASP.NET permettant principalement :

- De fournir des traitements qui agissent sur les requêtes ou (non exclusif) les réponses HTTP. Un tel traitement se nomme *module HTTP*. ASP.NET a ses propres modules HTTP qui traitent par exemple les identifiants de sessions ou l'authentification des utilisateurs. Nous allons voir comment créer nos propres modules HTTP.
- De fournir un traitement qui sert les requêtes HTTP (i.e qui fabrique une réponse HTTP en fonction d'une requête). Un tel traitement se nomme *handler HTTP*. ASP.NET a ses propres handlers HTTP, dont notamment celui qui s'occupe du traitement des pages `.aspx` en interne. Nous allons voir comment créer nos propres handlers HTTP.

La Figure 23-7 exhibe la place que tient le pipeline HTTP ainsi que les modules et handlers HTTP dans ASP.NET. Cette figure se base sur le modèle de processus de IIS 5.0 qui, rappelons le, est différent de celui de IIS 6.0 :

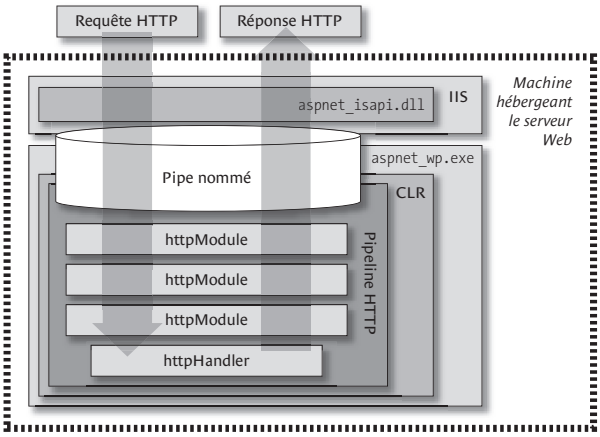


*Figure 23-7 : Module HTTP et Handler HTTP*

## *HttpApplication et le fichier Global.asax*

Avant d'aborder les modules et handlers HTTP il est nécessaire de comprendre la notion d'application HTTP. Nous avons déjà expliqué que le processus d'ASP.NET maintient un domaine d'application pour chaque application web hébergée. ASP.NET fait en sorte que chacun de ces domaines d'application contienne un objet instance de la classe `System.Web.HttpApplication` (ou d'une classe dérivée de celle-ci). Cette classe présente notamment des évènements qui sont déclenchés à différentes étapes clés, comme le lancement de l'application ou la création d'une nouvelle session. En vous abonnant à certains de ces évènements vous pouvez alors faire exécuter vos traitements au moment adéquat.

La technique standard pour s'abonner à ces évènements est de fournir les traitements dans des méthodes définies dans un fichier nommé `Global.asax`. Ce fichier doit être stocké à la racine de votre application. ASP.NET compilera ces méthodes dans une classe `ASP.Global_asax` qui dérive de la classe `HttpApplication`. L'objet application sera à l'exécution une instance de cette classe. Voici un exemple de fichier `Global.asax` :

*Exemple 23-16 :* *`Global.asax`*

```
<%@ Application Language="C#" %>
<script runat="server">
protected void Application_Start(object src, EventArgs e) { /*...*/ }
protected void Application_End(object src, EventArgs e) { /*...*/ }
protected void Application_AuthenticateRequest(object src, EventArgs e)
{ /*...*/ }
protected void Application_Error(object src, EventArgs e) { /*...*/ }
protected void Session_Start(object src, EventArgs e) { /*...*/ }
protected void Session_End(object src, EventArgs e) { /*...*/ }
protected void Session_BeginRequest(object src, EventArgs e) { /*...*/ }
protected void Session_EndRequest(object src, EventArgs e) { /*...*/ }
protected void Application_PostMapRequestHandler(object src,EventArgs e)
{ /*...*/ }
```

```
/*...*/
</script>
```

Nous vous invitons à consulter les MSDN pour obtenir la liste des évènements. Il est intéressant de remarquer qu'ASP.NET utilise la réflexion pour associer les méthodes définies dans le fichier `Global.asax` aux évènements de `HttpApplication`. Le nom d'une telle méthode commence par `Application_` ou `Session_` suivit du nom de l'évènement. En conséquence, vous devez porter une attention particulière au nom de ces méthodes puisque l'intellisense n'est pas là pour détecter les erreurs de syntaxe. Notez que certains évènements tel que `Application_Start()` ne sont exploitables que par l'intermédiaire du fichier `global.asax` tandis que d'autres sont en plus de cette technique, exploitables comme tout évènement d'une classe .NET.

## *Contexte HTTP*

Une instance de la classe `System.Web.HttpContext` est automatiquement créée par ASP.NET pour servir chaque requête HTTP. Lors du traitement d'une requête, cet objet contexte est accessible à tous les niveaux du pipeline HTTP grâce à la propriété statique `Current{get;}` de la classe `HttpContext`. Lors du traitement d'une requête, on a souvent besoin des informations accessibles au travers des propriétés de l'objet contexte, telles que le principal de l'utilisateur initiateur de la requête (si il est authentifié), la requête HTTP elle-même ou la réponse en cous de fabrication.

## *Module HTTP*

Nous pouvons maintenant nous intéresser au développement de nos propres modules HTTP. Un module HTTP est une instance d'une classe qui implémente l'interface `System.Web.IHttpModule` :

```
public interface IHttpModule {
   void Dispose() ;
   void Init( HttpApplication app ) ;
}
```

Un objet module est créé au lancement d'une application ASP.NET. Après l'appel au constructeur de sa classe, ASP.NET invoque la méthode `Init()`. Vous devez profiter de cette méthode pour abonner vos traitements sur les requêtes et les réponses HTTP. En général, on s'abonne aux évènements `BeginRequest()` et `EndRequest()` de `HttpApplication` mais vous pouvez choisir de vous abonner à n'importe quel autre évènement. Voici un exemple de module HTTP :

*Exemple 23-17 :*

```
using System ;
using System.Web ;
public class MyHttpModule : IHttpModule {
   public void Dispose() { }
   public void Init( HttpApplication app ) {
      app.BeginRequest += OnBeginRequest;
      app.EndRequest += OnEndRequest;
   }
   public void OnBeginRequest( object source, EventArgs args ) {
      HttpApplication app = source as HttpApplication ;
```

```
      app.Response.Write("BeginRequest : " +
                         DateTime.Now.ToLongTimeString()) ;
   }
   public void OnEndRequest( object source, EventArgs args ) {
      HttpApplication app = source as HttpApplication ;
      app.Response.Write("EndRequest : " +
                         DateTime.Now.ToLongTimeString()) ;
   }
}
```

Ce module ajoute la date de début et de fin de traitement à la réponse, en l'occurrence un fichier HTML. Remarquez que dans cet exemple les chaînes de caractères ajoutées en début et en fin de page HTML ne sont pas inclues dans la balise `<html>` mais la souplesse d'IE fait qu'il les affiche quand même :

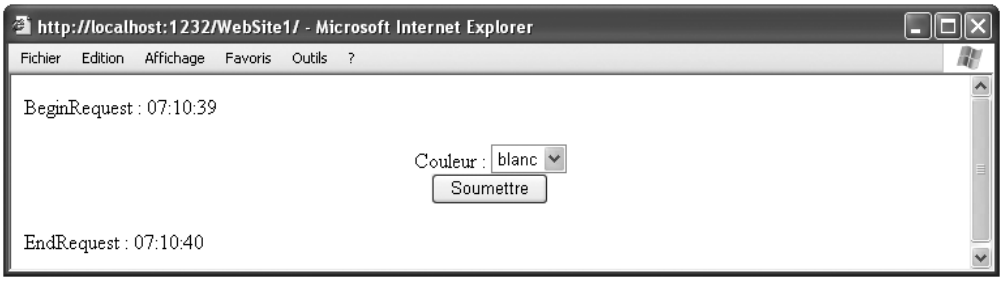


*Figure 23-8 : Effet de notre module HTTP*

Notez aussi que la méthode `Init()` de chaque module n'est appelée qu'une fois durant la durée de vie de l'application. C'est donc le même module qui est responsable des traitements sur toutes les requêtes. Puisque plusieurs requêtes peuvent être traitées par plusieurs threads simultanément, cette remarque est pertinente car elle souligne le besoin de synchronisation des accès aux ressources d'un module.

Enfin, il faut savoir que pour qu'ASP.NET prenne en compte un module particulier, il faut le lui préciser comme ceci dans le fichier de configuration de l'application :

*Exemple : `Web.Config`*

```
<?xml version="1.0"?>
<configuration
 xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <system.web>
      <httpModules>
         <add name="MyHttpModuleName" type="MyHttpModule,MyAsm"/>
      </httpModules>
...
```

L'attribut `type` précise le nom complet (i.e avec les espaces de noms) de la classe implémentant l'interface `IHttpModule`, suivi d'une virgule puis suivi du nom fort de l'assemblage contenant cette classe. Si le code de la classe est contenu dans le répertoire `/App_Code`, vous n'avez pas

besoin de préciser l'assemblage. Sinon, l'assemblage doit être contenu dans le répertoire `/bin`. Notez que la déclaration d'un module HTTP peut se faire dans la balise `<httpModules>` du fichier `Machine.config` afin qu'il soit pris en compte par toutes les applications ASP.NET de la machine. Dans ce cas, l'assemblage contenant la classe du module doit impérativement être présent dans le GAC.

Vous devez absolument concevoir des modules aussi indépendants que possible du fait que l'on ne maîtrise pas l'ordre de l'enchaînement des appels aux traitements des modules dans le pipeline HTTP. Cet ordre d'enchaînement est le même que l'ordre des appels aux méthodes `Init()` des modules. Donc, la Figure 23-7 n'est pas tout à fait exacte puisque les traitements `EndRequest()` sont appelés dans le même ordre que les traitements `BeginRequest()` et non dans l'ordre inverse. Empiriquement, on s'aperçoit que l'ordre des appels aux méthodes `Init()` des modules est le même que celui de la déclaration des modules dans la balise `<httpModules>` mais cet ordre n'est pas garanti par *Microsoft*.

## *Handler HTTP*

Vous pouvez définir vos propres handlers HTTP pour traiter vos propres types de ressources. Pour cela il faut définir une classe implémentant l'interface `System.Web.IHttpHandler` :

```
public interface IHttpHandler {
   bool IsReusable { get ; }
   void ProcessRequest(HttpContext context) ;
}
```

Une telle classe est en général déclarée dans un fichier d'extension `.ashx`. Par exemple, le handler HTTP suivant est une calculatrice qui prend une opération ainsi que deux opérandes dans les paramètres d'une requête GET et affiche le résultat dans la réponse. Notez que depuis Visual Studio 2005, nous avons l'intellisense sur les fichiers `.ashx`. Pour garder l'exemple simple, la réponse est un simple document texte sans balise `<html>` mais IE sait l'afficher :

*Exemple 23-18 :* *MyCalc.ashx*

```
<%@ WebHandler Language="C#" Class=MyCalcHttpHandler %>

using System ;
using System.Web ;
public class MyCalcHttpHandler : IHttpHandler {
   public bool IsReusable { get { return true ; } }
   public void ProcessRequest( HttpContext context ) {
      try {
         int a = int.Parse( context.Request["a"] ) ;
         int b = int.Parse( context.Request["b"] ) ;
         switch (           context.Request["op"]) {
            case "add": context.Response.Write(a + b) ; break ;
            case "sub": context.Response.Write(a - b) ; break ;
            case "mul": context.Response.Write(a * b) ; break ;
            case "div": context.Response.Write(a / b) ; break ;
            default: context.Response.Write("Invalid op!") ; break ;
         }
      } catch { context.Response.Write("Invalid params!") ;  }
```

```
    }
}
```

Vous pouvez alors vous servir de cette calculatrice en tapant la requête GET suivante dans votre navigateur :

```
http://localhost:1232/WebSite1/MyCalc.ashx?a=11&b=3&op=mul
```

Notez qu'une instance de `MyClassHttpHandler` est créée pour servir chaque requête. En outre, la ressource `MyCalc.ashx` est réelle car elle est matérialisée par un fichier à la racine de l'application. ASP.NET a su router la requête vers cette ressource et donc, vers le bon handler.

Vous pouvez aussi vous servir de handlers HTTP pour traiter des requêtes vers des ressources virtuelles i.e non matérialisées par un fichier à la racine de l'application. À l'instar de ce que l'on a vu pour les modules HTTP, il faut préciser un tel handler HTTP en indiquant la classe qui implémente `IHttpHandler` dans la balise `<httpHandlers>` du fichier `Web.Config` de l'application comme ceci :

*Exemple :* *`Web.Config`*

```
<?xml version="1.0"?>
<configuration
 xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <system.web>
      <httpHandlers>
         <add verb="GET" path="*.calc" type="MyCalcHttpHandler"/>
      </httpHandlers>
...
```

Ce que l'on a énoncé concernant l'assemblage contenant la classe, concernant le formatage de l'attribut `type` et concernant la déclaration dans le fichier `Machine.config` reste valable pour les handlers HTTP. Cependant cette technique est potentiellement plus puissante puisque vous pouvez par exemple router vers votre handler toutes les requêtes vers une ressource dont le nom se termine par l'extension `.calc`. Bien entendu, avec cette seconde technique il n'y a pas besoin d'un fichier `.ashx` et vous pouvez définir la classe `MyCalcHttpHandler` dans un fichier C# du répertoire `/App_Code`. Voici le genre de requête GET que peut alors servir notre handler :

```
http://localhost:1232/WebSite1/XYZ.calc?a=11&b=3&op=mul
```

Cette technique marche parfaitement avec le serveur web de *Visual Studio*. Il n'en est pas de même en production avec IIS. En effet, il faut indiquer à ce dernier que dans le cadre d'un répertoire virtuel, les requêtes d'extension `.calc` doivent être traitées par ASP.NET. Pour cela il faut effectuer la manipulation suivante sur l'outil de configuration d'IIS :

*[click droits sur le répertoire virtuel concerné]* ▶ *Propriétés* ▶ *Répertoire virtuel* ▶ *Configuration...* ▶ *Ajouter* ▶ *Extension=.calc* Chemin de l'exécutable=*C:\WINDOWS\Microsoft.NET\Framework\v2.0.XXXX\aspnet_isapi.dll* verbe=*GET*

Dans le cas où vous utilisez un fichier `.ashx`, cette manipulation est inutile car IIS est configuré par défaut pour router les requêtes concernées vers ASP.NET.

Dans le modèle de développement de handlers HTTP qui vient d'être exposé, un objet handler est créé par ASP.NET à la première requête et est alors exploité pour servir toutes les

requêtes suivantes. Vous pouvez agir sur ce comportement en ayant recours à l'interface `IHttpHandlerFactory`. Par exemple vous pouvez faire en sorte qu'un objet handler soit créé pour servir chaque requête ou même gérer un pool d'objets handler recyclable.

```
public interface IHttpHandlerFactory {
   IHttpHandler GetHandler( HttpContext context,
                            string      requestType,
                            string      url,
                            string      pathTranslated) ;
   void ReleaseHandler(IHttpHandler handler) ;
}
```

Pour cela, il suffit de préciser dans le fichier `Web.Config` votre classe qui implémente l'interface `IHttpHandlerFactory` à la place de celle qui implémente `IHttpHandler`. La méthode `GetHandler()` est invoquée par ASP.NET à chaque traitement d'une nouvelle requête et la méthode `ReleaseHandler()` à chaque terminaison de requête. Cela vous laisse la liberté de décider quand vos handler sont créés, quand ils doivent être recyclés et quand ils doivent être détruits.

# *Gestion des sessions et des états*

Les pages sont entièrement recréées à chaque requête vers le serveur. La conséquence est que l'on entend souvent dire que le Web est un environnement sans état. Néanmoins, de nombreuses applications web ont besoin de pouvoir gérer un état. Par exemple, la plupart des applications commerciales proposent un panier qui permet de stocker vos achats durant votre navigation. Cela permet de constamment avoir une vision globale de vos achats et de n'effectuer la transaction d'achat qu'une seule fois, à la fin de votre « shopping ».

Il y a de nombreuses façons de gérer des états avec ASP.NET. Nous avons déjà vu notamment la gestion de *viewstate/controlstate* qui a la particularité de faire transiter les données dans toutes les requêtes et les réponses. Cette technique est donc bien adaptée aux données non confidentielles de petite taille. Voici les principaux critères à prendre en compte lorsque vous décidez que votre application va gérer des états :

- La taille de vos données : Il est maladroit de faire transiter des données volumineuses (> 4Ko) sur le réseau. Il est aussi maladroit de stocker des données volumineuses dans la mémoire d'un processus. Au delà d'une certaine quantité de données, le mécanisme de mémoire virtuelle de *Windows* grève les performances d'une manière inacceptable à cause des nombreux accès au disque dur induits. On préfère stocker ce genre de données dans une base de données.
- Le niveau de sécurité requis : Il est dangereux de stocker chez le client ou de faire transiter à chaque requête des données confidentielles telles qu'un numéro de carte bancaire. Ces données transitent en général une fois sur le réseau sous une forme cryptée, lorsque le client les communique au serveur, puis sont stockées du côté serveur.
- Les performances souhaitées : Faire transiter des données sur le réseau, stocker des données en mémoire ou accéder à une base de données sont des opérations directement ou indirectement coûteuses en termes de performance. Il faut donc trouver un compromis selon vos besoins.

À l'instar de la gestion des *viewstate/controlstate*, le stockage de *cookies* chez le client permet de gérer des états dont les données associées sont peu volumineuses et non confidentielles. Cette

technique présente l'inconvénient majeur de ne pas être applicable à tous les clients puisque de nombreux utilisateurs décident de désactiver les cookies sur leurs navigateurs.

L'instance de `HttpApplication` globale à l'application maintient une instance de la classe `System.Web.HttpApplicationState`. Cet objet, lui aussi global à l'application, est accessible au travers de la propriété `Application{get;}` présentée à la fois par la classe `HttpApplication` et par la classe `Page`. Cet objet peut être vu comme un dictionnaire qui permet de stocker des données globales à l'application. Voici un exemple d'utilisation conjointe de ce dictionnaire et de cookies pour assigner un identifiant unique à chaque client qui se connecte (notez les appels aux méthodes `Lock()` et `UnLock()` pour synchroniser les accès au dictionnaire) :

*Exemple 23-19 :*

```
using System ;
using System.Web ;
public partial class MyDefaultPage : System.Web.UI.Page {
   protected void Btn_Click(Object sender, EventArgs e) {
      Msg.Text = "Vous avez sélectionné : "+Couleur.SelectedItem.Value ;
   }
   protected void Page_Load(object src, EventArgs args) {
      if (Application["ClientCounter"] == null) {
         Application["ClientCounter"] = 0 ;
      }
      HttpCookie cookie = Request.Cookies["ClientCounterCookie"] ;
      int clientNumber = -1 ;
      if (cookie == null) {
         Application.Lock() ;
         clientNumber = (int) Application["ClientCounter"] + 1 ;
         Application["ClientCounter"] = clientNumber ;
         Application.UnLock() ;
         cookie = new HttpCookie( "ClientCounterCookie" ) ;
         cookie.Value = clientNumber.ToString() ;
         Response.Cookies.Add(cookie) ;
      }
      else {
         clientNumber = Int32.Parse( cookie.Value ) ;
      }
      Response.Write("Client Number : " + clientNumber ) ;
   }
}
```

En fait, on pourrait s'inspirer de ce dernier exemple pour développer notre scénario de panier pour stocker les achats de chaque client. Les informations sur les achats courant seraient stockées du côté serveur et indexées par l'identifiant client. Cependant cette façon de faire a plusieurs points faibles :

- Si l'application redémarre, plusieurs clients peuvent avoir le même identifiant.
- Un client dont le navigateur ne supporte pas les cookies aura un identifiant différent à chaque requête.
- Il faudrait écrire pas mal de code pour pouvoir retrouver et stocker les informations relatives à chaque client hors du processus client.

- Il faudrait aussi maintenir une logique permettant d'invalider un identifiant d'un client après une certaine durée sans requête de sa part.

Pour toutes ces raisons, on préfère utiliser en général la notion de session fournie par ASP.NET.

## *Gestion d'une session*

Nous allons exposer comment utiliser une session pour sauvegarder du côté serveur l'historique des sélections des couleurs. Notre page ressemblera alors à ceci :

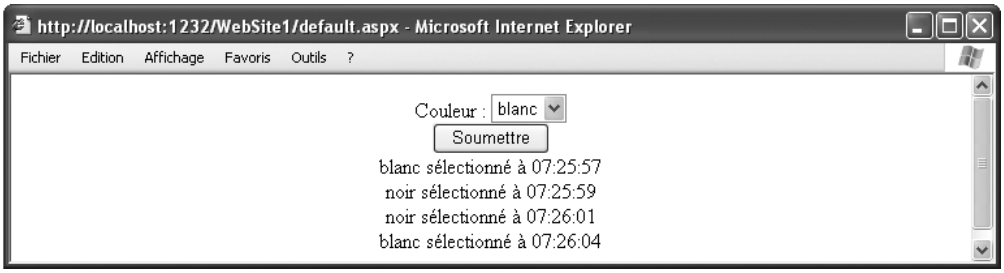


*Figure 23-9 : Sauvegarde des sélections dans une session*

Tout d'abord, nous prévoyons une classe `Item` dont chaque instance représente une sélection de couleur. Il est important de faire en sorte que tout objet stocké dans une session soit sérialisable. En effet, nous verrons que les sessions sont susceptibles d'être sérialisées selon leur mode de stockage. Notre classe `Item` est sérialisable puisqu'elle n'est constituée que de champs sérialisables :

*Exemple 23-20 :*

```
public class Item {
   public Item( string couleur , System.DateTime time ) {
      m_Couleur = couleur ; m_Time = time ;
   }
   private string m_Couleur ;
   private System.DateTime m_Time ;
   public override string ToString() {
      return  m_Couleur + " sélectionné à " + m_Time.ToLongTimeString() ;
   }
}
```

À chaque réception d'une nouvelle sélection, nous créons une nouvelle instance de `Item` que nous stockons dans la session courante. Puis, nous construisons l'historique des sélections qui sera inséré dans la page HTML retournée :

*Exemple 23-21 :*

```
...
void Btn_Click(Object sender, EventArgs e) {
   // Ajout d'une nouvelle sélection dans la session :
   System.Collections.Generic.List<Item> listItems =
```

```
      Session["ItemsSelected"] as System.Collections.Generic.List<Item> ;
   listItems.Add(
      new Item( Couleur.SelectedItem.Value , DateTime.Now ) ) ;
   Msg.Text = string.Empty ;
   // Fabrication de l'historique des sessions :
   foreach( Item item in listItems )
      Msg.Text += item.ToString() + "<br/>" ;
}
...
```

Pour que vous puissiez y sauver vos données, chaque session maintient en interne un dictionnaire. Pour que notre exemple fonctionne, il faut qu'à un moment donné nous initialisons l'entrée `"ItemsSelected"` de ce dictionnaire avec une nouvelle liste d'`Item`. Pour cela nous pouvons profiter de l'évènement `Session_Start` auquel nous nous abonnons au sein du fichier `Global.asax` :

*Exemple 23-22 :* *`Global.asax`*

```
<%@ Application Language="C#" %>
<script runat="server">
   protected void Session_Start(object src, EventArgs args) {
      Session["ItemsSelected"] = new
         System.Collections.Generic.List<Item>() ;
   }
</script>
```

Enfin, pour forcer ASP.NET à gérer les sessions, il suffit d'ajouter une balise `<sessionState>` à la configuration avec un attribut `mode` positionné à une valeur autre que `"Off"` (nous allons détailler les différentes valeurs que peut prendre l'attribut `mode`) :

*Exemple :* *`Web.Config`*

```
<?xml version="1.0"?>
<configuration
 xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <system.web>
      <sessionState mode="InProc"/>
...
```

Note que vous pouvez spécifier au moyen de l'attribut `timeout` la durée en minutes durant laquelle une session peut rester inactive avant qu'ASP.NET la détruise (la valeur par défaut est de 20 minutes) :

```
<sessionState mode="InProc" timeout="10"/>
```

## *Gestion de l'identifiant d'une session*

En interne, ASP.NET gère les sessions grâce à un mécanisme d'identifiants. Un nouvel identifiant de session est généré à chaque création d'une nouvelle session. Cet identifiant est renvoyé au client qui devra le communiquer lors de chaque requête dans le cadre de cette session. Par défaut, cet identifiant est stocké dans un *cookie* du côté client. Il est alors inséré par le navigateur dans chaque requête POST. Puisque tous les navigateurs ne supportent pas les *cookies*, vous

pouvez décider de configurer ASP.NET pour que les identifiants de sessions soient stockés dans l'URI. Pour cela il faut affecter la valeur "UseUri"à l'attribut cookieless :

```
<sessionState mode="InProc" cookieless="UseUri"/>
```

Une URI contenant un identifiant de session ressemble alors à ceci :

```
http://localhost/WebSite1/(S(e53uti455tgobvi2czsh4q45))/default.aspx
```

En plus de la valeur par défaut "UseCookies" et de la valeur "UseUri", l'attribut cookieless peut prendre la valeur "AutoDetect". Ainsi ASP.NET utilise un *cookie* pour passer l'identifiant de la session si le navigateur du client le permet. Sinon il se met en mode URI. Avec la valeur "UseDeviceProfile" vous pouvez spécifier à ASP.NET de choisir ou non le mode *cookie* selon les paramètres de *Device Profile* dans le fichier Machine.Config.

## *Les modes de stockage des sessions*

ASP.NET présente par défaut trois modes de stockage des sessions côté serveur. Vous communiquez à ASP.NET le mode à utiliser avec une des trois valeurs "InProc", "StateServer" et "SQLServer" pour l'attribut mode dans la configuration.

Le mode "InProc" stocke les sessions dans le domaine d'application contenant l'application web. Ce mode est à la fois le plus facile à exploiter et souvent le plus performant puisqu'il n'implique aucun accès hors du processus courant. Cependant il est à éviter en production pour de multiples raisons. Tous d'abord, si vous hébergez votre application dans une ferme web (i.e votre application tourne sur plusieurs machine) il faut s'assurer que les requêtes d'une même session soient toujours traitées sur le même serveur. Ensuite, les sessions sont volatiles et sont détruites à chaque déchargement du domaine d'application. Or, un domaine d'application d'une application web est susceptible d'être déchargé régulièrement pour différentes raisons : recompilation d'une page, crash, redémarrage régulier du processus etc. Ainsi, en production il vaut mieux préférer un mode de stockage des sessions hors du processus aspnet_wp.exe tel que "StateServer" ou "SQLServer".

Le mode "StateServer" stocke les sessions dans un processus dédié qui peut se trouver sur la même machine qui héberge l'application web ou sur une autre machine. Vous précisez cette machine ainsi qu'un port au moyen de l'attribut stateConnectionString :

```
<sessionState mode="StateServer"
   stateConnectionString="tcpip:127.0.0.1 :42424"/>
```

Le processus contenant les sessions se présente sous la forme du service *Windows* nommé *ASP.NET State Service*. Ce service est installé dés lors que vous installez ASP.NET sur une machine. Par défaut ce service est à démarrage manuel. Le port choisi par ce service pour écouter les requêtes de stockage de session est stocké dans la base des registres à l'entrée : HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\aspnet_state\Parameters.

Bien que le mode "StateServer" améliore grandement la fiabilité de la gestion de vos sessions par rapport au mode "InProc", ces dernières sont toujours volatiles et un crash du service entraîne leur perte. Ainsi, vous pouvez opter pour le mode "SQLServer" qui stocke les sessions dans une base de donnée type *SQL Server*. Ce mode est le plus fiable mais aussi le plus coûteux en terme de performances. Il présente notamment l'avantage de pouvoir exploiter les données concernant les sessions expirées afin de faire des statistiques par exemple. Pour exploiter ce mode il faut fournir la chaîne de connexion à la base de données au moyen de l'attribut sqlConnectionString comme ceci :

```
<sessionState mode="StateServer"
 sqlConnectionString="data source=127.0.0.1 ; user id=sa ;password="/>
```

Bien évidemment, il faut avoir installé au préalable les bases de données adéquates sur le serveur *SQL Server* concerné. Pour cela, il faut vous servir des scripts SQL `InstallSqlState.sql` ou `InstallPersistSqlState.sql` selon que vous souhaitez garder ou non vos sessions après leurs expirations. Ces scripts se trouvent dans le répertoire d'installation de .NET.

## *Fournir son propre mécanisme de gestion de sessions*

ASP.NET 2.0 vous permet de fournir votre propre mécanisme de gestion de sessions. Vous pouvez avoir besoin d'un tel mécanisme par exemple pour stocker vos sessions dans un autre SGBD que *SQL Server*. Une autre raison possible pourrait être le besoin de stocker les sessions dans des tables personnalisées, différentes de celles proposées par défaut.

En interne, le mécanisme de sessions est géré par un module HTTP nommé `"Session"` qui utilise un objet pour la gestion des IDs de session (dont la classe implémente l'interface `System.Web.SessionState.ISessionIDManager`) et un objet pour la gestion du stockage (dont la classe implémente l'interface `System.Web.SessionState.IHttpSessionState`). Vous pouvez communiquer à ASP.NET votre propre module ou bien votre propre classe pour la gestion des IDs de sessions ou bien votre propre classe pour la gestion du stockage des sessions au moyen du fichier `Web.Config` comme ceci :

*Exemple :* *`Web.Config`*

```
<?xml version="1.0"?>
<configuration
 xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <system.web>
      <httpModules>
         <remove name="Session" />
         <add    name="Session"
                 type="MyNamespace.MyStateModuleType,MyAsm"/>
      </httpModules>
      <sessionState mode="Custom"
         sessionIDManagerType="MyNamespace.MySessionIDType,MyAsm"
         customProvider="MySessionStateProvider" >
         <providers>
            <add name="MySessionStateProvider"
                 type="MyNamespace.MySessionStateProviderType,MyAsm" />
         </providers>
      </sessionState>
...
```

La description plus avant de l'implémentation de votre propre mécanisme de gestion de session dépasse le cadre de cet ouvrage. Nous vous invitons à consulter les articles **Implementing a Session-State Store Provider** et **Sample Session State Store Provider** des **MSDN**.

# *Le design pattern provider*

Le mécanisme que nous venons de présenter pour permettre de fournir son propre mécanisme de gestion de sessions est standard en ASP.NET. Il porte le nom de *design pattern provider*. Nous

aurons l'occasion de le retrouver dans d'autres domaines de fonctionnalités d'ASP.NET tels que la possibilité de fournir son propre mécanisme de gestion d'utilisateurs ou de rôles. Les caractéristiques du *design pattern provider* sont :

- Possibilité de fournir plusieurs fournisseurs avec des éléments `<add>` dans un élément `<providers>` qui est lui-même contenu dans l'élément représentant le domaine de fonctionnalité (en l'occurrence `<sessionState>`).
- Chaque fournisseur est nommé au moyen d'un attribut `name`.
- Chaque fournisseur est implémenté au moyen d'une classe qui dérive d'une certaine classe de base ou qui implémente une certaine interface.
- Cette classe ainsi que l'assemblage qui la contient son précisé au moyen d'un attribut `type`.
- L'implémentation d'un fournisseur est totalement libre de choisir le mode de persistance qu'elle gère pour stocker les données. Un tel mode de persistance peut être une base de données relationnelle, un fichier XML, un fichier texte, un service web etc.
- Le domaine de fonctionnalité présente un attribut qui prend le nom du fournisseur à exploiter à l'exécution.

Signalons que l'interface graphique web d'ASP.NET 2.0 présente un onglet *provider* qui vous permet d'administrer les fournisseurs par défauts pour chaque domaine de fonctionnalités.

# *Gestion des erreurs*

Il y a plusieurs types d'erreurs qui peuvent survenir dans une application ASP.NET en production :

- Le réseau peut être indisponible.
- Le serveur peut avoir crashé.
- Le serveur peut être surchargé.
- Un client demande une page qui n'existe pas ou qu'il n'est pas autorisé à voir.
- Un traitement applicatif du côté serveur lance une exception non rattrapée à cause d'un bug ou à cause de données corrompues.

Dans tous les cas, les conséquences de l'erreur seront une indisponibilité du service pour le client. En tant que développeur, seul les deux derniers types d'erreurs vous incombent. Dans le cas d'une demande d'une page qui ne peut être satisfaite, vous avez la possibilité de rediriger l'utilisateur vers une page d'erreur spéciale. Dans le cas d'une exception, ASP.NET produit par défaut une page HTML contenant le fragment de code responsable de l'exception ainsi que l'état de la pile à ce moment. Bien que ce comportement soit utile pour aider les développeurs à résoudre le problème, il est plutôt gênant qu'un utilisateur visualise une telle page. Cela peut aussi induire des problèmes de sécurité. Aussi, ASP.NET présente plusieurs techniques pour personnaliser la gestion des erreurs.

## La balise <system.web>/<customErrors>

Vous pouvez indiquer à ASP.NET la page qui doit être retournée à l'utilisateur en cas d'exception non rattrapée. La valeur de l'attribut `defaultRedirect` de la balise `<customError>` du fichier de configuration doit être le nom de cette page. En positionnant l'attribut `mode` à `"RemoteOnly"` vous pouvez faire en sorte que la page indiquée par `defaultRedirect` ne soit retournée qu'aux clients distants. Ainsi, lors de vos tests en local vous ne vous priverez pas des informations précieuses de la page d'erreur retournée par défaut par ASP.NET. L'attribut `mode` peut aussi prendre la valeur `"On"` pour toujours rediriger le client vers la page précisée même si ce dernier est en local, ou `"Off"` pour désactiver ce service. Enfin, vous pouvez aussi indiquer une page à retourner pour chaque erreur HTTP au moyen de sous balises `<error>` :

*Exemple :* *Web.Config*

```
<?xml version="1.0"?>
<configuration
 xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <system.web>
      <customErrors  mode="On" defaultRedirect="WebFormError.aspx">
         <error statusCode="403" redirect="MyError403.htm"/>
         <error statusCode="404" redirect="MyError404.aspx"/>
      </customErrors>
...
```

## L'évènement Application_Error

L'évènement `Error` de la classe `HttpApplication` est déclenché par ASP.NET lorsqu'une exception est non rattrapée. Vous pouvez vous abonner à cette évènement en fournissant une méthode `Application_Error()` dans le fichier `Global.asax`. Vous pouvez alors récupérer l'exception et construire une page réponse comme ceci :

*Exemple 23-23 :* *Global.asax*

```
<%@ Application Language="C#" %>
<script runat="server">
protected void Application_Error(object src, EventArgs args) {
   HttpUnhandledException eHttp =
      this.Server.GetLastError() as HttpUnhandledException ;
   // eApp est l'exception lancée non rattrapée.
   Exception eApp = eHttp.InnerException ;
   Response.Write("Erreur : " + eApp.Message) ;
   Response.End() ;  // <- ne pas oublier !!!
}
</script>
```

Pour qu'ASP.NET renvoie effectivement la page crée, il est obligatoire d'appeler la méthode `End()` sur l'objet `HttpResponse` courant durant l'exécution de la méthode `Application_Error()`. Autrement, l'abonnement à cet évènement empêche ASP.NET d'appliquer la stratégie de redirection de l'élément `<customErrors>`.

### *La propriété ErrorPage*

Durant le traitement d'une page vous pouvez à tout moment positionner la propriété `string Page.ErrorPage{get;set;}`. Vous précisez ainsi programmatiquement la page vers laquelle ASP.NET redirigera le client dans le cas d'une exception non rattrapée.

```
<%@ Page Language="C#" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <script language=C# runat="server">
      void Btn_Click(Object sender, EventArgs e) {
         this.ErrorPage = "WebFormError.aspx";
         Msg.InnerText = "Vous avez sélectionné : " + Couleur.Value ;
         throw new ApplicationException("Une erreur survient !") ;
      }
   </script>
...
```

Pour qu'ASP.NET applique effectivement la redirection, il faut que l'attribut `mode` soit positionné à `"On"` dans la balise `<customErrors>` du fichier de configuration.

Cette technique est très pratique puisqu'elle permet de préciser la page d'erreur en fonction du contexte courant d'exécution. Ainsi, si une page a plusieurs boutons, vous pouvez préciser une page d'erreur pour chaque traitement de chaque bouton. En outre, lors d'une exception non rattrapée lors du traitement d'une page, ASP.NET ne déclenche l'évènement `HttpApplication.Error` que si la valeur de cette propriété est nulle.

## *Traces, diagnostics et gestion des évènements*

### *Tracer le fonctionnement d'une application ASP.NET*

Vous pouvez tracer le fonctionnement d'une application grâce au handler HTTP `trace.axd`. Pour avoir accès à cette possibilité il faut activer les traces dans le fichier `Web.Config` comme ceci :

*Exemple :* *`Web.Config`*

```
<?xml version="1.0"?>
<configuration
 xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <system.web>
      <trace enabled="true" localOnly="true" />
...
```

Vous avez alors accès aux traces en tapant une URL du type `http://[machine]/[Racine]/Trace.axd` dans un navigateur. La page retournée contient un tableau de traces, une par requête HTTP traitée par le serveur. Ce tableau n'est pas mis à jour automatiquement et vous devez recharger la page (par exemple en cliquant F5) pour avoir accès à la liste des dernières requêtes. Vous avez la possibilité d'afficher une page d'information pour chaque trace en cliquant l'URL View Details d'une trace. Cette page est très complète. Elle contient des informations telles que l'arborescence des contrôles serveurs impliqués lors du traitement, l'état de la session et de l'application, les informations contenues dans la requête POST (si l'on

a affaire à une requête POST) l'état des variables internes au serveur etc. Naturellement, il n'est pas souhaitable que de telles informations soient visibles par les clients. Aussi en général vous positionnerez l'attribut `localOnly` à `"true"` pour empêcher que le handler `Trace.axd` soit exploitable à distance. Nous vous invitons à consulter les MSDN pour en savoir plus sur les attributs de la balise `<trace>`.

Vous pouvez vous servir de ce système pour inclure vos propres traces. Pour cela, il suffit de vous servir de l'objet de type `System.Web.TraceContext` accessible par la propriété `Page.Trace{get;}` comme ceci :

```
<%@ Page Language="C#" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <script language=C# runat="server">
      void Btn_Click(Object sender, EventArgs e) {
         Msg.InnerText = "Vous avez sélectionné : " + Couleur.Value ;
         Trace.Write("Ma trace write.");
         Trace.Warn("Ma trace warn.");
      }
   </script>
...
```

Dans une page de trace spécifique à une requête, les traces provoquées par la méthode `Warn()` sont affichées en rouge, alors que les traces provoquées par la méthode `Write()` sont affichées en noir.

## *Les compteurs de performance d'ASP.NET*

Lors de l'installation d'ASP.NET, une cinquantaine de compteurs de performances *Windows* sont aussi installés. On peut citer le nombre d'applications web en cours d'exécution, le nombre d'entrées dans le cache ou le nombre de requêtes traitées par seconde. Vous trouverez ces compteurs dans les deux catégories *ASP.NET Apps v 2.0.XXXXX* et *ASP.NET v 2.0.XXXXX*. Précisons que les compteurs de performances sont visualisables avec l'outil `perfmon.exe` (accessible avec *Menu démarrer* ▶ *Exécuter...* ▶ *perfmon.exe*). En outre, en page 115, nous expliquons comment accéder programmatiquement à la valeur d'un compteur de performance.

## *Gestion standard des évènements ASP.NET*

ASP.NET 2.0 présente un *framework* permettant de gérer d'une manière standard les évènements qui surviennent durant la vie d'une application web. Les types de ce *framework* sont dans le nouvel espace de noms `System.Web.Management`.

Un type d'évènement est représenté par une classe qui dérive directement ou indirectement de la classe `WebManagementEvent`. Le *framework* présente par défaut plusieurs classes d'évènements. Par exemple un évènement de type `WebErrorEvent` survient lorsqu'une exception n'est pas rattrapée lors du traitement d'une requête tandis qu'un évènement de type `WebRequestEvent` survient au début du traitement de chaque requête. Vous pouvez créer vos propres classes dérivées de `WebManagementEvent` pour représenter vos propres types d'évènements.

Ici aussi, l'architecture ASP.NET 2.0 utilise le *design pattern provider* pour la notion de *fournisseur de traitement d'évènements*. Un tel fournisseur est une classe qui dérive de la classe `WebEventProvider`. Trois fournisseurs sont exploitables par défaut :

- Le fournisseur `EventLogWebEventProvider` permet de loguer les évènements dans le log de *Windows*.
- Le fournisseur `TraceWebEventProvider` permet de loguer les évènements dans les traces ASP.NET.
- Le fournisseur `WmiLogWebEventProvider` permet de loguer les évènements en exploitant le framework *WMI* (*Windows Management Instrumentation*) de *Windows*.

En outre, la classe abstraite `BufferedWebEventProvider` représente une classe de base permettant de développer des fournisseurs qui stockent en mémoire des évènements. Vous pouvez créer vos propres fournisseurs de traitement d'évènements avec des classes dérivées de `WebEventProvider`.

Vous pouvez vous servir de la sous section `<eventMappings>` de la section `<healthMonitoring>` du fichier `Web.Config` pour préciser quels types d'évènements ASP.NET doit gérer. La sous section `<providers>` permet de définir les fournisseurs de traitement d'évènements exploitables. Enfin, la sous section `<rules>` contient les associations entre évènements et fournisseurs. Par exemple, le fichier `Web.Config` suivant contraint l'infrastructure ASP.NET à loguer toutes les exceptions non rattrapées dans les traces :

*Exemple :* *Web.Config*

```
<?xml version="1.0"?>
<configuration
 xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <system.web>
      <trace enabled="true"/>
      <healthMonitoring enabled="true">
         <providers>
            <add name="TraceLogProvider"
               type="System.Web.Management.TraceWebEventProvider,
                     System.Web,Version=2.0.3600.0,Culture=neutral,
                     PublicKeyToken=b03f5f7f11d50a3a"/>
         </providers>
         <eventMappings>
            <add name="Erreurs"
              type="System.Web.Management.WebErrorEvent,System.Web,
                    Version=2.0.3600.0,Culture=neutral,
                    PublicKeyToken=b03f5f7f11d50a3a" />
         </eventMappings>
         <rules>
            <add name="Traçage des erreurs."
               eventName="Erreurs"
               provider="TraceLogProvider"/>
         </rules>
      </healthMonitoring>
...
```

# Validation des données saisies

## Contrôle de validation

ASP.NET présente des contrôles spécialisés dans la validation des données saisies dans des contrôles de type HTMLInputText, HTMLTextArea, HTMLSelect , HTMLInputFile, TextBox, DropDownList, ListBox et RadioButtonList. L'exemple suivant illustre cette possibilité en utilisation deux contrôles de validation qui agissent sur une TextBox. Le contrôle de type RequiredFieldValidator assure que la valeur saisie dans la TextBox n'est pas vide. Le contrôle de type CompareValidator assure que la valeur saisie dans la TextBox est un entier supérieur ou égal à dix. Remarquez l'utilisation de la propriété ControlToValidate pour préciser sur quel contrôle agit chaque contrôle de validation :

*Exemple 23-24 :*

```
<%@ Page Language=C#  %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form1" action="Default.aspx" method="post" runat="server">
      <asp:Button ID="MyButton" runat="server" Text="Soumettre" />
      <asp:TextBox ID="MyTextBox" runat="server" />
      <asp:RequiredFieldValidator
         ID="MyRequiredFieldValidator"     runat="server"
         ControlToValidate="MyTextBox"     SetFocusOnError="true"
         ErrorMessage="Ne peut être vide !" Display="Dynamic" />
      <asp:CompareValidator
         ID="MyCompareValidator"           runat="server"
         ControlToValidate="MyTextBox"     SetFocusOnError=" true "
         ErrorMessage="CompareValidator"   Operator="GreaterThanEqual"
         Type="Integer"                    ValueToCompare="10">
         Doit être supérieur ou égal à10 !</asp:CompareValidator>
   </form>
   </body>
</html>
```

Si vous exécutez cet exemple, vous vous apercevrez que la validation se fait côté client. En effet, lorsque vous utilisez des contrôles de validation, ASP.NET génère automatiquement du code javascript pour la validation. Ce code est inséré dans la page HTML envoyée au client. Si le navigateur du client supporte javascript, ce code est exécuté du côté client à chaque click d'un bouton. Si la validation échoue, le code javascript de chacun des contrôles de validation qui a échoué génère une balise . Une telle balise affiche le message d'erreur spécifié par la propriété ErrorMessage sans déclencher d'évènement *postback*. Vous pouvez décider d'empêcher la validation côté client en positionnant la propriété EnableClientScript à false.

Lors du traitement d'une requête *postback* d'un client, chaque contrôle de validation effectue sa vérification du côté serveur. Ces validations se font indépendamment du fait que la validation avec du code javascript s'est effectuée ou non côté client. Si la validation échoue, la page est renvoyée au client. Sur une telle page, les contrôles de validation qui ont détecté un problème ont insérés leurs textes d'erreur dans des balises . Malgré cette garantie de validation côté serveur, nous vous conseillons de toujours activer la validation côté client pour éviter des évènements *postback* superflus.

Chaque type de contrôle de validation implémente l'interface `IValidator` qui expose notamment la propriété `bool IsValid{get;set;}`. La classe `Page` présente aussi une propriété `bool IsValid{get;}`. La validation côté serveur se fait lorsqu'ASP.NET appelle la méthode `Page.Validate()`. Cette étape positionne les propriétés `IsValid` de la page et des contrôles de validation. Un peu plus loin dans ce chapitre, nous expliquerons que cet appel se fait notamment après l'appel à `Page_Load()`. En conséquence, soit vous appelez vous-même la méthode `Page.Validate()` durant l'exécution de `Page_Load()`, soit vous ne testez aucune des propriétés `IsValid` durant cette exécution.

Dans l'exemple précédent nous avons positionné la propriété `Display` du contrôle de validation `MyRequiredFieldValidator` à la valeur `Dynamic`. Cela signifie que lorsque ce contrôle a effectué sa validation avec succès, il n'occupe pas d'espace sur la page. Cela permet de traiter proprement le cas où un autre contrôle de validation placé physiquement juste après `MyRequiredFieldValidator` et se rapportant aussi à `MyTextBox` échoue dans sa validation. En effet, le texte de ce deuxième contrôle de validation sera alors physiquement placé directement à coté de `MyTextBox`.

En plus des contrôles `RequiredFieldValidator` et `CompareValidator` que nous avons illustré, ASP.NET présente les contrôles de validation `RangeValidator` et `RegularExpressionValidator`. Ces quatre classes de contrôles dérivent de la classe `BaseValidator`.

- Grâce aux propriétés `ValidationDataType Type{get;set;}`, `string MaximumValue{get ; set;}` et `string MinimumValue{get;set;}` de la classe `RangeValidator` vous pouvez vous assurer que la valeur dans un contrôle est bien dans un intervalle donné. Les types possibles présentés par l'énumération `ValidationDataType` sont `String`, `Integer`, `Double`, `Date` et `Currency`.
- Grâce à la propriété `string ValidationExpression{get;set;}` de la classe `RegularExpressionValidator` vous pouvez fournir une expression régulière pour la validation d'une donnée saisie. Les expressions régulières sont décrites en page 625.
- En ce qui concerne la classe `CompareValidator`, nous attirons votre attention sur le fait que les types disponibles sont aussi dans l'énumération `ValidationDataType`. En outre, plutôt que de comparer la donnée saisie à une valeur fixe comme dans notre exemple, vous pouvez la comparer à celle d'un autre contrôle en précisant ce dernier dans la propriété `string ControlToCompare{get;set;}`.

Il est courant d'utiliser un contrôle de type `RequiredFieldValidator` en plus d'un de ces trois contrôles de validation puisque aucun d'eux ne détecte un problème si la donnée saisie est vide.

ASP.NET 2.0 introduit la propriété `bool SetFocusOnError{get;set;}` présentée par chacun des contrôles de validation. Elle permet de préciser s'il faut mettre le focus sur le contrôle à valider lorsque sa donnée est invalide. L'action de cette propriété s'effectue indépendamment du fait que la vérification échoue du côté client ou serveur.

## *Logique personnalisée de validation*

Si les validations proposées par les quatre classes de contrôles présentées ne vous conviennent pas, vous avez la possibilité de fournir votre propre logique de validation en ayant recours à un contrôle de validation de type `CustomValidator`. L'exemple suivant illustre cette possibilité en fournissant deux fonctions qui testent si un entier saisi est bien multiple de cinq. La fonction `MultipleDeCinq_Clnt()` est rédigée en `javascript` et est destinée à être exécutée du côté client.

La fonction `MultipleDeCinq_Svr()` est rédigée en C# et est destinée à être exécutée du côté serveur :

*Exemple 23-25 :*

```
<%@ Page Language=C#  %>
<script language="JavaScript">
   function MultipleDeCinq_Clnt( source, args) {
      if( args.Value % 5 == 0 )
         args.IsValid = true;
      else
         args.IsValid = false;
   }
</script>
<script language="C#" runat="server">
   void MultipleDeCinq_Svr(object source, ServerValidateEventArgs e) {
      e.IsValid = false;
      int num;
      if ( Int32.TryParse(e.Value, out num) )
         if ( num % 5 == 0 )
            e.IsValid = true;
   }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form1" action="Default.aspx" method="post" runat="server">
      <asp:Button ID="MyButton" runat="server" Text="Soumettre" />
      <asp:TextBox ID="MyTextBox" runat="server" />
      <asp:CustomValidator
         ID="MyCustomValidator" runat="server"
         ControlToValidate="MyTextBox"
         ClientValidationFunction="MultipleDeCinq_Clnt"
         OnServerValidate="MultipleDeCinq_Svr"
         ErrorMessage="Doit être multiple de 5 !"
         ValidateEmptyText="False" />
   </form>
   </body>
</html>
```

ASP.NET 2.0 introduit la propriété `bool CustomValidator.ValidateEmpty{get;set;}` qui permet de préciser si une valeur vide doit être considérée comme valide ou non.

## *Groupe de validation*

ASP.NET 2.0 introduit la notion de *groupe de validation*. Cette possibilité résout un problème courant rencontré en ASP.NET 1.x. Par défaut, à chaque événement tous les contrôles de validation sont exécutés. Or, il est courant que sur une même page on puisse déclencher un évènement sans nécessairement devoir valider toutes les données saisies. Par exemple, une page de formulaire peut contenir un bouton « *Chercher sur le web* » dont l'action n'a rien à voir avec les données saisies du formulaire.

Grâce à la nouvelle propriété `string ValidationGroup{get;set;}` présentée par les contrôles de validation, la classe `Button` et les classes dont les données peuvent être validées, vous pouvez partitionner l'ensemble des contrôles d'une page dans des groupes. Un click sur un bouton déclenchera seulement la vérification des contrôles de son groupe. Cette possibilité est illustrée par l'exemple suivant qui expose une page avec deux groupes de validation, chacun contenant un bouton, une `TextBox` et un `RequiredFieldValidator` :

*Exemple 23-26 :*

```
<%@ Page Language=C#  %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form1" action="Default.aspx" method="post" runat="server">
      <asp:Button  ID="MyButton1"       runat="server"
                   Text="Soumettre 1"   ValidationGroup="Groupe1"/>
      <asp:TextBox ID="MyTextBox1"      runat="server"
                   ValidationGroup="Groupe1"/>
      <asp:RequiredFieldValidator ID="MyRequiredFieldValidator1"
         ControlToValidate="MyTextBox1" runat="server"
         ValidationGroup="Groupe1"      ErrorMessage="Champ 1 vide !" />
      <br/>
      <asp:Button  ID="MyButton2"       runat="server"
                   Text="Soumettre 2"   ValidationGroup="Groupe2"/>
      <asp:TextBox ID="MyTextBox2"      runat="server"
                   ValidationGroup="Groupe2"/>
      <asp:RequiredFieldValidator ID="MyRequiredFieldValidator2"
         ControlToValidate="MyTextBox2" runat="server"
         ValidationGroup="Groupe2"      ErrorMessage="Champ 2 vide !" />
   </form>
   </body>
</html>
```

Vous pouvez déclencher la validation d'un groupe de validation du côté serveur grâce à la nouvelle méthode `void Validate(string validationGroup)` de la classe `Page`.

Les contrôles qui n'ont pas de groupe sont placés implicitement par ASP.NET dans un groupe global et anonyme. Ainsi la compatibilité ascendante avec ASP.NET 1.x est assurée.

## *La classe ValidationSummary*

Dans un formulaire avec plusieurs données à saisir vous pouvez souhaiter réunir tous les messages d'erreurs des contrôles de validation à un seul endroit. Ceci est possible grâce à la classe de contrôle de validation `ValidationSummary`. Un tel contrôle affiche la liste des messages d'erreur des contrôles de validation de son groupe. Vous pouvez choisir la façon dont les messages d'erreur sont listés grâce à la propriété `ValidationSummaryDisplayMode DisplayMode{get;set;}`. En outre, plutôt que d'afficher cette liste dans un élément `<span>` de la page vous pouvez choisir de l'afficher dans un message box en positionnant les propriétés `bool ShowMessageBox{get;set;}` et `bool ShowSummary{get;set;}`.

# Contrôles utilisateurs

ASP.NET vous offre la possibilité de définir vos propres contrôles serveur. Un tel contrôle est nommé *contrôle utilisateur*. Un contrôle utilisateur est en général défini au moyen d'un fichier d'extension `.ascx` débutant par une directive `<%@ Control>`. Ce fichier sera compilé par ASP.NET en une classe dérivée de la classe `Control`. Le nom de cette classe est précisé par la sous directive `ClassName`. Voici un exemple de contrôle utilisateur très simple :

*Exemple 23-27 :* *MyUserCtrl.ascx*

```
<%@ Control Language=C# ClassName="MyUserCtrl" %>
<% Response.Write("HTML du contrôle.") ; %>
```

Voici un exemple d'une page cliente de ce contrôle. On voit qu'une directive `<%@ Register%>` est nécessaire pour importer le nom de la classe, le nom du fichier `.ascx` qui la définit ainsi qu'un préfixe qui joue le rôle d'un espace de noms :

*Exemple 23-28 :* *MyUserCtrlClient.aspx*

```
<%@ Page Language=C#  %>
<%@ Register TagPrefix="PRATIQUE" Src="~/MyUserCtrl.ascx"
             TagName=UserCtrl %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <% Response.Write("HTML de la page.") ; %>
      <PRATIQUE:UserCtrl runat="server" />
   </body>
</html>
```

On souhaite en général avoir des contrôles utilisateurs paramétrables. Pour cela, on se sert de propriétés présentées par la classe qui représente notre contrôle :

*Exemple 23-29 :* *MyUserCtrl.ascx*

```
<%@ Control Language=C# ClassName="MyUserCtrl" %>
<script language=C# runat="server">
   private string m_Color ;
   public string Color{ get{ return m_Color;} set{ m_Color = value ; } }
   private string m_Text ;
   public string Text { get{ return m_Text ; } set{ m_Text = value ; } }
</script>
<p><font color="<%= Color %>"><%= Text %></font></p>
```

*Exemple 23-30 :* *MyUserCtrlClient.aspx*

```
<%@ Page Language=C#  %>
<%@ Register TagPrefix="PRATIQUE" Src="~/MyUserCtrl.ascx"
             TagName=UserCtrl %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <PRATIQUE:UserCtrl runat="server" Color="red" Text="hello red"/>
      <PRATIQUE:UserCtrl runat="server" Color="green" Text="hello green"/>
   </body>
</html>
```

Vous pouvez définir vous-même la classe représentant un contrôle utilisateur sans vous servir d'un fichier `.ascx`. Dans ce cas, chaque page cliente d'un tel contrôle doit inclure une sous directive `namespace` de la directive `<@ Register>` pour préciser l'espace de noms contenant la classe :

*Exemple 23-31 :* *MyUserCtrl.cs*

```
using System.Web.UI ;
namespace MyUserCtrls {
   public class MyUserCtrl : Control {
      private string m_Color ;
      public string Color{ get{return m_Color;} set{m_Color = value;}}
      private string m_Text ;
      public string Text{ get{return m_Text ; } set{m_Text = value;}}
      protected override void Render(HtmlTextWriter writer) {
         writer.Write("<p><font color=\"" + m_Color +
                      "\">" + m_Text + "</font></p>") ;
      }
   }
}
```

*Exemple 23-32 :* *MyUserCtrlClient.aspx*

```
<%@ Page Language=C#  %>
<%@ Register TagPrefix="PRATIQUE" namespace="MyUserCtrls" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <PRATIQUE:MyUserCtrl runat="server" Color="red" Text="hello red"/>
      <PRATIQUE:MyUserCtrl runat="server" Color="green"
                           Text="hello green"/>
   </body>
</html>
```

## *Contrôles utilisateurs composites*

On se sert souvent de contrôles utilisateurs pour exploiter un même formulaire sur plusieurs pages. Aussi, à l'instar de tous contrôles, un contrôle utilisateur peut avoir des contrôles serveurs enfants. Notez que vous pouvez vous aider du mode *design* de *Visual Studio* pour éditer un tel contrôle utilisateur :

*Exemple 23-33 :* *MyUserCtrl.ascx*

```
<%@ Control Language=C# ClassName="MyUserCtrl" %>
<script language=C# runat="server">
   void Btn_Click(Object sender, EventArgs e) {
      Msg.Text = "Vous avez sélectionné : "+Couleur.SelectedItem.Value ;
   }
</script>
Couleur : <asp:dropdownlist id="Couleur" runat="server">
            <asp:listitem>blanc</asp:listitem>
            <asp:listitem>noir</asp:listitem>
          </asp:dropdownlist>
```

```
<asp:button id="Button1" text="Soumettre" OnClick="Btn_Click"
   runat="server"/>
<asp:label id="Msg" runat="server"/>
```

*Exemple 23-34 :* *MyUserCtrlClient.aspx*

```
<%@ Page Language=C#  %>
<%@ Register TagPrefix="PRATIQUE" Src="~/MyUserCtrl.ascx"
             TagName=UserCtrl %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <form id="Form1" action="Default.aspx" method="post" runat="server">
         <PRATIQUE:UserCtrl ID="UserCtrl1" runat="server" /><br/>
         <PRATIQUE:UserCtrl ID="UserCtrl2" runat="server" />
      </form>
   </body>
</html>
```

En outré ASP.NET renomme automatiquement à l'exécution les noms des contrôles enfants en les préfixant par le nom du contrôle utilisateur parent suivi d'un _ (i.e un underscore, un souligné). Cela évite que plusieurs contrôles d'une même page aient des noms identiques.

## *Evènements sur un contrôle utilisateur*

Vous avez la possibilité de créer vos évènements dans un contrôle utilisateur et de vous y abonner à partir d'une page cliente avec la syntaxe `On[NomDeLévnènement]`. Ceci est illustré par l'exemple ci-dessous :

*Exemple 23-35 :* *MyUserCtrl.ascx*

```
<%@ Control Language=C# ClassName="MyUserCtrl" %>
<script language=C# runat="server">
   public event EventHandler BlancSelected;
   void Btn_Click(Object sender, EventArgs e) {
      if (Couleur.SelectedItem.Value == "blanc")
         BlancSelected(this, EventArgs.Empty);
   }
</script>
Couleur : <asp:dropdownlist id="Couleur" runat="server">
            <asp:listitem>blanc</asp:listitem>
            <asp:listitem>noir</asp:listitem>
          </asp:dropdownlist>
<asp:button id="Button1" text="Soumettre" OnClick="Btn_Click" runat="server"/>
```

*Exemple 23-36 :* *MyUserCtrlClient.aspx*

```
<%@ Page Language=C#  %>
<%@ Register TagPrefix="PRATIQUE" Src="~/MyUserCtrl.ascx"
             TagName=UserCtrl %>
<script language=C# runat="server">
   void BlancSelectedHandler(Object sender, EventArgs e) {
```

```
      Msg.Text = "Blancs sélectionnés !" ;
   }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <form id="Form1" action="Default.aspx" method="post" runat="server">
         <PRATIQUE:UserCtrl ID="UserCtrl1" runat="server"
                            OnBlancSelected="BlancSelectedHandler"/>
         <asp:label id="Msg" runat="server" EnableViewState="false"/>
      </form>
   </body>
</html>
```

## *Maintenir l'état d'un contrôle utilisateur entre les requêtes*

Un contrôle utilisateur peut se servir du *viewstate* pour maintenir son état entre les requêtes d'un même utilisateur. L'exemple suivant illustre cette possibilité en réécrivant notre contrôle utilisateur `MyUserCtrl` avec une classe sans état (i.e sans champs). Les états `Color` et `Text` sont maintenus entre les requêtes grâce au *viewstate*. Ils ne sont initialisés par le client que lors de la première requête :

*Exemple 23-37 :* *MyUserCtrl.cs*

```
using System.Web.UI ;
namespace MyUserCtrls {
   public class MyUserCtrl : Control {
      public MyUserCtrl() {
         ViewState["Color"] = string.Empty;
         ViewState["Text"] = string.Empty;
      }
      public string Color {
         get { return ViewState["Color"] as string ; }
         set { ViewState["Color"] = value ; }
      }
      public string Text {
         get { return ViewState["Text"] as string ; }
         set { ViewState["Text"] = value ; }
      }
      protected override void Render(HtmlTextWriter writer) {
         writer.Write("<p><font color=\"" + ViewState["Color"] +
                      "\">" + ViewState["Text"] + "</font></p>") ;
      }
   }
}
```

*Exemple 23-38 :* *MyUserCtrlClient.aspx*

```
<%@ Page Language=C#  %>
<%@ Register TagPrefix="PRATIQUE" namespace="MyUserCtrls" %>
<script language=C# runat="server">
```

```
   void Page_Load(Object sender, EventArgs e) {
      if ( !IsPostBack ) {
         MyUserCtrl1.Color = "red";
         MyUserCtrl1.Text = "hello red";
      } else { /* Pas la peine d'initialiser MyUserCtrl1 */ }
   }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <form id="Form1" action="Default.aspx" method="post" runat="server">
         <PRATIQUE:MyUserCtrl ID="MyUserCtrl1" runat="server" />
         <asp:button ID="Button1" runat="server" text="Soumettre" />
      </form>
   </body>
</html>
```

Notez la nécessité d'initialiser les entrées du *viewstate* requises dans le constructeur de notre contrôle utilisateur. Nous pourrions réécrire nos accesseurs get comme ceci pour nous affranchir de cette contrainte :

*Exemple 23-39 :* *MyUserCtrl.cs*

```
...
      public string Color {
         get {
            string s = ViewState["Color"] as string;
            return (s == null) ? string.Empty : s;
         }
         set { ViewState["Color"] = value ; }
      }
      public string Text {
         get {
            string s = ViewState["Text"] as string;
            return (s == null) ? string.Empty : s;
         }
         set { ViewState["Text"] = value ; }
      }
...
```

On peut aussi surcharger les méthodes `LoadViewState()` et `SaveViewState()` de la classe `Control` pour maintenir l'état d'un contrôle utilisateur dans le *viewstate*. Voici notre contrôle `MyUserCtrl` réécrit avec cette technique :

*Exemple 23-40 :* *MyUserCtrl.cs*

```
using System.Web.UI ;
namespace MyUserCtrls {
   public class MyUserCtrl : Control {
      private string m_Color ;
      public string Color { get{return m_Color;} set{ m_Color=value;} }
      private string m_Text ;
```

```
      public string Text { get{return m_Text;} set{ m_Text=value;} }
      protected override object SaveViewState() {
         object[] state = new object[2];
         state[0] = m_Color;
         state[1] = m_Text;
         return state;
      }
      protected override void LoadViewState(object _state) {
         if (_state != null) {
            object[] state = _state as object[];
            if (state[0] != null) m_Color = state[0] as string;
            if (state[1] != null) m_Text = state[1] as string;
         }
      }
      protected override void Render(HtmlTextWriter writer) {
         writer.Write("<p><font color=\"" + m_Color +
                      "\">" + m_Text + "</font></p>") ;
      }
   }
}
```

Sachez que vous pouvez aussi vous servir du *controlstate* pour sauver vos états entre les différentes requêtes. Pour cela il faut réécrire les méthodes `Control.LoadControlState()` et `Control.SaveControlState()` et en plus déclarer à la page que ce contrôle supporte le *controlstate* en appelant la méthode `RegisterRequiresControlState()` :

*Exemple 23-41 :* *MyUserCtrl.cs*

```
using System.Web.UI ;
namespace MyUserCtrls {
   public class MyUserCtrl : Control {
      ...
      protected override void OnInit(System.EventArgs e) {
         Page.RegisterRequiresControlState(this);
         base.OnInit(e);
      }
      protected override object SaveControlState() {
         ...
      }
      protected override void LoadControlState(object _state) {
         ...
      }
      ...
   }
}
```

## *Intégration d'un contrôle utilisateur avec le mode design de Visual Studio*

ASP.NET présente des facilités pour peaufiner l'intégration d'un contrôle utilisateur avec le mode *design* de *Visual Studio*. La description de ces possibilités dépasse le cadre du présent ouvrage aussi nous vous invitons à consulter l'article **Attributes and Design-Time Support** des **MSDN**.

# *Améliorer les performances avec la mise en cache*

ASP.NET présente un service de *cache*. Un service de cache permet d'améliorer sensiblement les performances en gardant en mémoire des données à la fois coûteuses à obtenir et souvent utilisées. Dans le cas d'une application web, de nombreuses données vérifient ces deux propriétés. En effet, la plupart des données sont obtenues à partir d'une base de données et sont donc coûteuses à obtenir. De plus, la génération de pages HTML requiert des ressources côté serveur. Or, les clients demandent souvent les mêmes pages qui contiennent les mêmes données. Des objets comme des instances de `DataSet`, déconnectés de la base de données et coûteux à obtenir, constituent aussi de bons candidats pour la mise en mémoire cache. L'offre de service de cache d'ASP.NET se décline sous trois formes : La mise en cache de pages, la mise en cache de fragments de page et la mise en cache de données.

## *Mise en cache de pages*

L'exemple suivant montre comment signifier à ASP.NET qu'il doit mettre une page en cache pour une minute avec la directive `OutputCache` :

*Exemple 23-42 :*

```
<%@ Page Language="C#"  %>
<%@ OutputCache Duration="60" VaryByParam="none" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <% Response.Write("Page générée à : " + DateTime.Now) ; %>
   </body>
</html>
```

La page est générée une première fois puis mise en cache pour une minute. Il est ici aisé de vérifier ce comportement puisque la page contient l'heure à laquelle elle a été générée.

Vous pouvez vous passer de la directive `OutputCache` et configurer programmatiquement la mise en cache grâce aux différentes méthodes de la classe `System.Web.HttpCachePolicy`. En effet, la classe `Page` maintient un objet de ce type accessible au travers de la propriété `Cache{get;}` :

*Exemple 23-43 :*

```
<%@ Page Language="C#"  %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <% Response.Write("Page générée à : " + DateTime.Now) ; %>
   </body>
   <script language=C# runat="server">
```

```
      void Page_Load(Object sender, EventArgs e) {
         Response.Cache.SetExpires( DateTime.Now.AddSeconds(60) );
         Response.Cache.SetCacheability( HttpCacheability.Public );
         Response.Cache.SetSlidingExpiration(true);
      }
   </script>
</html>
```

Dans cet exemple, nous déclarons une stratégie d'expiration décalée grâce à l'appel de la méthode `SetSlidingExpiration()` avec l'argument `true`. Cela signifie que le délai d'une minute après lequel la page sera détruite du cache est réactivé après chaque requête à la page. Les stratégies d'expiration décalées sont très utiles puisqu'elles permettent de garantir que le cache ne contient que des ressources effectivement utilisées. Cependant, sachez que vous ne pouvez déclarer une stratégie d'expiration décalée au moyen de la directive `OutputCache`. Vous êtes donc contraint de le faire programmatiquement.

L'appel à la méthode `SetCacheability()` permet de communiquer les endroits où une page peut être cachée. En effet, le protocole HTTP 1.1 connaît la notion de cache. Il permet de cacher une page directement chez le client ou dans un proxy transparent qui se place entre le serveur et les clients. Avec le cache maintenu par ASP.NET, cela fait trois caches possibles où une page peut être stockée. Veuillez vous référer à la description de l'énumération `HttpCacheability` dans les MSDN pour connaître les différentes options possibles. Sachez aussi que vous pouvez vous servir de la sous directive `Location` de la directive `OutputCache` pour communiquer les endroits où une page peut être cachée.

Enfin sachez que lorsque l'expiration décalée est activée, chaque requête au serveur entraîne la régénération de la page. Cette technique n'est donc utile que si vous avez des caches autres que celui maintenu par ASP.NET.

## *Mise en cache de plusieurs versions d'une page*

Vous pouvez mettre en cache plusieurs versions d'une même page selon les paramètres contenus dans les requêtes GET et POST. Pour cela, vous n'avez qu'à communiquer à ASP.NET l'ensemble des paramètres concernés au moyen de la sous directive `VaryByParam` de la directive `OutputCache`. Voici les différentes valeurs que peut prendre la sous directive `VaryByParam` :

- La présence de la sous directive `VaryByParam` est obligatoire. Si vous ne souhaitez pas exploiter cette possibilité, il faut lui fournir la valeur `"none"`.
- Si vous souhaitez utiliser plusieurs paramètres il faut séparer leurs noms avec des virgules. Dans ce cas, ASP.NET cachera une page pour chaque valeur du produit cartésien des paramètres spécifiés.
- Si vous souhaitez utiliser tous les paramètres, vous pouvez utiliser la valeur `"*"`.

L'exemple suivant se base sur l'Exemple 23-3. Il montre comment mettre en cache deux versions de cette page : une lorsque le client sélectionne la valeur `blanc`, l'autre lorsqu'il sélectionne la valeur `noir`.

*Exemple 23-44 :*

```
<%@ Page Language=C#  %>
<%@ OutputCache Duration=60  VaryByParam="Couleur" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
...
   <body>
      <% Response.Write("Page générée à : " + DateTime.Now) ; %> <br/>
...
         Couleur : <asp:dropdownlist id=Couleur runat="server">
                     <asp:listitem>blanc</asp:listitem>
                     <asp:listitem>noir</asp:listitem>
                   </asp:dropdownlist>
...
```

Attention : Comprenez bien qu'il ne faut pas exploiter cette possibilité avec des paramètres susceptibles de varier pour chaque client, tels que les noms des clients. Cela serait inefficace puisque le cache serait encombré avec des pages rarement demandées.

Il existe aussi une sous directive `VaryByControl` dont le comportement est assez proche de celui de `VaryByParam`. En effet, en spécifiant l'identifiant d'un contrôle dans `VaryByControl`, plusieurs versions d'une page peuvent être cachées selon la valeur fournie par le client pour ce contrôle. Ainsi, l'exemple suivant a un comportement similaire à celui de l'exemple précédent (notez que la présence d'une sous directive `VaryByControl` rend optionnelle la présence d'une sous directive `VaryByParam`) :

*Exemple 23-45 :*

```
<%@ Page Language=C#  %>
<%@ OutputCache Duration=60  VaryByControl="Couleur" %>
<script language=C# runat="server">
   void Btn_Click(Object sender, EventArgs e) {
      Msg.Text = "Vous avez sélectionné : "+Couleur.SelectedItem.Value ;
   }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <% Response.Write("Page générée à : " + DateTime.Now) ; %> <br/>
...
         Couleur : <asp:dropdownlist id=Couleur runat="server">
                     <asp:listitem>blanc</asp:listitem>
                     <asp:listitem>noir</asp:listitem>
                   </asp:dropdownlist>
...
         <asp:label id=Msg runat="server"/>
...
```

Comprenez bien qu'ASP.NET se base sur l'état du contrôle fourni par le client et non l'état du contrôle qu'il assigne lors de la fabrication de la page. Une conséquence est que cet exemple ne fonctionnerait plus si nous avions écrit `VaryByControl="Msg"`.

Vous pouvez aussi utiliser la sous directive `VaryByHeader` pour signifier à ASP.NET qu'il peut mettre en cache plusieurs versions d'une même page selon le contenu des entêtes des requêtes

des clients. Par exemple, la langue désirée par le client est mis dans l'entête `Accept-Language`. Aussi, plusieurs versions de la page suivante peuvent être cachées, une pour chaque langue spécifiée par un client :

*Exemple 23-46 :*

```
<%@ Page Language="C#"  %>
<%@ OutputCache Duration="60"
                VaryByParam="none" VaryByHeader="Accept-Language" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <script language=C# runat="server" >
      void Page_Load(Object sender, EventArgs e) {
         if (!IsPostBack) {
            switch ( Request.UserLanguages[0] ) {
               case "fr": Response.Write("Bonjour !") ; break ;
               case "de": Response.Write("Guten Tag!") ; break ;
               default: Response.Write("Hello!") ; break ;
            }
         }
      }
   </script>
   <body>
      <% Response.Write("Page générée à : " + DateTime.Now) ; %>
   </body>
</html>
```

Enfin, vous pouvez utiliser la sous directive `VaryByCustom` pour signifier à ASP.NET qu'il peut mettre en cache plusieurs versions d'une même page selon vos propres critères. Ces critères sont en général relatifs au type de navigateur qu'utilise le client. Si vous désirez cacher une version de page par type/version de navigateur, vous pouvez affecter la valeur `"Browser"` à cette sous directive. Sinon, il faut réécrire la méthode `GetVaryByCustomString()` de la classe `HttpApplication` pour fabriquer vous-même vos paramètres à tester.

L'exemple suivant expose une page qui est générée différemment selon que le navigateur client supporte ou nom les chaînes de caractères en gras. Nous fournissons donc notre propre paramètre `SupportsBold`. Pour chaque requête, ASP.NET peut connaître la valeur de ce paramètre en le demandant à notre code dans la méthode `GetVaryByCustomString()`.

*Exemple 23-47 :*

```
<%@ Page Language="C#"  %>
<%@ OutputCache Duration="60"  VaryByParam="none"
                VaryByCustom="SupportsBold" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <script language=C# runat="server" >
      void Page_Load(Object sender, EventArgs e) {
         if ( Request.Browser.SupportsBold )
            Response.Write("<B> Bonjour ! </B>") ;
         else
            Response.Write("Bonjour ! ") ;
      }
```

```
   </script>
   <body>
      <% Response.Write("Page générée à : " + DateTime.Now) ; %>
   </body>
</html>
```

*Exemple 23-48 :* *Global.asax*

```
<%@ Application Language="C#" %>
<script runat="server">
   public override string GetVaryByCustomString(
                              HttpContext ctx, string custom) {
      if( custom  == "SupportsBold" )
         return "SupportsBold=" + ctx.Request.Browser.SupportsBold;
      return string.Empty ;
   }
</script>
```

## *Mise en cache de fragments de pages*

Il est plutôt courant que seule une partie d'une page soit spécifique à chaque client. Par exemple, dans un site marchand la partie qui affiche les achats en cours varie avec chaque client tandis que la partie de la page qui affiche les caractéristiques des produits reste identique. Aussi, ASP.NET vous permet de ne mettre en cache qu'un fragment de page. Plus exactement, vous avez la possibilité de ne gardez en cache que des fragments HTML issus de contrôles utilisateurs. Cela est possible grâce à l'utilisation de la directive `OutputCache` lors de la définition d'un contrôle utilisateur dans un fichier `.ascx` :

*Exemple 23-49 :* *MyUserCtrl.ascx*

```
<%@ Control Language=C# ClassName="MyUserCtrl" %>
<%@ OutputCache Duration=60 VaryByParam=none %>
<br/>
<% Response.Write("Fragment généré à : " + DateTime.Now) ; %>
```

Par exemple la page suivante exploite le contrôle `MyUserCtrl`. En exécutant cet exemple, vous voyez que la page est générée à chaque requête tandis que le fragment HTML généré par le contrôle est caché et réutilisé.

*Exemple 23-50 :* *MyUserCtrlClient.aspx*

```
<%@ Page Language=C#  %>
<%@ Register TagPrefix="PRATIQUE" Src="~/MyUserCtrl.ascx"
             TagName=UserCtrl %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
      <% Response.Write("Page générée à : " + DateTime.Now) ; %>
      <form runat="server">
         <PRATIQUE:UserCtrl runat="server" />
      </form>
   </body>
</html>
```

Dans ce contexte, vous pouvez stocker plusieurs fragments HTML générés par un même contrôle utilisateur. En effet, ici aussi les sous directives `VaryByParam`, `VaryByControl` et `VaryByCustom` sont utilisables au sein de la directive `OutputCache`. En revanche les sous directives `VaryByHeader` et `Location` ne sont pas utilisables dans ce contexte.

Si votre contrôle présente des propriétés publiques, ASP.NET stockera automatiquement dans le cache un fragment HTML généré pour chaque valeur du produit cartésien de ces propriétés. Pour cela, il faut que ces valeurs soient fournies statiquement à la création du contrôle. Par exemple :

*Exemple 23-51 :* *MyUserCtrl.ascx*

```
<%@ Control Language=C# ClassName=MyUserControl %>
<%@ OutputCache Duration=60 VaryByParam=none  %>

<script runat="server">
   private string m_Couleur ;
   public  string Couleur { get { return m_Couleur ; }
                            set { m_Couleur = value ; } }
</script>
<br/>
<% Response.Write("Couleur : " + Couleur) ; %>
<br/>
<% Response.Write("Fragment généré à : " + DateTime.Now) ; %>
```

*Exemple 23-52 :* *MyUserCtrlClient.aspx*

```
...
         <PRATIQUE:UserCtrl Couleur=blanc runat="server" />
...
```

Enfin, soyez conscient qu'un contrôle qui produit des fragments HTML susceptibles d'être cachés ne doit pas être manipulé programmatiquement dans le code. En effet, lorsqu'ASP.NET décide de réutiliser un fragment HTML caché il ne crée pas le contrôle sous-jacent. Ainsi la référence vers ce contrôle est nulle.

## *Substitution post-cache*

ASP.NET 2.0 vous permet de ne régénérer que certaines parties d'une page cachée. Cette technique est connue sous le nom *substitution post-cache*. Elle permet d'adresser la problématique du cache des « pages semi variables » d'une manière complémentaire à l'utilisation de fragments cachés. En revanche, la substitution post-cache n'est pas exploitable à partir de contrôles utilisateurs ni à partir de master page cachées.

Pour exploiter la substitution post-cache, il vous suffit d'avoir recours au contrôle `<asp:Substitution>`. Ce dernier prend en paramètre nommé `MethodName` le nom d'une méthode statique. Cette dernière renvoie le fragment HTML variable sous forme d'une chaîne de caractères. Par exemple :

*Exemple 23-53 :*

```
<%@ Page Language=C#  %>
<%@ OutputCache Duration=60 VaryByParam=none %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <script runat="server">
      public static string Fct( HttpContext ctx ) {
         return "Substitution effectuée à : " + DateTime.Now;
      }
   </script>
   <body>
      <% Response.Write("Page générée à : " + DateTime.Now) ; %> <br/>
      <asp:Substitution Id=Substitution1 MethodName=Fct runat="server" />
   </body>
</html>
```

Vous pouvez aussi exploiter la substitution post-cache au moyen de la méthode `HttpResponse.WriteSubstitution()`.

## *Mise en cache de données*

Pour cacher des pages HTML ou des fragments de pages HTML, ASP.NET utilise en interne son propre moteur de cache. Vous pouvez exploiter ce moteur pour cacher vos propres données. L'exemple suivant illustre cette possibilité en cachant une instance de `DataView` construite à partir d'un `DataSet` récupéré à partir de la base de données `ORGANISATION`. Cette base est décrite en page 710 :

*Exemple 23-54 :*

```
<%@ Page Language=C# %>
<%@ Import Namespace ="System.Data" %>
<%@ Import Namespace ="System.Data.Common" %>
<%@ Import Namespace ="System.Data.SqlClient" %>
<script runat="server">
protected void Page_Load(object sender, EventArgs e) {
   DataView dv = Cache["Employes"] as DataView;
   if ( dv == null ) {
      using ( DbDataAdapter dAdapter = new SqlDataAdapter(
       "SELECT * FROM EMPLOYES",
       "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION")){
         DataSet ds = new DataSet() ;
         dAdapter.Fill(ds) ;
         dv = ds.Tables[0].DefaultView ;
         dv.AllowDelete = false ;
         dv.AllowEdit = false ;
         dv.AllowNew = false ;
         Cache["Employes"] = dv;
      } // end using dAdapter.
   }
   else Response.Write("Chargé du cache !") ;
   MyList.DataSource = dv ;
```

```
   MyList.DataTextField = "Nom" ;
   DataBind() ;
}
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form1" runat="server">
   <asp:ListBox ID="MyList" runat="server"/>
   </form>
   </body>
</html>
```

La mise en cache de données se fait d'une manière globale à l'application web. En conséquence, cette technique de mise en mémoire de données s'apparente avec l'utilisation du dictionnaire global, instance de `HttpApplicationState`. Il faut noter cependant deux différences :

- Lors de la récupération d'une donnée dans le cache, il n'est pas certains qu'elle y soit encore. Aussi, à l'instar de notre exemple, il faut toujours prévoir un mécanisme pour la récupérer autrement que par le cache pour prévoir le cas où elle n'y est plus.
- La philosophie de mise à jour des données cachées est différente de celle des données globales. La copie stockée dans le cache d'une donnée ne doit pas être modifiée. Si la donnée vient à changer alors qu'une copie est toujours dans le cache, il faut prévoir un mécanisme qui détruit la copie périmée pour la remplacer avec une copie à jour. Sachez qu'en interne, les accès aux données stockées dans le cache sont automatiquement synchronisés avec l'utilisation d'une instance de `ReaderWriterLock`.

## *Dépendances dans le cache*

ASP.NET présente un mécanisme de dépendance pour adresser la problématique des données du cache périmées que nous venons d'évoquer. Concrètement, lors de l'insertion d'une donnée dans le cache, vous pouvez la coupler avec une dépendance. Pour cela, la classe `System.Web.Caching.Cache` prévoit plusieurs surchargent de la méthode `Insert()` qui acceptent une instance de `System.Web.Caching.CacheDependency`.

Une dépendance peut être utilisée à l'insertion d'une ou de plusieurs données dans le cache. Une même instance de `CacheDependency` peut représenter une dépendance vers zéro, un ou plusieurs fichiers, zéro, un ou plusieurs répertoires et zéro, une ou plusieurs autres dépendances. Si l'état d'une seule de toutes ces entités vient à être modifié, la dépendance concernée le détecte et fait en sorte que les données qui lui sont associées dans le cache soient détruites.

L'exemple suivant illustre la mise en cache du contenu d'un fichier texte dans le cache. La page affiche le contenu de ce fichier texte en le récupérant dans le cache. Grâce à une dépendance sur ce fichier vous pouvez garantir que chaque client visualisera la dernière version de ce contenu :

*Exemple 23-55 :*

```
<%@ Page Language=C#  %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <% string content = this.Cache["Content"] as string ;
      if( content == null ) {
```

```
            Response.Write(@"Chargement dans le cache.<br/>") ;
            content = System.IO.File.ReadAllText(@"D:\Temp\Test.txt") ;
            CacheDependency dep = new CacheDependency(@"D:\Temp\Test.txt");
            Context.Cache.Insert("Content", content, dep ) ;
         }
         Response.Write(@"Contenu du fichier D:\Temp\Test.txt:" + content) ;
      %>
      </body>
   </html>
```

La classe `CacheDependency` présente de nombreux constructeur vous permettant de communiquer les entités associées avec une dépendance.

En outre, la méthode `Cache.Insert()` présente une surcharge qui prend en paramètre une valeur de l'énumération `CacheItemPriority`. Cette valeur vous permet de donner plus ou moins d'importance à vos données cachées. Bien entendu, l'algorithme de purge du cache d'ASP.NET tient compte de cette priorité.

Enfin, cette surcharge de la méthode `Cache.Insert()` prend aussi en paramètre un délégué de type `CacheItemRemovedCallback`. Cela vous donne un moyen d'être averti par un appel de méthode lorsqu'une donnée du cache est détruite.

## *Dépendances sur des données d'une base SQL Server*

ASP.NET 2.0 présente aussi la classe `SqlCacheDependency` qui permet de créer des dépendances vers une table ou des lignes d'une table d'une base de données type *SQL Server*. Une telle dépendance peut provoquer la destruction des données du cache associées dés que les données d'une table sont modifiées. Voici comment modifier l'Exemple 23-54 pour créer une dépendance sur la table `EMPLOYES` :

*Exemple 23-56 :*

```
...
         dv.AllowNew = false ;
         Cache.Insert("Employes", dv,
            new SqlCacheDependency("DbORGANISATION","EMPLOYES"));
      } // end using dAdapter.
...
```

Naturellement, il faut préciser à quelle base de données correspond l'alias `DbORGANISATION` comme ceci :

*Exemple :* *Web.config*

```
<?xml version="1.0"?>
<configuration
 xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <connectionStrings>
      <add name="CnxStrORGANISATION"
         connectionString="Data Source=localhost;
                Integrated Security=SSPI;Initial Catalog=ORGANISATION"
         providerName="System.Data.SqlClient"/>
```

```
   </connectionStrings>
   <system.web>
      <caching>
         <sqlCacheDependency enabled="true">
            <databases>
               <add name="DbORGANISATION"
                  connectionStringName="CnxStrORGANISATION"
                  pollTime="500"/>
            </databases>
         </sqlCacheDependency>
      </caching>
...
```

Si vous utilisez SQL Server 2005 comme SGBD sous-jacent, vous pouvez même ne dépendre que sur la mise à jour de certaines lignes d'une table. Vous précisez ces lignes au moyen d'une commande SELECT comme ceci :

*Exemple 23-57 :*

```
...
         dv.AllowNew = false ;
         SqlConnection cnx = new SqlConnection(
      "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION");
         SqlCommand cmd = new SqlCommand(
      "SELECT * FROM EMPLOYES WHERE EmployeID=6 OR EmployeID=7", cnx);
         Cache.Insert("Employes", dv,
               new SqlCacheDependency(cmd));
      } // end using dAdapter.
...
```

Cette possibilité de dépendance sur les données d'une base est aussi exploitable lors de la mise en cache d'une page ou d'un fragment de page au travers de la sous directive `SqlDependency` de la directive `<%@ OutputCache>` :

```
<%@ OutputCache Duration=60 VaryByParam="none"
                SqlDependency="DbORGANISATION:EMPLOYES" %>
```

En interne, le mécanisme mis en œuvre pour détecter les changements est complètement différent selon la version de votre SGBD *SQL Server*.

- Dans le cas de *SQL Server 2005*, le mécanisme de *Query Notification* de ce SGBD est exploité. En interne, une dépendance ADO.NET 2.0 est créée sur la table ou les lignes concernées. Elle est déclenchée par ce mécanisme dés qu'une mise à jour a eu lieu.
- Dans le cas d'une version antérieure à *SQL Server 2005* (7 ou 2000), un mécanisme de polling est mis en œuvre du côté serveur ASP.NET. Un thread background est utilisé régulièrement pour détecter si les données d'une table ont été modifiées. L'intervalle utilisé est spécifié en milli secondes par l'attribut `pollTime` de l'élément `caching\sqlCacheDependency\databases`. Sa valeur est de 5 secondes par défaut. Une certaine préparation avec l'outil en ligne de commande `aspnet_regsql.exe` est nécessaire pour activer ce mécanisme. Il faut tout d'abord indiquer à votre SQBD quelle base de données supporte ce mécanisme (`ed` pour `enable database`) :

```
>aspnet_regsql.exe -S <host> -U <usr> -P <pwd> -d <database> -ed
```

Ensuite, il faut utiliser cet outil pour activer une à une chaque table qui supporte ce mécanisme (`et` pour `enable table`) :

```
>aspnet_regsql.exe -S <host> -U <usr> -P <pwd> -d <database>
-t <table> -et
```

En interne, cet outil crée dans votre base de données des procédures stockées, des déclencheurs et des tables nécessaires à l'exécution de ce mécanisme.

## *Dépendances personnalisées dans le cache*

ASP.NET 2.0 offre la possibilité de fournir des dépendances vers vos propres types de ressources. Pour cela, il faut prévoir une classe dérivée de `CacheDependency` qui surveille l'état de la ressource. Une instance de cette classe doit appeler la méthode `NotifyDependencyChanged()` lorsqu'elle détecte un changement d'état. Le moteur de cache détruira alors les données du cache qui sont associées à cette instance et appellera la méthode `DependencyDispose()` pour lui données une chance de libérer les ressources qu'elle utilise.

Tout ceci est illustré par l'exemple suivant qui présente un squelette de classe dérivée de `CacheDependency`. Nous utilisons un timer de type `System.Timers.Timer` pour surveillez périodiquement l'état d'une ressource fictive :

*Exemple 23-58 :*

```
using System ;
using System.Web.Caching ;
using System.Timers ;
public class CustomCacheDependency : CacheDependency {
   private Timer m_Timer = new Timer() ;
   private string m_RessourceId ;
   public CustomCacheDependency( int pollIntervalSec,
                                 string ressourceId ) {
      m_RessourceId = ressourceId ;
      m_Timer.Interval = pollIntervalSec * 1000 ;
      m_Timer.Elapsed += this.CheckDependencyHandler ;
      m_Timer.Start() ;
   }
   private void CheckDependencyHandler( object sender,
                                        ElapsedEventArgs e) {
      // Ici, placez le code pour détecter si l'état de la resource
      // indexée par m_RessourceId a changé.
      bool ressourceHasChanged = false ;
      if (ressourceHasChanged)
         NotifyDependencyChanged(this, EventArgs.Empty);
   }
   protected override void DependencyDispose() {
      m_Timer.Stop() ;
   }
}
```

# *Sources de données*

## *Lier programmatiquement contrôles et sources de données*

Dans la section précédente, l'Exemple 23-54, page 924 montre comment exploiter un contrôles ASP.NET de type `ListBox` pour présenter les données d'une colonne contenue dans une `DataView`. Pour cela, il suffit de fournir la `DataView` au travers de la propriété `object ListBox.DataSource{set;}`, de fournir le nom de la colonne à afficher au travers de la propriété `string ListBox.DataTextField{set;}`, puis d'appeler la méthode `DataBind()` sur l'objet représentant la page courante. Cette méthode est présentée par la classe `Control`. Par conséquent, elle est aussi présentée par la classe `Page` et donc par toutes les classes représentant des pages, ainsi que tous les contrôles serveurs. L'implémentation par défaut de cette méthode consiste à appeler la méthode `DataBind` sur chacun des contrôles enfants.

Seuls certains types de contrôles serveur tels que `ListBox` implémentent effectivement cette méthode. Cette implémentation consiste à récupérer dans une copie en interne toutes les données de l'objet fourni à l'aide de la propriété `DataSource`. Ces données sont alors exploitées par la méthode `Render()` du contrôle qui les affiche dans la page HTML.

La classe représentant une source de données doit implémenter une des interfaces :

- `System.ComponentModel.IListSource` (implémentée par les classes `DataSet`, `DataTable`).
- `IEnumerable` (implémentée par les classes `DataView`, `ArrayList`, `List<>`, `Dictionary<,>` etc).
- `IDataReader` (implémentée par la classe `DbDataReader` ainsi que ses classes dérives).

Dans l'Exemple 23-54, les données sont déjà contenues dans le `DataSet` sous-jacent lors de l'appel à la méthode `DataBind`. Cela implique qu'il existe trois versions de ces données : dans le `DataSet`, dans la `ListBox` après l'appel à la méthode `DataBind()` puis dans la section `__VIEWSTATE` de la page HTML rendue. Si vous pouvez vous le permettre, vous pouvez désactiver le stockage de données d'un contrôle dans la section `__VIEWSTATE`. En outre, certaines sources de données telles qu'un `DataReader` ne contiennent pas en interne les données. Elles ne représentent qu'un moyen d'accéder aux données. Ainsi dans l'exemple suivant il n'y a plus qu'une seule copie des données, celle créée dans l'objet `ListBox` lors de l'appel à la méthode `DataBind()`.

*Exemple 23-59 :*

```
<%@ Page Language=C# %>
<%@ Import Namespace ="System.Data" %>
<%@ Import Namespace ="System.Data.Common" %>
<%@ Import Namespace ="System.Data.SqlClient" %>
<script runat="server">
protected void Page_Load(object sender, EventArgs e) {
   string sCnx =
     "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION" ;
   string sCmd = "SELECT * FROM EMPLOYES" ;
   using (DbConnection cnx =new SqlConnection(sCnx)) {
      using (DbCommand cmd =new SqlCommand(sCmd, cnx as SqlConnection)){
         cnx.Open() ;
         using ( DbDataReader rdr = cmd.ExecuteReader() ) {
```

```
            MyList.DataSource = rdr;
            MyList.DataTextField = "Nom";
            DataBind();
            MyList.EnableViewState = false;
         } // end using rdr.
      } // end using cmd.
   } // end using cnx.
}
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form1" runat="server">
   <asp:ListBox ID="MyList" runat="server" />
   </form>
   </body>
</html>
```

Ainsi nous voyons apparaître trois pratiques permettant l'amélioration des performances lorsque l'on travaille avec des sources de données.

- Soit on limite le nombre de copies des données au strict minimum à chaque chargement de page à l'instar de l'exemple précédent.
- Soit on ne charge les données qu'à la première utilisation d'une page par un utilisateur puis on se sert du *viewstate* pour les stocker.
- Soit on se permet de maintenir une copie en mémoire (au niveau du cache, de la session ou de l'application) à l'instar de l'Exemple 23-54.

Selon le contexte, l'une ou l'autre de ces pratiques peut donner les meilleures performances.

## *Lier déclarativement contrôles et sources de données*

ASP.NET 2.0 introduit plusieurs nouveaux contrôles serveur permettant de se lier déclarativement à une source de donnée. Ils soulagent le développeur de l'écriture du code manipulant l'API ADO.NET nécessaire pour récupérer et modifier des données d'une base. Concrètement, l'exemple suivant est équivalent à l'Exemple 23-59 :

*Exemple 23-60 :*

```
<%@ Page Language=C# %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form2" runat="server">
      <asp:SqlDataSource ID="DataSrc" runat="server" ConnectionString=
          "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION"
          SelectCommand="SELECT * FROM EMPLOYES"
          DataSourceMode="DataReader" />
      <asp:ListBox ID="MyList" DataSourceID="DataSrc" runat="server"
          DataTextField="Nom" EnableViewState="False" />
   </form>
   </body>
</html>
```

On voit que le contrôle source de données de type `SqlDataSource` a besoin de connaître la chaîne de connexion à la base, la commande SQL permettant de récupérer les données et optionnellement le mode de récupération des données : connecté (valeur `DataReader` pour la propriété `DataSourceMode`) ou déconnecté (valeur `DataSet`). La valeur `DataSet` est choisie par défaut. Cet exemple ne montre pas que le contrôle serveur `SqlDataSource` présente de nombreuses autres possibilités telles que la mise en cache des données (propriétés `EnableCaching`, `CacheDuration`, `CacheExpirationPolicy`) ou le filtrage des données.

L'exemple suivant illustre comment exploiter cette notion de filtre afin d'afficher les employés appartenant à un département. Le département est choisi dans une liste déroulante remplie avec la source de données `DataSrc1`. À chaque changement de sélection, cette liste déclenche un évènement *postback*. Du côté serveur, on se sert d'un contrôle de type `ControlParameter` pour récupérer le département sélectionné et le fournir en paramètre à la requête SQL de type `SELECT` de la source de données utilisée pour remplir la liste :

*Exemple 23-61 :*

```
<%@ Page Language=C# %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form1" runat="server">

      <asp:SqlDataSource ID="DataSrc1" runat="server" ConnectionString=
        "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION"
    SelectCommand="SELECT DISTINCT DepId FROM EMPLOYES ORDER BY DepId"/>
      <asp:DropDownList ID="ListDep" DataSourceID="DataSrc1"
        DataTextField="DepId" autopostback="true" runat="server" />

      <asp:SqlDataSource ID="DataSrc2" runat="server" ConnectionString=
        "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION"
        SelectCommand="SELECT * FROM EMPLOYES WHERE DepId=@ParamDepId">
         <SelectParameters>
            <asp:ControlParameter Name="ParamDepId" ControlID="ListDep"
                                  PropertyName="SelectedValue" />
         </SelectParameters>
      </asp:SqlDataSource>
      <asp:ListBox ID="MyList" DataSourceID="DataSrc2" runat="server"
        DataTextField="Nom" />
   </form>
   </body>
</html>
```

L'exécution de cette page ressemble à ceci :

Nous verrons qu'un contrôle serveur source de données peut aussi servir à la modification des données (insertion, mise à jour, destruction). Nous verrons aussi que les contrôles serveur qui exploitent les contrôles source de données rajoutent des facilités telles que la pagination ou le tri des données.


*Figure 23-10 : Filtrage des données d'une source*

## *Les types de source de données*

Il existe principalement deux types de sources de données : les sources de données plates et les sources de données hiérarchiques.

Les sources de données plates sont par exemple les tables d'une base de données relationnelles ou une liste d'objets. Les contrôles serveur qui permettent l'exploitation de telles sources dérivent de la classe `DataSourceControl`. En voici la liste (ces classes sont toutes dans l'espace de noms `System.Web.UI.WebControls`) :

- La classe `SqlDataSource` que nous venons de présenter. Elle peut servir à l'exploitation de tous types de SGBD puisqu'elle supporte les fournisseurs de données *SQL Server*, *ODBC* et *OleDB*.
- La classe `ObjectDataSource` que nous allons décrire dans la prochaine section.

Les sources de données hiérarchiques sont utilisées pour exploiter les données des documents XML. Les contrôles serveur qui permettent l'exploitation de telles sources sont `XmlDataSource` et `SiteMapDataSource`. Ces classes dérivent de la classe `HierarchicalDataSourceControl`. La classe `SiteMapDataSource` permet d'exploiter des documents XML spéciaux représentant l'arborescence d'un site. Elle est décrite page 956. En outre, nous revenons sur la notion de sources de données hiérarchiques un peu plus loin en page 945.

## *La source de données ObjectDataSource*

La classe `ObjectDataSource` permet d'exploiter vos objets de données comme une source de données plate. Contrairement aux autres sources de données qui encouragent une architecture 2-tiers en liant directement les contrôles de données graphiques aux contrôles de source de données, cette classe permet l'élaboration d'une architecture 3-tiers voire N-tiers. Par exemple, supposons que la classe d'objets de données suivante `Employe` représente des employés. Vous pouvez vous servir de la classe `ObjectDataSource` pour lier une collection d'employés à un contrôle de présentation des données tel qu'une `ListBox`. Pour cela, il faut prévoir une classe sans état (i.e sans champs d'instance) présentant une méthode publique qui retourne la collection d'employés. Cette méthode publique, en l'occurrence `ICollection GetData()`, est connue par la source de donnée par la valeur de la propriété `SelectMethod`. Elle est appelée par réflexion. Elle peut être statique ou d'instance. Tout ceci est illustré par les deux listings suivants :

*Exemple 23-62 :*

```
using System.Collections ;
using System.Collections.Generic ;
public class Employe {
   private int m_EmployeID = -1 ;
   private string m_Nom ;
   private string m_Prenom ;
   public int EmployeID{get{return m_EmployeID;}set{m_EmployeID=value;}}
   public string Nom{   get{return m_Nom;}      set{m_Nom = value;} }
   public string Prenom{get{return m_Prenom;}   set{m_Prenom = value;} }
}
public class Helper {
   private static List<Employe> list = new List<Employe>() ;
   static Helper() {
      Employe emp = new Employe() ;
      emp.EmployeID = 1 ; emp.Nom = "Lafleur" ; emp.Prenom = "Léon" ;
      list.Add(emp) ;
      emp = new Employe() ;
      emp.EmployeID = 2 ; emp.Nom = "Dupont" ; emp.Prenom = "Anne" ;
      list.Add(emp) ;
   }
   static public ICollection GetData() { return list ; }
}
```

*Exemple 23-63 :*

```
<%@ Page Language=C# %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form2" runat="server">
      <asp:ObjectDataSource ID="ObjDataSrc"        runat="server"
                            SelectMethod="GetData" TypeName="Helper"/>
      <asp:ListBox ID="MyList"           DataSourceID="ObjDataSrc"
                   DataTextField="Nom" runat="server" />
   </form>
   </body>
</html>
```

## *Utiliser une source de données pour la modification*

La classe `ObjectDataSource` permet aussi la modification des données. Pour cela, il suffit que la classe *helper* intermédiaire présente des méthodes d'insertion, de mise à jour et/ou de destruction. Vous n'avez plus alors qu'à préciser ces méthodes dans la déclaration du contrôle `ObjectDataSource` au moyen des propriétés `InsertMethod`, `UpdateMethod` et `DeleteMethod`. Ces méthodes sont respectivement appelées lorsqu'une des méthodes `IEnumerable Select()`, `int Update()`, `int Insert()` et `int Delete()` est appelée sur le contrôle `ObjectDataSource`. La mise à jour de données est illustrée par l'exemple suivant. En reprenant l'exemple précédent, cette page permet de mettre à jour le nom d'un employé :

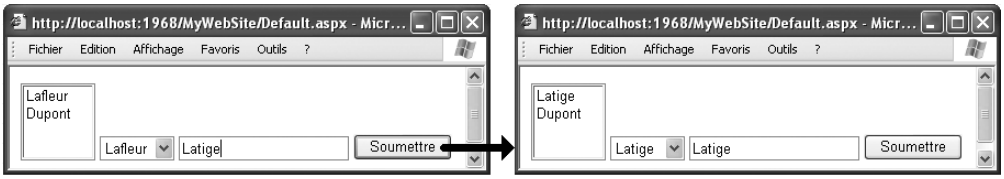


*Figure 23-11 : Mise à jour des données avec ObjectDataSource*

*Exemple 23-64 :*

```
...
public class Employe { ... }
public class Helper {
...
   static public void UpdateName(string oldName, string newName) {
      foreach (Employe emp in list)
         if (emp.Nom == oldName)
            emp.Nom = newName;
   }
}
```

*Exemple 23-65 :*

```
<%@ Page Language=C# %>
<script language=C# runat="server">
   void Btn_Click(Object s, EventArgs e) { ObjDataSrc.Update(); }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form2" runat="server">
      <asp:ObjectDataSource ID="ObjDataSrc" runat="server"
                            TypeName="Helper" SelectMethod="GetData"
                            UpdateMethod="UpdateName" >
          <UpdateParameters>
             <asp:ControlParameter name="oldName" type="String"
                                   controlid="OldName"
                                   propertyname="SelectedValue" />
             <asp:FormParameter Name="newName" formfield="NewName"
                                Type=String />
          </UpdateParameters>
      </asp:ObjectDataSource>
      <asp:ListBox ID="MyList"           DataSourceID="ObjDataSrc"
                   DataTextField="Nom" runat="server" />
      <asp:DropDownList ID="OldName" runat="server" DataTextField="Nom"
                        DataSourceID="ObjDataSrc"  />
      <asp:TextBox ID="NewName" runat="server"/>
      <asp:Button ID="Button1" runat="server" Text="Soumettre"
                  OnClick="Btn_Click" />
```

```
    </form>
    </body>
</html>
```

Notez la façon dont les paramètres en entrée de la méthode `UpdateName()` sont liés avec les contrôles `OldName` et `NewName` au sein d'une section `<UpdateParameters>`. Naturellement, pour la sélection, l'insertion et la destruction vous pouvez pareillement exploiter des sections `<SelectParameters>`, `<InsertParameters>` et `<DeleteParameters>`. Ces sections contiennent les déclarations de contrôles serveurs permettant de récupérer les paramètres en entrée du traitement. Par exemple le contrôle `ControlParameter` permet de récupérer une valeur à partir d'une propriété d'un autre contrôle tandis qu'un contrôle `FormParameter` permet de récupérer une valeur à partir du contenu d'une `TextBox`. Tous ces contrôles dérivent du contrôle `Parameter`. Aussi vous pouvez trouver la liste exhaustive en consultant l'article **Parameter Hierarchy** des **MSDN**.

# *Présentation et édition des données*

ASP.NET 2.0 introduit une nouvelle hiérarchie de contrôles serveurs permettant de présenter et d'éditer des données provenant d'une source de données :

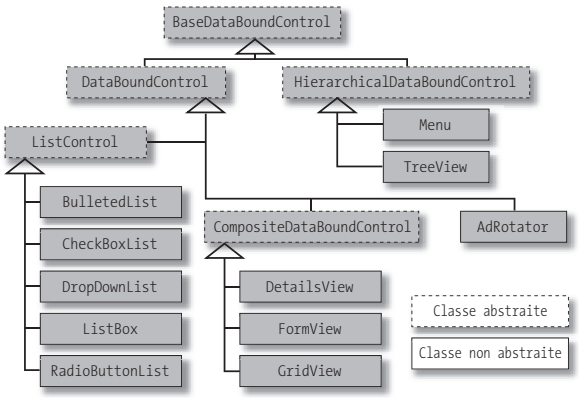


*Figure 23-12 : Hiérarchie de contrôles de présentation et d'édition des données*

Si vous connaissez déjà ASP.NET 1.x, vous remarquerez l'absence du contrôle `DataGrid`. Ce contrôle est toujours présenté par le *framework* .NET 2.0 pour des raisons de compatibilité ascendante. Cependant, il est conseillé d'utiliser son successeur, le contrôle `GridView` qui est plus puissant.

Dans les exemples des sections précédentes nous avons eu l'occasion d'exploiter les contrôles `ListBox` et `DropDownList` ainsi que leur capacité à être liés à une source de données.

Les contrôles `Menu` et `Treeview` qui peuvent être utilisés pour présenter une source de données hiérarchique sont brièvement décrits en page 956 lorsque nous les utilisons pour afficher l'organisation d'un site.

Le contrôle `AdRotator` permet d'afficher aléatoirement une bannière en général publicitaire. Nous vous invitons à consulter les **MSDN** pour en savoir plus à son sujet.

Nous allons surtout nous intéresser aux contrôles GridView, DetailsView et FormView, particulièrement adaptés à la présentation et à l'édition de données d'une source de données plate.

## *Le contrôle GridView*

À l'instar de son prédécesseur le contrôle DataGrid, le contrôle serveur GridView permet de présenter et d'éditer des données dans un tableau. En plus de multiples améliorations fonctionnelles, la puissance de ce contrôle tient surtout dans son efficacité et sa simplicité à travailler avec des sources de données. Voici pour preuve, le code d'une page permettant de remplir une GridView avec les données d'une table d'une base de données :

*Exemple 23-66 :*

```
<%@ Page Language=C# %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form1" runat="server">
      <asp:SqlDataSource ID="DataSrc" runat="server" ConnectionString=
          "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION"
       SelectCommand="SELECT * FROM EMPLOYES" />
      <asp:GridView ID="Grid" DataSourceID="DataSrc" runat="server" />
   </form>
   </body>
</html>
```

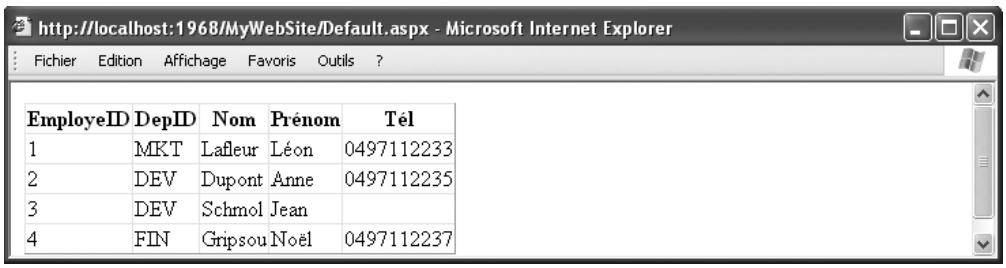


*Figure 23-13 : Utilisation d'une GridView*

Derrière cette simplicité d'utilisation apparente se cache le contrôle serveur le plus complexe d'ASP.NET 2.0. Pour des raisons de place, nous nous bornerons à présenter ses fonctionnalités principales. Aussi n'hésitez pas à consulter les **MSDN** pour plus d'informations.

La propriété bool AllowPaging du contrôle GridView permet de décider si vous souhaitez paginer les données présentées sur plusieurs pages. Par défaut, la pagination se fait sur 10 lignes mais vous pouvez indiquer un autre nombre de lignes avec la propriété PageSize. Vous pouvez obtenir ou fixer l'index de la page affichée avec la propriété int PageIndex. Lorsque plus d'une page est nécessaire à l'affichage des données, le contrôle GridView génère une colonne spécialisée dans la navigation des pages. Vous pouvez personnaliser cette ligne en ayant accès à l'objet de type PagerSettings que vous pouvez récupérer par la propriété GridView.PagerSettings{get;}. Une grosse amélioration de la pagination de GridView par rapport à celle de DataGrid est que seules les données affichées sont stockées dans le *viewstate*.

La propriété `bool AllowSorting` permet de décider si vous souhaitez que l'utilisateur puisse trier les données en fonction du contenu d'une colonne. L'activation de cette possibilité fait en sorte que les entêtes des colonnes soient des hyperliens. Le trie se fait du côté serveur et est retourné au client lorsqu'une telle entête est cliquée.

Vous pouvez très simplement éditer les données d'une ligne, voire détruire une ligne, en vous servant d'une `GridView`. Pour cela, il suffit que la source de donnée sous-jacente supporte ces opérations et que vous positionniez à `true` les propriétés `AutoGenerateEditButton` et `AutoGenerateDeleteButton`. L'exemple suivant illustre ceci :

*Exemple 23-67 :*

```
<%@ Page Language=C# %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form1" runat="server">
      <asp:SqlDataSource ID="DataSrc" runat="server" ConnectionString=
          "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION"
   SelectCommand="SELECT * FROM EMPLOYES"
   UpdateCommand= "UPDATE EMPLOYES SET
                   DepID=@DepID,Nom=@Nom,Prénom=@Prénom,Tél=@Tél
                   WHERE EmployeId=@Original_EmployeId"
   DeleteCommand="DELETE EMPLOYES WHERE EmployeId=@Original_EmployeId"/>
      <asp:GridView ID="MyGrid" DataSourceID="DataSrc" runat="server"
        AutoGenerateEditButton="true"
        AutoGenerateDeleteButton="true"
        DataKeyNames="EmployeId"/>
   </form>
   </body>
</html>
```

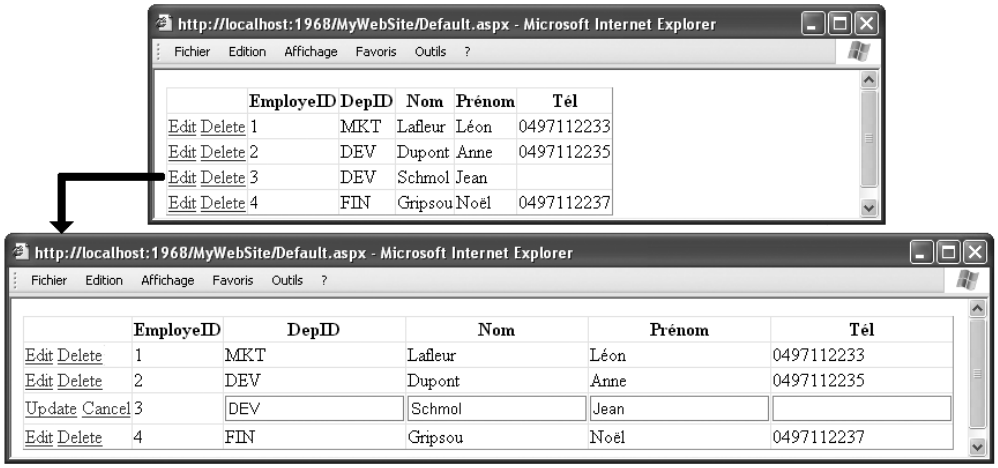


*Figure 23-14 : Édition/Délétion des lignes d'une GridView*

Le contrôle GridView présente des évènements tels que RowDeleting ou RowDeleted permettant d'appeler une méthode côté serveur juste avant ou juste après une modification. L'insertion de lignes nécessite le recours à un contrôle DetailsView aussi nous décrivons cette opération dans la prochaine section.

Vous pouvez agir sur la façon dont les données sont affichées dans la grille en modifiant les colonnes affichées. ASP.NET 2.0 fournit plusieurs styles de colonnes prédéfinies grâce aux types dérivant de la classe DataControlField tels que ButtonField ou BoundField. Cependant, en utilisant le type TemplateField vous pouvez définir exactement le code HTML contenu dans une cellule. Ceci est illustré par l'exemple suivant qui affiche une grille contenant :

- Une colonne de type ButtonField contenant des boutons ayant le nom de l'employé. Lorsqu'un tel bouton est cliqué, la méthode Grid_RowCommand() et invoquée et change le contenu du label Msg.
- Une colonne de type TemplateField qui contient des CheckBox. Une CheckBox est tickée seulement si le nom de l'employé correspondant contient la lettre « o ».
- Une colonne de type BounText contenant les noms des employés.

*Exemple 23-68 :*

```
<%@ Page Language=C# %>
<script language=C# runat="server">
   void Grid_RowCommand(Object sender, GridViewCommandEventArgs e) {
      if (e.CommandName == "Hello") {
         int index = Convert.ToInt32(e.CommandArgument);
         GridViewRow selectedRow = Grid.Rows[index];
         TableCell cell = selectedRow.Cells[2];
         string nom = cell.Text;
         Msg.Text = "Vous aves sélectionné " + nom + ".";
      }
   }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form1" runat="server">
      <asp:SqlDataSource ID="DataSrc" runat="server" ConnectionString=
          "server = localhost ; uid=sa ; pwd ; database = ORGANISATION"
       SelectCommand="SELECT * FROM EMPLOYES" />
      <asp:GridView ID="Grid" DataSourceID="DataSrc" runat="server"
             AutoGenerateColumns=False OnRowCommand="Grid_RowCommand" >
        <Columns>
          <asp:ButtonField DataTextField="Nom" ButtonType=Button
                 HeaderText="Cliquez svp..." CommandName="Hello"/>
          <asp:TemplateField HeaderText="Le nom contient la lettre 'o'">
            <ItemTemplate>
               --<asp:CheckBox runat="server" Enabled=False
               Checked=<%# ((string)Eval("Nom")).Contains("o") %>  />--
            </ItemTemplate>
          </asp:TemplateField>
```

```
            <asp:BoundField DataField="Nom"  HeaderText="Nom" />
          </Columns>
        </asp:GridView>
        <asp:Label ID="Msg" runat="server"></asp:Label>
    </form>
    </body>
</html>
```

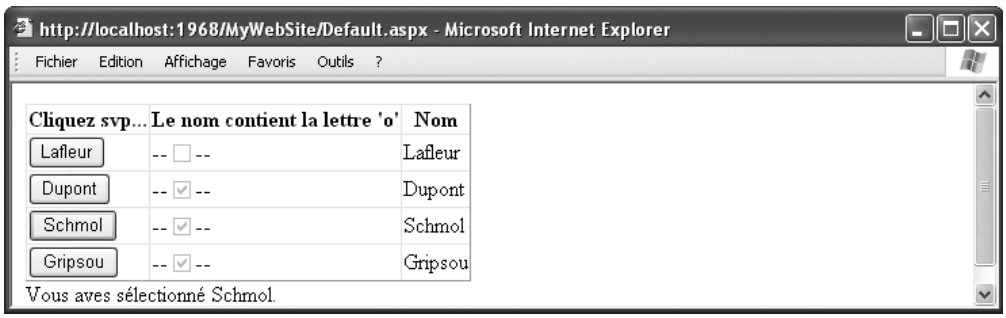


*Figure 23-15 : Personnalisation des colonnes d'une GridView*

## *Les templates*

Dans l'exemple précédent, peut être avez-vous remarqué ce bloc qui ressemble à un bloc de restitution de code :

```
<%# ((string)Eval("Nom")).Contains("o") %>
```

On nomme un tel bloc un *template*. Ce template se remplace par une valeur `false` ou `true` selon que le nom de l'employé de la ligne courante contient un « o » ou non. Les templates constituent un mécanisme permettant de décider comment un contenu est affiché. Les contrôles serveurs de présentation de données qui utilisent les templates sont les contrôles `GridView`, `DetailsView`, `FormView`, `LoginView`, `TreeView`, `Repeater`, `DataList` et `DataGrid`. Contrairement aux blocs de restitution de code, les templates sont évalués au moment où la liaison de données se fait (i.e lors de l'appel à `DataBind()` sur le contrôle). Chaque template est évalué une fois par ligne.

En interne, chaque template induit un contrôle enfant du contrôle de présentation dans lequel il est définit. Ces contrôles enfants sont de type `DataBoundLiteralControl`. Lors de la compilation, ASP.NET ajoute à la classe représentant la page courante une méthode qui est abonnée à l'évènement `DataBinding` d'un tel contrôle. Cet évènement est déclenché pour chaque ligne récupérée durant l'appel à la méthode `DataBind()` du contrôle de présentation parent. Voici la méthode générée pour notre exemple :

```
public void __DataBinding__control6(object sender, EventArgs e) {
    CheckBox box1 = (CheckBox)sender ;
    IDataItemContainer container1 =
        (IDataItemContainer) box1.BindingContainer ;
    box1.Checked = ( (string) DataBinder.Eval(
```

```
      container1.DataItem,"Nom") ).Contains("o") ;
}
```

La propriété BindingContainer du contrôle CheckBox retourne une référence vers la ligne en cours de traitement. En l'occurrence, cette ligne est matérialisée par un contrôle de type GridViewRow qui présente l'interface IDataItemContainer. Enfin, la méthode Eval() de la classe DataBinder est capable par réflexion d'extraire le nom de l'employé courant.

Bien qu'ils ne soient pas présents dans la nouvelle hiérarchie de contrôle de présentation de données d'ASP.NET 2.0, nous signalons la possibilité d'exploiter deux types de contrôles particulièrement adaptés à l'utilisation des templates : les contrôles Repeater et DataList. Leur utilisation est illustrée par l'exemple suivant :

*Exemple 23-69 :*

```
<%@ Page Language=C# %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form1" runat="server">
      <asp:SqlDataSource ID="DataSrc" runat="server" ConnectionString=
          "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION"
       SelectCommand="SELECT EmployeId, Nom, Tél FROM EMPLOYES" />
      Repeater:<br/>
      <asp:Repeater ID="MyRepeater" DataSourceID="DataSrc" runat="server">
          <ItemTemplate>
 Nom:<%# Eval("Nom") %> Téléphone:<%# Eval("Tél") %><br/>
          </ItemTemplate>
      </asp:Repeater>
      <br/>DataList:<br/>
      <asp:DataList ID="MyDataList" DataSourceID="DataSrc" runat="server"
                    RepeatColumns="3" RepeatDirection="Vertical">
          <ItemTemplate>
 Nom:<%# Eval("Nom") %> Téléphone:<%# Eval("Tél") %><br/>
          </ItemTemplate>
       </asp:DataList>
   </form>
   </body>
</html>
```

Il existe trois autres types de templates en plus du template Eval :

```
<%# Eval( "Colonne|Propriété|Champ" [,"format"] ) %>
<%# Bind( "Colonne|Propriété|Champ" [,"format"]) %>
<%# XPath( "Expression-XPath" [,"format"]) %>
<%# XPathSelect( "Expression-XPath") %>
```

La chaîne de caractères optionnelle format permet de mettre en forme la chaîne de caractères qui représente la donnée évaluée (voir page 375 pour plus de détails concernant la mise en forme d'une chaîne de caractères).

Un template de type Bind s'utilise de la même manière qu'un template de type Eval. Cependant, un tel template permet aussi de récupérer les informations saisies par l'utilisateur et de les communiquer au contrôle de présentation/édition de données parent. Typiquement, les templates


*Figure 23-16 : Utilisation des contrôles Repeater et DataList*

de type `Bind` s'utilisent lorsque vous activez le mode édition sur un contrôle de présentation. Un tel template est exploité un peu plus loin dans l'Exemple 23-73 lorsque nous exposons le mode édition d'un contrôle de type `FormView`. Plus d'informations sur les templates `Bind` sont disponibles dans l'article **Two-Way Data Binding Syntax** des **MSDN**.

Les templates de type `XPath` et `XPathSelect` s'utilisent sur des structures de données hiérarchiques et sont décrits un peu plus loin.

## *Le contrôle DetailsView*

Le contrôle `DetailsView` permet de présenter une vue détaillée de chaque ligne. Comme illustré par l'exemple suivant, il est aisé d'utiliser une `DetailsView` pour naviguer dans les différentes lignes d'une source de données plate :

*Exemple 23-70 :*

```
<%@ Page Language=C# %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form1" runat="server">
      <asp:SqlDataSource ID="DataSrc" runat="server" ConnectionString=
          "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION"
       SelectCommand="SELECT * FROM EMPLOYES" />
      <asp:DetailsView ID="Details" DataSourceID="DataSrc" runat="server"
                       AllowPaging="True" >
         <PagerSettings Mode="NextPreviousFirstLast" />
      </asp:DetailsView>
   </form>
   </body>
</html>
```

Le type de vue du contrôle `DetailsView` est complémentaire à celui de contrôle `GridView`. Aussi, il est courant d'utiliser une symbiose de ces deux contrôles pour obtenir ce que l'on appelle une *vue maîtresse détaillée* d'une source de données plate (*master details view* en anglais). Pour obtenir une telle vue, il est nécessaire d'activer la sélection d'une ligne sur la `GridView` au moyen de la propriété `AutoGenerateSelectButton`. En outre, il faut définir deux sources de données : une


*Figure 23-17 : Utilisation d'une DetailsView*

exploitée par la `GridView` qui présente l'ensemble des lignes et une exploitée par la `DetailsView` qui présente la ligne couramment sélectionnée. Cette deuxième source de données utilise la propriété `FilterExpression` pour déterminer quelle est la ligne couramment sélectionnée :

*Exemple 23-71 :*

```
<%@ Page Language=C# %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form1" runat="server">
      <asp:SqlDataSource ID="DataSrc1" runat="server" ConnectionString=
          "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION"
          SelectCommand="SELECT * FROM EMPLOYES" />
      <asp:GridView ID="MyGrid" DataSourceID="DataSrc1" runat="server"
                    DataKeyNames="EmployeId" SelectedIndex="0"
                    AutoGenerateSelectButton=true />

      <asp:SqlDataSource ID="DataSrc2" runat="server" ConnectionString=
          "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION"
          SelectCommand="SELECT * FROM EMPLOYES"
          FilterExpression="EmployeId={0}">
         <FilterParameters>
            <asp:ControlParameter Name="EmployeId" ControlID="MyGrid"
                                  PropertyName="SelectedValue" />
         </FilterParameters>
      </asp:SqlDataSource>
      <asp:DetailsView ID="MyDetails" DataSourceID="DataSrc2"
                       runat="server" DataKeyNames="EmployeId" />
   </form>
   </body>
</html>
```

Une vue maîtresse détaillée représente l'alternative optimale pour l'édition et l'insertion des données de chaque ligne. Comme le montre l'exemple suivant, pour obtenir une telle vue il suffit d'activer ces possibilités d'insertion et d'édition sur la source de données de la `DetailsView` ainsi que sur la `DetailsView` elle-même :

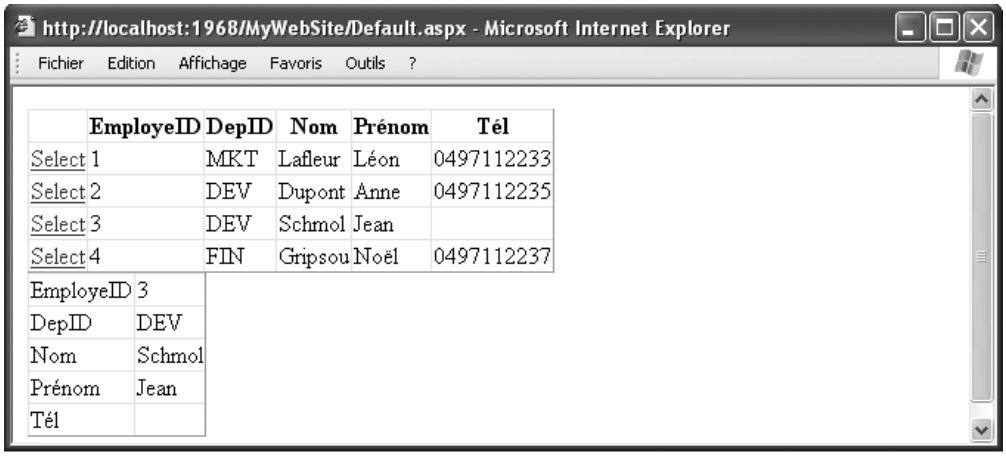


*Figure 23-18 : Vue maîtresse détaillée*

*Exemple 23-72 :*

```
<%@ Page Language=C# %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form1" runat="server">
      <asp:SqlDataSource ID="DataSrc1" runat="server" ConnectionString=
          "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION"
          SelectCommand="SELECT * FROM EMPLOYES"
          />
      <asp:GridView ID="MyGrid" DataSourceID="DataSrc1" runat="server"
                    DataKeyNames="EmployeId" SelectedIndex="0"
                    AutoGenerateSelectButton=true />

      <asp:SqlDataSource ID="DataSrc2" runat="server" ConnectionString=
          "server = localhost ; uid=sa ; pwd ; database = ORGANISATION"
   SelectCommand="SELECT * FROM EMPLOYES"
   UpdateCommand= "UPDATE EMPLOYES SET
                   DepID=@DepID,Nom=@Nom,Prénom=@Prénom,Tél=@Tél
                   WHERE EmployeId=@Original_EmployeId"
   DeleteCommand="DELETE EMPLOYES WHERE EmployeId=@Original_EmployeId"
   InsertCommand="INSERT INTO EMPLOYES (DepID,Nom,Prénom,Tél)
                  VALUES(@DepID,@Nom,@Prénom,@Tél)"
   FilterExpression="EmployeId={0}"
   OnDeleted=OnChgData  OnInserted=OnChgData  OnUpdated=OnChgData >
         <FilterParameters>
            <asp:ControlParameter Name="EmployeId" ControlID="MyGrid"
                                  PropertyName="SelectedValue" />
         </FilterParameters>
      </asp:SqlDataSource>
```

```
      <asp:DetailsView ID="MyDetails" DataSourceID="DataSrc2"
                       runat="server" DataKeyNames="EmployeId"
                       AutoGenerateDeleteButton="True"
                       AutoGenerateEditButton="True"
                       AutoGenerateInsertButton="True"/>
   </form>
   </body>
</html>
<script language=C# runat="server">
   void OnChgData(Object sender, EventArgs e) { MyGrid.DataBind() ; }
</script>
```

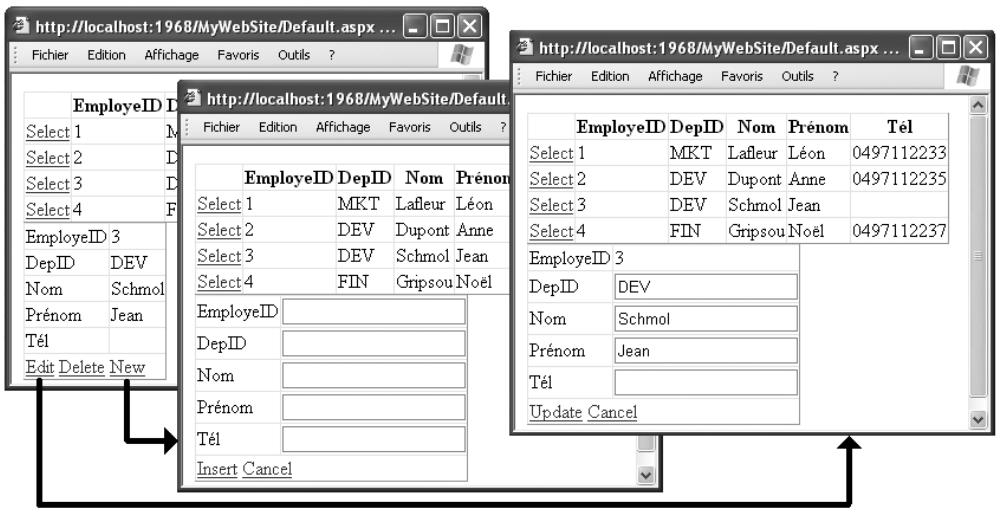


*Figure 23-19 : Édition/Insertion avec vue Maîtresse/Détaillée*

Notez l'obligation d'invoquer la méthode `DataBind()` sur notre `GridView` à chaque fois qu'une données est mise à jour par l'intermédiaire du `DetailsView`. Sans cette manipulation, les données de la `GridView` ne seraient pas mises à jour automatiquement.

Le contrôle `DetailsView` présente de nombreuses propriétés de configuration ainsi que plusieurs évènements qui peuvent vous être utile. Nous vous invitons à consulter les **MSDN** pour plus d'informations. En page 688 nous exposons comment obtenir ce type de vue dans le cadre d'une fenêtre *Windows Forms*.

## *Le contrôle FormView*

Le contrôle serveur `FormView` est similaire au contrôle `DetailsView` à ceci prés qu'il est particulièrement adapté à la personnalisation de la présentation et de l'édition des données d'une ligne. Contrairement au contrôle `DetailsView`, le contrôle `FormView` ne fournit pas de code par défaut pour présenter les données. Il est nécessaire de fournir votre propre code dans des sections `<ItemTemplate>`, `<EditItemTemplate>` et `<InsertItemTemplate>`. Comme le montre l'exemple suivant, l'utilisation des templates devient particulièrement pratique au sein de ces sections :

*Exemple 23-73 :*

```
<%@ Page Language=C# %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form1" runat="server">
      <asp:SqlDataSource ID="DataSrc" runat="server" ConnectionString=
          "server = localhost ; uid=sa ; pwd = ; database = ORGANISATION"
       SelectCommand="SELECT EmployeId, Nom, Tél FROM EMPLOYES"
       UpdateCommand=
    "UPDATE EMPLOYES SET Tél=@Tél WHERE EmployeId=@Original_EmployeId"/>
      <asp:FormView ID="Details" DataSourceID="DataSrc" runat="server"
                    DataKeyNames="EmployeId"  AllowPaging="True" >
         <ItemTemplate>
           Nom : <%# Eval("Nom") %> <br/>
           Téléphone : <%# Eval("Tél") %> <br/>
           <asp:Button ID="BtnEdit" runat="server" CommandName="Edit"
           Text="Edition du numero de téléphone"/>
         </ItemTemplate>
         <EditItemTemplate>
           Nom : <%# Eval("Nom") %> <br/>
           Téléphone : <asp:TextBox ID="EditPhone" runat="server"
                                    Text=<%# Bind("Tél") %> /> <br/>
           <asp:Button ID="BtnUpdate" runat="server" CommandName="Update"
             Text="Mise à jour du numero de téléphone."/>
           <asp:Button ID="BtnCancel" runat="server" CommandName="Cancel"
             Text="Annuler la mise à jour."/>
         </EditItemTemplate>
      </asp:FormView>
   </form>
   </body>
</html>
```

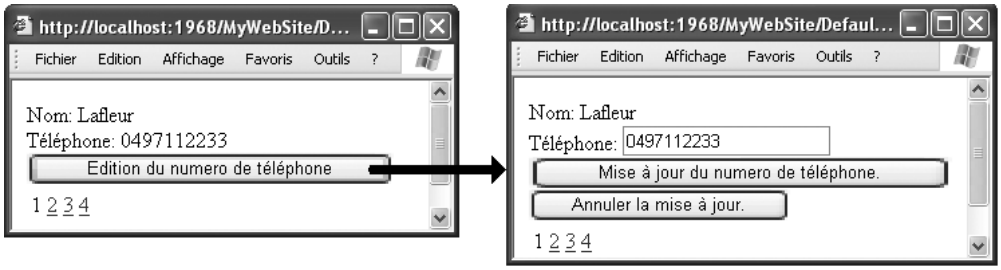


*Figure 23-20 : Utilisation du contrôle FormView*

## *Présenter un document XML*

Nous ne parlons pas ici des contrôles de présentation de données hiérarchiques `TreeView` et `Menu`. Sachez qu'ils sont brièvement couverts en page 955.

Nous portons plutôt notre attention sur le fait qu'une source de données hiérarchique peut être exploitée par des contrôles de présentation originalement destinés à être liés à des sources de données plates. Dans ce cas, un tel contrôle présente le premier niveau de hiérarchie des données. L'avantage est que vous pouvez utiliser un autre contrôle de présentation de données imbriqué dans le premier afin d'accéder au second niveau de hiérarchie. Par exemple considérez le document XML suivant qui présente des livres, et pour chaque livre, la liste des chapitres :

*Exemple 23-74 :* *books.xml*

```
<?xml version="1.0" encoding="utf-8" ?>
<bookstore>
   <book title="L'insoutenable légèreté de l'être"
         author="Kundera" publicationdate="1985" >
      <chapter name="La légèreté et la pesanteur"/>
      <chapter name="L'âme et le corps"/>
   </book>
   <book title="Lanzarote"
         author="Michel Houellebecq" publicationdate="2002" >
      <chapter name="Lanzarote"/>
      <chapter name="Compte rendu de mission"/>
   </book>
   <book title="Trilogie New_Yorkaise"
         author="Paul Auster" publicationdate="2002" >
      <chapter name="Cité de verre"/>
      <chapter name="Revenants"/>
      <chapter name="La chambre dérobée"/>
   </book>
</bookstore>
```

La page suivante permet de présenter les livres publiés après l'an 2000 ainsi que pour chaque livre, la liste des chapitres. Remarquez l'utilisation des templates de type `XPath` et `XPathSelect` pour sélectionner respectivement un attribut et plusieurs nœuds. Notez aussi l'utilisation de deux contrôles `Repeater` imbriqués permettant de présenter deux niveaux de hiérarchie. Enfin, notez l'utilisation d'une requête `XPath` au niveau de la source de données pour ne sélectionner que les livres publiés après l'an 2000 :

*Exemple 23-75 :*

```
<%@ Page Language=C# %>
<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
   <form id="Form1" runat="server">
      Liste des livres publiés aprés 2000:<br/>
      <asp:XmlDataSource id="MySrc" DataFile="~/App_Data/books.xml"
                         XPath="/bookstore/book[@publicationdate>2000]"
                         runat="server"/>
      <asp:Repeater id="ListBook" DataSourceId="MySrc" runat="server">
         <ItemTemplate>
  Titre : <%# XPath("@title") %>
  Auteur:<%# XPath("@author") %>
  Date de publication:<%# XPath("@publicationdate")%><br/>
```

```
            <asp:Repeater id="ListChapter" runat="server"
                          DataSource='<%# XPathSelect("chapter") %>'  >
               <ItemTemplate>- <%# XPath("@name")%><br/></ItemTemplate>
            </asp:Repeater>
         </ItemTemplate>
      </asp:Repeater>
   </form>
   </body>
</html>
```

Voici le contenu de cette page :

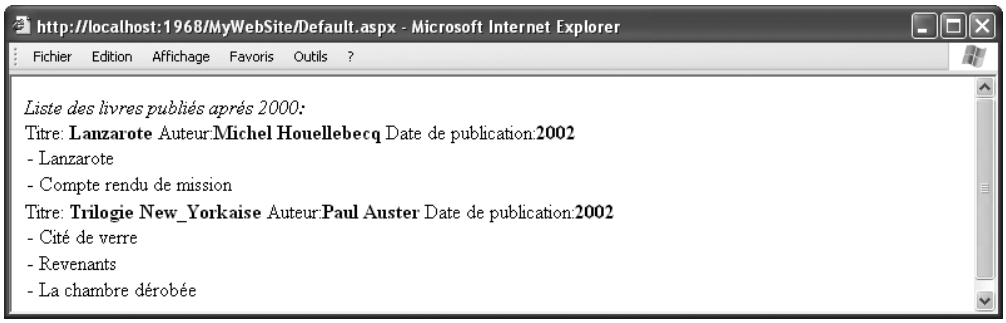


*Figure 23-21 : Exploitation d'un document XML*

# *Master pages*

## *Master page et page de contenu*

Pour garantir la cohérence visuelle d'un site, il est courant que toutes ses pages présentent des parties similaires telles que l'entête ou le bas de page. En ASP.NET 1.x, il faut avoir recours aux contrôles utilisateurs pour pouvoir réutiliser des parties similaires. Cette approche a cependant l'inconvénient d'obliger le développeur à créer plusieurs contrôles utilisateur, par exemple un pour l'entête et un pour le bas de page. En outre, il faut copier la déclaration de chacun de ces contrôles dans chaque page du site, ce qui entraîne des problèmes de mise à jour.

ASP.NET 2.0 présente la notion de *master page* qui permet de réutiliser simplement un modèle de design de page sur les pages d'un site. L'application d'un tel modèle est illustrée par les deux pages de la figure suivante :

Un modèle de design de page (i.e une master page) se présente sous la forme d'un fichier d'extension `.master` qui commence par une directive `<%@ master%>`. Le contenu d'un tel fichier est similaire à celui d'une page `.aspx` au détail prés qu'il peut contenir des contrôles de type `<asp:ContentPlaceHolder>`. Un tel contrôle définit un emplacement où les pages du site pourront placer leur propre contenu. Voici le code de la master page de l'exemple de notre figure :

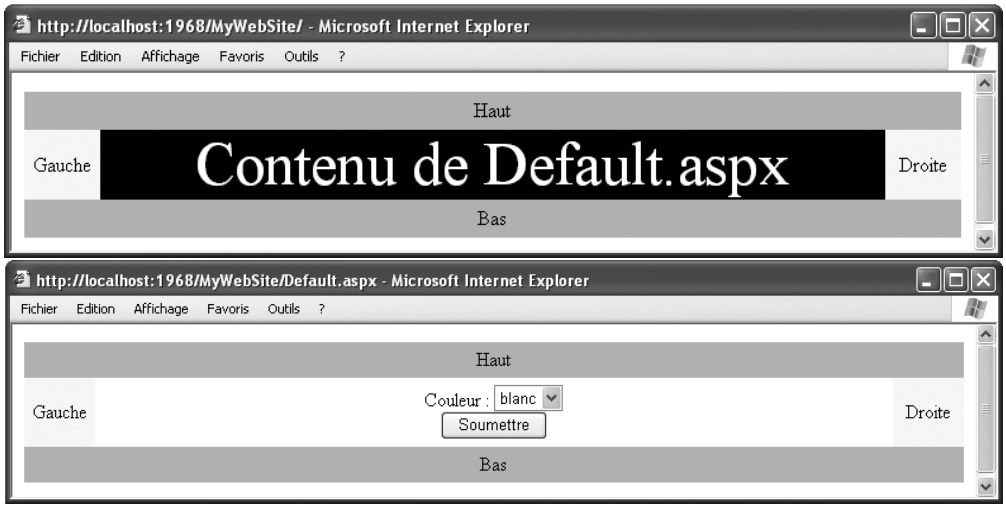


*Figure 23-22 : Exemple d'une master page*

*Exemple 23-76 :* *MyMasterPage.master*

```
<%@ master language="C#" %>
<html><body>
<form id="Form1" runat="server">
   <table id="Tbl1" width="100%" height="30px"
                    cellspacing="0" cellpadding="0">
      <tr> <td width="100%" align="center" bgcolor="silver">Haut</td> </tr>
   </table>
   <table id="Tbl2" width="100%" height="50%"
                    cellspacing="0" cellpadding="0">
      <tr>
         <td width="60px" align="center" bgcolor="yellow">Gauche</td>
         <td align="center">
            <asp:contentplaceholder id="MyContentPH" runat="Server">
            </asp:contentplaceholder>
         </td>
         <td width="60px" align="center" bgcolor="yellow">Droite</td>
      </tr>
   </table>
   <table id="Tbl3"  width="100%" height="30px"
                     cellspacing="0" cellpadding="0">
      <tr> <td width="100%" align="center" bgcolor="silver">Bas</td> </tr>
   </table>
</form></body></html>
```

Voici le code d'une page `.aspx` qui exploite la master page `MyMasterPage.master`. On dit qu'une telle page est une *page de contenu* :

*Exemple 23-77 :* *Default.aspx*

```
<%@ Page Language="C#" MasterPageFile="~/MyMasterPage.master"  %>
<asp:content id="MyContent" contentplaceholderid="MyContentPH"
             runat="server" >
   <asp:label id="lbl"          Height="100%"     Width="100%"
              BackColor="Black" ForeColor="White" runat="server"
              Font-Size="XX-Large">Contenu de Default.aspx</asp:label>
</asp:content>
```

On remarque que l'on spécifie la master page grâce à la sous directive `MasterPageFile` de la directive `<%@ Page%>`. Une page de contenu ne peut avoir pour contrôles racines que des contrôles de type `<asp:Content>`. La propriété `ContentPlaceHolderId` d'un tel contrôle est obligatoire. Elle doit être initialisée avec l'identificateur d'un contrôle de type `<asp:ContentPlaceHolder>` de la master page. Ainsi, le contenu défini au sein d'un contrôle de type `<asp:Content>` sera injecté à l'exécution dans la master page à la place du contrôle `<asp:ContentPlaceHolder>` associé.

Sachez qu'une page de contenu n'associe pas obligatoirement un contrôle `<asp:Content>` à chaque contrôle `<asp:ContentPlaceHolder>` de sa master page. En outre, la définition d'une master page peut contenir des contrôles `<asp:Content>` associés à ses propres contrôles `<asp:ContentPlaceHolder>`. Le contenu d'un tel contrôle `<asp:Content>` sera injecté à la place du contrôle `<asp:ContentPlaceHolder>` associé par défaut lorsque la page de contenu ne fournit pas un contrôle `<asp:Content>` adéquat.

Une particularité intéressante des master pages est que *Visual Studio .NET 2005* est capable d'afficher en mode lecture seule le modèle de design lors de l'édition d'une page qui exploite une master page.

À l'exécution, une master page est matérialisée par un objet, instance d'une classe dérivée de la classe `System.Web.UI.MasterPage`. Cette classe dérive elle-même de la classe `UserControl`. Une page de contenu se compile comme toute page `.aspx`. Cependant, l'accesseur `get` de la propriété `MasterPage Master{get;set;}` d'une page de contenu retourne à l'exécution une référence vers l'objet qui représente la master page associée. Le moteur ASP.NET 2.0 s'occupe de fabriquer l'arborescence de contrôles de la page rendue. Pour cela, il fusionne l'arborescence de contrôles définie dans le contrôle utilisateur master page et les arborescences de contrôles définies dans chaque contrôle `<asp:Content>` de la page de contenu.

Une conséquence du fait qu'ASP.NET implémente la notion de master page sous la forme de contrôles utilisateurs est que les techniques de cache de pages que nous avons déjà vu sont exploitables dans une page de contenu.

## *Master pages encapsulées*

Vous avez la possibilité d'encapsuler une master page dans une autre master page. Cette possibilité est illustrée par la page ci-dessous :

Pour reproduire cette page, il suffit de construire dans notre solution une nouvelle master page que nous nommons `MyMasterPageNested.master` qui est associée à notre master page initiale `MyMasterPage.aspx` grâce à une sous directive `MasterPageFile` de la directive `<%@ master%>`. À l'instar d'une page de contenu, une master page encapsulée ne peut contenir que des contrôles `<asp:Content>`. En revanche ces contrôle peuvent eux-mêmes contenir des contrôles `<asp:ContentPlaceHolder>`.


*Figure 23-23 : Exemple d'une master page encapsulée*

*Exemple 23-78 :* *MyMasterPageNested.master*

```
<%@ master language="C#" MasterPageFile="~/MyMasterPage.master"%>
<asp:Content ContentPlaceHolderID=MyContentPH runat="server">
   <table id="Tbl1" width="100%" height="30px"
                    cellspacing="0" cellpadding="0">
      <tr> <td width="100%" align="center"  bgcolor="olive">Haut</td> </tr>
   </table>
   <table id="Tbl2" width="100%" height="50%"
                    cellspacing="0" cellpadding="0">
      <tr>
         <td width="60px" align="center" bgcolor="red">Gauche</td>
         <td align="center">
            <asp:contentplaceholder id="MyContentPH2" runat="Server">
            </asp:contentplaceholder>
         </td>
         <td width="60px" align="center" bgcolor="red">Droite</td>
      </tr>
   </table>
   <table id="Tbl3" width="100%" height="30px"
                    cellspacing="0" cellpadding="0">
      <tr> <td width="100%" align="center" bgcolor="olive">Bas</td> </tr>
   </table>
</asp:Content>
```

Une page de contenu peut être associée à une master page encapsulée de la même manière que si cette dernière n'était pas encapsulée :

*Exemple 23-79 :* *Default.aspx*

```
<%@ Page Language="C#" MasterPageFile="~/MyMasterPageNested.master"  %>
<asp:content id="MyContent" contentplaceholderid="MyContentPH2"
             runat="server" >
   <asp:label id="lbl"          Height="100%"     Width="100%"
              BackColor="Black" ForeColor="White" runat="server"
```

```
                Font-Size="XX-Large">Contenu de Default.aspx</asp:label>
</asp:content>
```

Notez que *Visual Studio 2005* ne sait pas passer en mode design de page dés lors que vous utilisez des master pages encapsulées.

## *Configuration de master page*

Dans des gros sites avec un grand nombre de pages, le simple fait de devoir mettre une sous directive `MasterPageFile` dans chaque page de contenu peut être problématique pour des raisons de maintenance. Aussi, vous pouvez avoir recours à la propriété `masterPageFile` de l'élément de configuration `<pages>` afin de préciser à ASP.NET que toutes les pages impactées par cet élément doivent être associée avec une certaine master page. Seules seront impactées les pages de contenu qui n'ont pas de sous directive `MasterPageFile`.

*Exemple :* *`Web.Config`*

```
<?xml version="1.0"?>
<configuration
 xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <system.web>
      <pages masterPageFile="~/MyMasterPage.master"/>
...
```

En plus de ces deux techniques d'association statique (i.e à la compilation) de master page à des pages de contenu vous pouvez dynamiquement (i.e à l'exécution) choisir avec quelle master page une page de contenu doit être rendue. Pour cela il suffit de positionner la propriété `string MasterPageFile{get;set;}` d'une page de contenu dans l'évènement `OnPreInit()`.

*Exemple 23-80 :* *`Default.aspx`*

```
<%@ Page Language="C#" %>
<asp:content id="MyContent" contentplaceholderid="MyContentPH"
             runat="server" >
   <script language=C# runat="server">
      protected override void OnPreInit(EventArgs e) {
         this.MasterPageFile ="~/MyMasterPage2.master";
         base.OnPreInit(e);
      }
   </script>
   <asp:label id="lbl"          Height="100%"     Width="100%"
              BackColor="Black" ForeColor="White" runat="server"
              Font-Size="XX-Large">Contenu de Default.aspx</asp:label>
</asp:content>
```

Cette technique peut se révéler très puissante pour certains gros sites mais nous vous conseillons de l'utiliser avec parcimonie du fait des contraintes qu'elle induit. Notamment, elle vous rend responsable de la maintenance des correspondances entre identificateurs des contrôles `<asp:ContentPlaceHolder>` car elle empêche *de facto* toute vérification de la part du compilateur.

## *Accéder à une master page à partir d'une page de contenu*

Il peut être utile de pouvoir accéder aux contrôles de la master page directement à partir du code d'une page de contenu. Pour illustrer cette possibilité, modifions notre master page `MyMasterPage` de façon à introduire un contrôle serveur qui représente la cellule d'entête :

*Exemple 23-81 : MyMasterPage.master*

```
<%@ master language="C#" %>
   <script language=C# runat="server">
      public HtmlTableCell CellHeader {
         get { return _CellHeader ; }
         set { _CellHeader = value ; }
      }
   </script>
<html>
<body>
<form id="Form1" runat="server">
   <table id="Tbl1" width="100%" height="30px"
                   cellspacing="0" cellpadding="0" >
      <tr>
         <td id="_CellHeader" runat="server" width="100%" align="center"
             bgcolor="silver" >Haut</td>
      </tr>
   </table>
...
```

D'après la façon dont les master pages sont compilées par ASP.NET, il est clair que nous pouvons avoir accès à ce contrôle `_CellHeader` à partir du code de notre page de contenu comme ceci :

*Exemple 23-82 : Default.aspx*

```
<%@ Page Language="C#" MasterPageFile="~/MyMasterPage.master" %>
<asp:content id="MyContent" contentplaceholderid="MyContentPH"
             runat="server" >
   <script language=C# runat="server">
      void Page_Load(object sender, EventArgs e) {
         HtmlTableCell cell =
              Master.FindControl("_CellHeader") as HtmlTableCell;
         cell.BgColor = "white";
      }
   </script>
   <asp:label id="lbl"          Height="100%"     Width="100%"
              BackColor="Black" ForeColor="White" runat="server"
              Font-Size="XX-Large">Contenu de Default.aspx</asp:label>
</asp:content>
```

Comprenez bien que dans cet exemple, nous créons un lien tardif entre le code de la méthode `Page_Load()` de la page de contenu et l'objet représentant le contrôle responsable du rendu de la cellule entête dans la master page. Les liens tardifs introduisent de la souplesse au détriment des performances et des vérifications statiques.

Dans le cas où la master page est définie statiquement dans une sous directive `MasterPageFile` de la page de contenu nous n'avons pas besoin de la souplesse des liens tardifs. En effet, dans ce cas nous connaissons dés la compilation le type de master page utilisé. Aussi, vous pouvez utiliser une directive `<%@ MasterType%>` dans la page de contenu afin de spécifier au compilateur quel type de master page vous souhaitez utiliser. Cela peut se faire au moyen d'une sous directive `VirtualPath` qui spécifie le nom du fichier contenant la master page ou au moyen d'une sous directive `TypeName` qui nomme directement la classe représentant la master page. Le compilateur fera alors en sorte que le type de la propriété `Master{get;set;}` de la classe représentant la page de contenu soit le bon. Vous pourrez alors utiliser des liens précoces sur cette master page.

Pour exploiter au mieux cette possibilité, il est conseillé de fournir des propriétés publiques sur la classe de master page donnant accès aux contrôles susceptibles d'être exploités à partir de la page de contenu (à l'instar de ce que nous avons fait avec la propriété `CellHeader` définie dans notre master page) :

*Exemple 23-83 :* *`Default.aspx`*

```
<%@ Page Language="C#" MasterPageFile="~/MyMasterPage.master" %>
<%@ MasterType VirtualPath="~/MyMasterPage.master" %>
<asp:content id="MyContent" contentplaceholderid="MyContentPH"
             runat="server" >
   <script language=C# runat="server">
      void Page_Load(object sender, EventArgs e) {
         Master.CellHeader.BgColor = "white";
      }
   </script>
   <asp:label id="lbl"          Height="100%"     Width="100%"
              BackColor="Black" ForeColor="White" runat="server"
              Font-Size="XX-Large">Contenu de Default.aspx</asp:label>
</asp:content>
```

# *Internationaliser une application ASP.NET 2.0*

Avons d'aborder la présente section, il faut avoir assimilé les concepts d'internationalisation d'une application .NET qui font l'objet de la section 2 « Internationalisation et assemblages satellites », page 34.

Dans le cas d'une application *ASP.NET 2.0*, *Visual Studio 2005* présente des facilités pour éditer plusieurs versions localisées d'une même page web. Lorsque vous sélectionnez une page web XXX.aspx en mode *design*, vous pouvez sélectionner le menu *Tools* ▶ *Generate Local Resources.* Un fichier ressource XXX.aspx.resx est alors ajouté dans le répertoire /App_LocalResources de l'application web. Ce fichier contient les valeurs relatives à la culture invariante des ressources de cette page web. Vous pouvez alors ajouter des fichiers ressources relatifs à d'autres cultures en faisant : répertoire App_LocalResources ▶ *Add New Item* ▶ *Assembly Ressource File* ▶ XXX.yy-ZZ.resx. En éditant ce fichier, vous pouvez modifier les valeurs des ressources de la version de la page relative à la culture yy-ZZ. Notez que tout ceci s'applique aussi aux contrôles utilisateurs (i.e aux fichiers d'extension .ascx) et au master pages (i.e aux fichiers d'extension .master).

Il est intéressant de remarquer que le code ASP.NET d'une page est différent selon que celle-ci est « internationalisée » ou pas. Notamment, les valeurs des propriétés « internatio-

nalisables » des contrôles serveurs ne sont plus présentes puisqu'elles sont contenues dans les fichiers ressources associés à la page. En outre, chaque contrôle contient un attribut `meta:resourcekey="XXX"` où `XXX` désigne une chaîne de caractères utilisée pour identifier le contrôle au niveau des fichiers ressources.

La valeur de la propriété CurrentUICulture du thread qui traite une requête web est déterminée durant l'appel à la méthode virtuelle Page.InitializeCulture() au début du cycle de vie de la requête. Par défaut, la culture du client est envoyée par le navigateur à chaque requête et elle adoptée par ASP.NET 2.0 pour cette requête. Vous pouvez changer ce mode de fonctionnement en réécrivant cette méthode InitializeCulture().

Pour tester une page selon différentes cultures soit :

- Vous modifiez la culture par défaut de votre navigateur. Sous *Internet Explorer* il suffit d'aller dans le menu *Outils* ▶*Options internet* ▶*Général* ▶ *Langues* ▶ *fournir une liste de culture,* celles-ci seront traitées par ordre de priorité par ASP.NET.
- Vous positionnez la propriété UICulture de la page à la culture souhaitée. Attention à bien remettre cette propriété à la valeur Auto une fois la page testée.

Vous pouvez fournir une culture par défaut à toutes les pages situées (récursivement) dans un répertoire au moyen des attributs culture et uiCulture de la section <globalization> du fichier `Web.Config`. Cette culture sera choisie pour traiter une requête qui ne spécifie pas de culture ou pour traiter une requête dont la culture n'est pas supportée.

Enfin, notez que vous pouvez placer des fichiers de ressources globaux à votre application web dans le répertoire /App_GlobalResources.

# *Aides à la navigation dans un site*

ASP.NET 2.0 présente une infrastructure extensible permettant d'insérer dans vos pages des contrôles d'aide à la navigation dans un site. L'aide à la navigation peut se faire principalement avec trois types de contrôles : les `TreeView`, les `Menu` et les `SiteMapPath`. Ils sont illustrés par les trois captures d'écran suivantes :

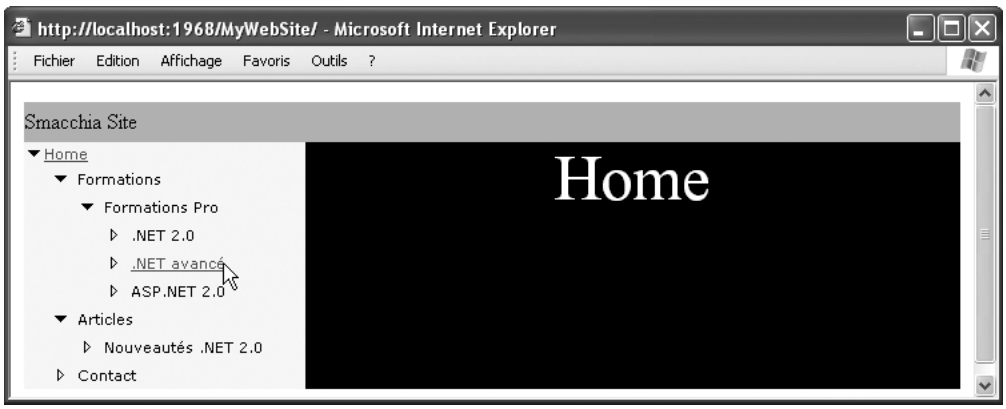


*Figure 23-24 : Navigation avec un TreeView*


*Figure 23-25 : Navigation avec un Menu*

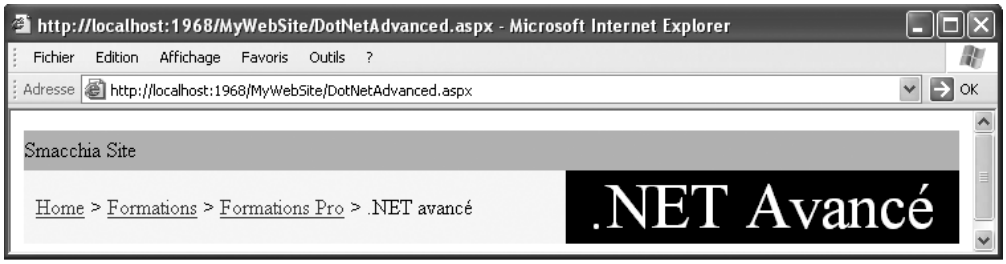


*Figure 23-26 : Rappel du chemin menant à la page courante avec un SiteMapPath*

Pour bénéficier de l'aide à la navigation, il faut tout d'abord ajouter un fichier d'extension `.SiteMap` à votre site web. Ce fichier XML permet de décrire l'organisation du site grâce à une arborescence XML où chaque nœud représente une page :

*Exemple 23-84 :* *App.SiteMap*

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap>
  <siteMapNode title="Home" url="~/Default.aspx">
    <siteMapNode title="Formations" url="~/Formations.aspx">
      <siteMapNode title="Formations Pro" url="~/FormationsPro.aspx">
        <siteMapNode title=".NET 2.0" url="~/DotNet2.aspx"/>
        <siteMapNode title=".NET avancé" url="~/DotNetAdvanced.aspx"/>
        <siteMapNode title="ASP.NET 2.0" url="~/ASPDotNet2.aspx"/>
      </siteMapNode>
    </siteMapNode>
    <siteMapNode title="Articles" url="~/Articles.aspx">
      <siteMapNode title="Nouveautés .NET2.0" url="~/DotNet2News.aspx"/>
    </siteMapNode>
    <siteMapNode title="Contact" url="~/Contact.aspx"/>
  </siteMapNode>
</siteMap>
```

Ensuite, il faut communiquer à ASP.NET notre souhait d'exploiter les facilités d'aide à la navigation dans ce site au moyen d'une une balise `<siteMap>` dans le fichier de configuration :

*Exemple :* *Web.Config*

```
<?xml version="1.0"?>
<configuration
 xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <system.web>
      <pages masterPageFile="~/MyMasterPage.master"/>
      <siteMap defaultProvider="MySiteMapProvider" enabled="true"  >
         <providers>
            <add type="System.Web.XmlSiteMapProvider"
                 name="MySiteMapProvider"
                 siteMapFile= "App.SiteMap"/>
         </providers>
      </siteMap>
...
```

On voit apparaître la notion de *fournisseur d'organisation de site* (*SiteMap provider* en anglais). Vous configurez un tel fournisseur au moyen d'une classe dérivant de `System.Web.SiteMapProvider` et d'une source de donnée contenant l'organisation du site. Ici, nous nous servons de la classe `System.Web.XmlSiteMapProvider` et de notre fichier `App.SiteMap`. Il s'agit bien d'une utilisation du *design pattern provider* présenté en page 902.

Un même site peut contenir plusieurs fournisseurs d'organisation de site. De plus, l'organisation d'un site peut être stockée dans plusieurs fichiers `.SiteMap`. Ces deux possibilités font l'objet de l'article **How to: Configure Multiple Site Maps and Site-Map Provider** des **MSDN**.

Vous pouvez créer votre propre type de fournisseur d'organisation de site au moyen d'une classe propriétaire dérivée de `SiteMapProvider`. Par exemple, vous pouvez souhaiter stocker l'organisation de votre site dans une base de données ou bien intervenir dynamiquement sur l'arborescence présentée au client. La fabrication d'une telle classe est exposée dans l'article **SiteMapProvider Class (System.Web)** des **MSDN**.

Une fois que vous avez configuré un fournisseur d'organisation de site, vous devez ajouter à chaque page susceptible de contenir un contrôle d'aide à la navigation, un contrôle de type `<asp:SiteMapDataSource>` liée à ce fournisseur. Vous pouvez alors vous servir de contrôles de type `<asp:Menu>` `<asp:TreeView>` et `<asp:SiteMapPath>` pour aider à la navigation. En plaçant de tels contrôles dans des master pages, on peut facilement développer un modèle puissant de design de page d'un site. Bien entendu, chacun de ces contrôles présente de nombreuses propriétés permettant de modifier leur rendu graphique.

L'exemple suivant illustre l'utilisation d'un contrôle `<asp:Menu>` au sein d'une master page :

*Exemple 23-85 :* *MyMasterPage.master*

```
<%@ master language="C#" %>
<html><head runat="server"/><body>
<form id="Form1" runat="server">
   <table id="Tbl1" width="100%" height="30px"
                    cellspacing="0" cellpadding="0" >
      <tr> <td width="100%" bgcolor="silver">Smacchia Site</td> </tr>
   </table>
   <table id="Tbl2" width="100%" height="50%"
                    cellspacing="0" cellpadding="0">
```

```
      <tr>
         <td width="30%"  bgcolor="yellow">
           <asp:SiteMapDataSource ID="MySiteMapDataSource"
                                  runat="server" />
            <asp:Menu ID="Menu1" runat="server" Orientation="Horizontal"
                      DataSourceID="MySiteMapDataSource" >
            </asp:Menu>
         </td>
         <td align="center">
            <asp:contentplaceholder id="MyContentPH" runat="Server">
            </asp:contentplaceholder>
         </td>
      </tr>
   </table>
</form></body></html>
```

# *Sécurité*

Il est conseillé d'avoir assimilé les concepts présentés dans le chapitre 6 consacré à la sécurité avant d'aborder cette section.

La gestion de la sécurité est particulièrement importante dans le cas d'applications web. En effet, lorsque l'application est accessible à partir d'internet, il faut absolument qu'elle interdise l'accès aux ressources critiques du serveur aux utilisateurs non autorisés. Il faut donc un mécanisme d'authentification, permettant d'identifier l'entité à l'origine d'une requête. Il faut aussi un mécanisme d'autorisation permettant de spécifier quelles sont les ressources accessibles durant l'exécution d'une requête, en fonction de l'identité à l'origine de la requête.

## *Authentification des utilisateurs Windows avec IIS*

IIS présente un mécanisme d'authentification indépendant des mécanismes d'ASP.NET. Vous pouvez choisir de n'activer l'authentification qu'au niveau d'IIS, qu'au niveau d'ASP.NET ou au niveau d'IIS et d'ASP.NET.

Pour configurer l'authentification IIS pour un répertoire virtuel donné, il faut effectuer la manipulation suivante avec l'outil de configuration d'IIS : *[click droits sur le répertoire virtuel concerné]* ► *Propriétés* ► *Sécurité du répertoire* ► *Connexion anonyme et contrôle d'identification* ► *Modifier.* Vous obtenez alors la fenêtre suivante :

IIS offre quatre modes d'authentification : Connexion Anonyme, Authentification Digest, Authentification de base et Authentification intégrée *Windows*. Ils sont tous basés sur les comptes utilisateurs *Windows*.

- Le mode d'authentification *Connexion anonyme* n'offre aucun contrôle. Il implique que tous les clients peuvent accéder à l'application. Dans le cas où ce mode est activé avec d'autres modes d'authentification IIS, tout se passera comme si seul le mode d'authentification anonyme était activé. Vous avez la possibilité de fournir un compte *Windows* qui représente les utilisateurs anonymes. Ce compte doit obligatoirement avoir un mot de passe non vide.
- Le mode d'*Authentification Digest* envoie une valeur de hachage du mot de passe d'un utilisateur *Windows* plutôt que le mot de passe lui-même. Cette technique a donc l'avantage


*Figure 23-27 : Méthodes d'authentification sous IIS*

d'éviter de faire circuler le mot de passe sur le réseau. Cependant cette technique n'est pas encore supportée par tous les navigateurs.

- Le mode d'*Authentification de base* n'est pas vraiment sécurisé. En effet, le nom de l'utilisateur ainsi que son mot de passe sont envoyés au serveur sous une forme encodée et non pas chiffrée. La technique d'encodage base-64 est utilisée. Vous pouvez cependant avoir recours au mode HTTPS afin de crypter ces données sensibles.
- Le mode d'*Authentification intégrée Windows* utilise un système d'échange de données chiffrées avec le client. Ce mode n'est supporté que par les clients .NET et le navigateur *Internet Explorer*.

Au niveau d'une application ASP.NET, les mécanismes d'authentifications d'IIS ne comptent que si vous activer le mécanisme d'*emprunt d'identité* comme ceci :

*Exemple :* *Web.Config*

```
<?xml version="1.0"?>
<configuration
 xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <system.web>
      <identity impersonate = "true" />
...
```

En page 209, nous expliquons que le mécanisme d'emprunt d'identité permet d'affecter un utilisateur *Windows* authentifié au contexte de sécurité d'un thread non géré. En l'occurrence il s'agit de l'utilisateur *Windows* authentifié par IIS et du thread non géré sous-jacent au thread géré qui

sert la requête dans le processus d'ASP.NET. Rappelons alors que *Windows* vérifie la validité de tout accès aux ressources *Windows* (fichiers, répertoires etc) de la part d'un thread. Pour cela, il se base sur le compte utilisateur associé au thread et au descripteur de sécurité de la ressource (voir page 211). Rappelons que par défaut, le processus d'ASP.NET et chacun de ses threads s'exécutent dans le contexte de l'utilisateur *Windows* spécifié dans la section `<processModel>` du fichier de configuration `Machine.Config` (voir page 890).

Vous pouvez aussi contrôler programmatiquement l'appartenance à un rôle *Windows* de l'utilisateur *Windows* associé au thread courant comme ceci :

```
...
   IPrincipal p = Thread.CurrentPrincipal ;
   if( p.IsInRole(@"BUILTIN\Administrators") ){
      // Effectuer ici les opérations réservées aux administrateurs
   }
...
```

ASP.NET permet de simplifier considérablement cette opération de vérification. En effet, dans le fichier de configuration de l'application, vous pouvez définir une liste de *règles d'accès* qu'ASP.NET doit appliquer à chaque requête. Concrètement une règle est une permission (allow) ou une interdiction (deny) d'effectuer la requête, en fonction du principal ou du rôle joué par le principal. Par exemple pour n'autoriser que les utilisateurs *Mathieu* et *Julien* et les administrateurs locaux à effectuer des requêtes, il faut écrire ceci dans le fichier de configuration :

*Exemple :* *Web.Config*

```
<?xml version="1.0"?>
<configuration
 xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <system.web>
      <authentication mode="Windows" />
      <authorization>
         <allow users = "Mathieu,Julien" />
         <allow roles = "BUILTIN\Administrators" />
         <deny  users = "*" />
      </authorization>
...
```

Le nom d'utilisateur `"*"` représente tous les utilisateurs. Le nom d'utilisateur `"?"` représente l'utilisateur anonyme.

Pour chaque règle, vous pouvez aussi spécifier un verbe qui précise la méthode HTTP autorisée ou interdite. Par exemple, pour autoriser tous les utilisateurs à utiliser la méthode HTTP-GET, et seulement les administrateurs locaux à utiliser la méthode HTTP-POST il faut écrire :

*Exemple :* *Web.Config*

```
...
      <authorization>
         <allow users = "*" verb = "GET" />
         <allow roles = "BUILTIN\Administrators" verb = "POST" />
         <deny  users = "*" />
```

```
      </authorization>
...
```

Comme vous le voyez, l'algorithme d'application de ces règles est très simple : la liste des règles est parcourue linéairement et dès qu'une règle est satisfaite par le principal courant, elle est appliquée. Notez que la liste des règles est en fait la concaténation des listes de règles spécifiées dans chaque fichier de configuration. Donc, cette liste commence par les règles du fichier de configuration du sous répertoire courant de l'application et se finit par les règles du fichier de configuration de la machine.

## *L'authentification au niveau d'ASP.NET*

ASP.NET supporte l'authentification par l'intermédiaire de couches logicielles appelées *fournisseurs d'authentification*. À l'heure actuelle, ASP.NET supporte les fournisseurs d'authentification *Windows*, *Forms* et *Passport*. Vous pouvez développer vos propres fournisseurs d'authentification.

Pour configurer le fournisseur d'authentification ASP.NET qu'une application web doit utiliser, il suffit de paramétrer l'attribut `mode` de l'élément `<authentication>` dans le fichier de configuration de l'application :

```
<configuration>
   <system.web>
      <authentication mode="Windows|Form|Passport|none" />
...
```

En pratique, on utilise le fournisseur d'authentification *Windows* dans un environnement intranet où les comptes *Windows* des utilisateurs sont centralisés sur un serveur. On préfère utiliser les fournisseurs *Passport* et *Forms* dans le cadre d'applications accessibles par internet. Dans le premier cas l'application exploite le mécanisme d'authentification *Microsoft Passport*. Dans le second cas, l'application doit gérer elle-même ses propres comptes utilisateurs, en général à l'aide d'une base de données. Nous allons nous concentrer ici sur le fournisseur d'authentification *Forms*. Nous vous invitons à consulter l'article **The Passport Authentication Provider** des **MSDN** pour plus d'information concernant le fournisseur d'authentification *Passport*.

## *Le fournisseur d'authentification Forms d'ASP.NET*

Le fournisseur d'authentification *Forms* permet aux applications de présenter leur propre interface utilisateur pour saisir les données nécessaires à l'authentification, sous la forme d'une page `.aspx` classique. La méthode interne d'authentification à partir de ces données incombe aussi à l'application web. Lorsqu'un client non authentifié tente d'accéder à une page, ASP.NET le redirige automatiquement vers le formulaire d'authentification. Une fois authentifié, ASP.NET redirige automatiquement l'utilisateur vers la page demandée. Pour éviter au client de devoir s'authentifier explicitement à chaque requête de page, un cookie est créé par ASP.NET lors de l'authentification. Ce cookie contient une clé qui n'est éventuellement valide que pendant une certaine durée. Ce cookie est communiqué au navigateur client qui le stocke et qui l'utilise alors pour les prochaines requêtes. La page suivante illustre une implémentation minimale de ce scénario :

*Exemple 23-86 :* *Login.aspx*

```
<%@ Page Language="C#" %>
<script runat="server">
      void Btn_Click(Object sender, EventArgs e) {
         // En général, on accède ici à une base de données
         // pour authentifier l'utilisateur.
         if (Usr.Text == "pat" && Pwd.Text == "pwd")
            FormsAuthentication.RedirectFromLoginPage(Usr.Text, true);
         else
            Msg.Text = "Mauvais login, veuillez réessayer." ;
      }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
    <form id="form1" runat="server">
       <asp:Label ID="Lbl1" runat="server" Text="Utilisateur:"/>
       <asp:TextBox ID="Usr" runat="server"/>
       <asp:Label ID="Lbl2" runat="server" Text="Mot de passe:"/>
       <asp:TextBox ID="Pwd" runat="server" TextMode="Password"/>
       <asp:Button ID="Button1" runat="server" Text="Login"
                   OnClick="Btn_Click"  /><br/>
       <asp:Label ID="Msg" runat="server"/>
    </form>
</body>
</html>
```

Il faut bien entendu spécifier à ASP.NET que les requêtes anonymes (i.e sans cookie d'authentification) doivent être redirigées vers cette page :

*Exemple :* *Web.Config*

```
<?xml version="1.0"?>
<?xml version="1.0"?>
<configuration
 xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <system.web>
      <authorization>
         <deny users="?"/>
      </authorization>
      <authentication mode="Forms">
         <forms loginUrl="Login.aspx" />
      </authentication>
...
```

Le simple appel à la méthode `RedirectFromLoginPage()` suffit à ASP.NET pour créer un cookie, le communiquer au client et le rediriger vers la page d'origine. On s'aperçoit qu'ASP.NET stocke la page vers laquelle le client doit être redirigé dans l'URL utilisée lors de la première redirection vers la page d'authentification :

```
http://localhost:1968/MyWebSite/...
... Login.aspx?ReturnUrl=%2fMyWebSite%2fdefault.aspx
```

La section <form> présente plusieurs attributs pour nommer le cookie, protéger les informations qui y sont contenues et lui assigner une durée de validité.

À l'instar de ce que nous avons vu en page 901 concernant la gestion de session sans cookies, la gestion de l'authentification peut se faire aussi sans cookies. Tout comme l'élément <sessionState> l'élément <forms> présente un attribut nommé cookieless. Vous pouvez vous servir de cet attribut pour contraindre ou laisser le choix à ASP.NET d'utiliser l'URI pour stocker la clé représentant la preuve que l'utilisateur est authentifié. Voici à quoi ressemble une telle URI :

```
http://localhost:1968/MyWebSite/(F(Xd6tEmGNyNKdZ-Df9B69q4F2JUXxLd-
7fk8QzQt1qhqWk1FLyUX2_gw31a5XePl8YdR-YzYxEm6kuareBQqgjw2))/default.aspx
```

Soyez conscient que dans cet exemple, le mot de passe n'est pas crypté dans la requête d'authentification. Pour éviter ce problème majeur de confidentialité, il faut se servir de HTTPS pour encrypter cette requête. Comme nous venons de le voir, les autres requêtes n'ont pas besoin d'être cryptées puisqu'elles ne contiennent pas le mot de passe.

Nous allons maintenant nous intéresser aux possibilités ajoutées par ASP.NET 2.0 au niveau de la sécurité. Vous pouvez maintenant gérer les utilisateurs et leur appartenance à des rôles d'une manière standard au moyen d'une base de données. En outre, plusieurs nouveaux contrôles serveurs viennent grandement simplifier le développement d'application ASP.NET supportant l'authentification.

## *Gestion des utilisateurs*

ASP.NET 2.0 présente un *framework* pour standardiser la gestion des comptes utilisateurs par exemple dans une base de données. Ce *framework* a été conçu de façon à être indépendant de l'authentification par formulaire. Cependant, l'intégration de la gestion des utilisateurs et de l'authentification par formulaire présente un modèle particulièrement souple et puissant qui vous permet de réécrire notre Exemple 23-86 en quatre lignes de code C# :

*Exemple 23-87 :* *Login.aspx*

```
<%@ Page Language="C#" %>
<script runat="server">
      void Btn_Click(Object sender, EventArgs e) {
         if ( Membership.ValidateUser( Usr.Text , Pwd.Text ) )
            FormsAuthentication.RedirectFromLoginPage(Usr.Text, true) ;
         else
            Msg.Text = "Mauvais login, veuillez réessayer." ;
      }
</script>
...
```

Bien entendu, pour que cet exemple fonctionne, il faut préciser à ASP.NET où sont stockés les données relatives aux utilisateurs et comment y accéder. Pour cela, il faut avoir recours à un *fournisseur d'utilisateurs* que l'on défini dans le fichier de configuration :

*Exemple :* *Web.Config*

```
<?xml version="1.0"?>
<configuration
     xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
```

```
   <connectionStrings>
      <add name="MyLocalSqlServer"
           connectionString="Data Source=localhost;Integrated
                     Security=SSPI ;Initial Catalog=MyWebSiteUsers;" />
   </connectionStrings>
   <system.web>
      <membership defaultProvider="MyMembershipSqlProvider" >
         <providers>
            <add    name="MyMembershipSqlProvider"
                    type="System.Web.Security.SqlMembershipProvider,
                          System.Web, Version=2.0.0.0, Culture=neutral,
                          PublicKeyToken=b03f5f7f11d50a3a"
                    connectionStringName="MyLocalSqlServer" />
         </providers>
      </membership>
...
```

Les données relatives aux utilisateurs de ce site sont stockées dans une base de données nommée `MyWebSiteUsers`. Cette base se trouve dans un SGBD local de type *SQL Server*. Pour construire une telle base de données, il suffit d'utiliser l'outil `aspnet_regsql.exe` décrit en page :

```
>aspnet_regsql.exe -S <host> -U <usr> -P <pwd> -d <database> -ed
```

Puisque nos données utilisateurs sont stockées dans une base de données relationnelle, nous avons précisé à ASP.NET d'utiliser le fournisseur d'utilisateurs `System.Web.Security.SqlMembershipProvider`. Un fournisseur d'utilisateurs est une classe qui dérive de la classe `System.Web.Security.MembershipProvider`. Vous pouvez créer votre propre fournisseur en développant une telle classe. Notez qu'ASP.NET 2.0 fournit aussi par défaut le fournisseur `System.Web.Security.ActiveDirectoryMembershipProvider`. Il s'agit bien d'une utilisation du *design pattern provider* présenté en page 902.

Dans notre Exemple 23-87, la méthode statique `bool ValidateUser(string username, string password)` de la classe `System.Web.Security.Membership` est capable d'aller vérifier si un utilisateur existe en exploitant le fournisseur d'utilisateurs de l'application. Cette classe présente d'autres méthodes statiques dont la signification et l'utilisation sont particulièrement intuitives :

```
MembershipUser CreateUser(string username, string password) ;
MembershipUser CreateUser(string username, string password, string email) ;
bool DeleteUser(string username) ;
bool DeleteUser(string username, bool deleteAllRelatedData) ;
MembershipUserCollection FindUsersByEmail(string emailToMatch) ;
MembershipUserCollection FindUsersByName(string usernameToMatch) ;
string GeneratePassword(int length, int numOfNonAlphanumericCharacters) ;
MembershipUserCollection GetAllUsers() ;
int GetNumberOfUsersOnline() ;
MembershipUser GetUser() ; // retourne l'utilisateur courant
MembershipUser GetUser(string username) ;
string GetUserNameByEmail(string emailToMatch) ;
void UpdateUser(MembershipUser user) ;
bool ValidateUser(string username, string password) ;
```

Les instances de la classe `MembershipUser` représentent à l'exécution les utilisateurs. Cette classe contient quelques propriétés permettant d'obtenir le nom, le email, la date de création, la date de dernier login, la date de dernière demande de page, la question à demander à l'utilisateur en cas de perte de mot de passe et quelques méthodes permettant de changer le mot de passe ou la question associée. Comme vous pouvez le constater, seules les informations relatives à l'authentification sont stockées dans une instance de `MembershipUser`. Nous verrons dans la prochaine section consacrée à la personnalisation comment étendre ce mécanisme pour associer à un utilisateur des informations personnelles comme son adresse, son numéro de téléphone voire l'historique de ses achats.

Sachez qu'un mécanisme de verrouillage est prévu en cas de trop nombreuses tentatives de login ratées. Dans ce cas, la méthode `Membership.ValidateUser()` ne peut retourner `true` tant que le verrouillage n'a pas été désactivé pour l'utilisateur concerné par l'appel à la méthode `MembershipUser.UnlockOut()`. Vous pouvez configurer ce mécanisme avec les attributs entiers `passwordAttemptThreshold` et `passwordAttemptWindows` qui se placent dans l'élément de configuration `<membership>`. Le verrouillage est automatiquement mis en place si plus de `passwordAttemptThreshold` logins infructueux ont lieu en moins de `passwordAttemptWindows` minutes.

## *Gestion des rôles*

En général, il est beaucoup plus pratique d'utiliser la notion de rôle pour valider dans votre code les permissions des utilisateurs. Aussi, ASP.NET 2.0 présente un *framework* permettant de gérer des rôles ainsi que l'appartenance des comptes utilisateurs à ces rôles. Pour exploiter ce *framework*, il faut avoir recours à un *fournisseur de rôles* que l'on définit dans le fichier de configuration :

*Exemple :* *Web.Config*

```
<?xml version="1.0"?>
<configuration
     xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <connectionStrings>
      <add name="MyLocalSqlServer"
           connectionString="Data Source=localhost;Integrated
                     Security=SSPI ;Initial Catalog=MyWebSiteUsers;" />
   </connectionStrings>
   <system.web>
      <roleManager defaultProvider="MyRoleSqlProvider" enabled="true" >
         <providers>
            <add    name="MyRoleSqlProvider"
                    type="System.Web.Security.SqlRoleProvider,
                          System.Web, Version=2.0.0.0, Culture=neutral,
                          PublicKeyToken=b03f5f7f11d50a3a"
                    connectionStringName="MyLocalSqlServer" />
         </providers>
      </roleManager>
...
```

Comme vous pouvez le constater, la notion de fournisseur de rôles se définit de la même manière que la notion de fournisseur d'utilisateurs puisque ici aussi le *design pattern provider*

présenté en page 902 est utilisé. Un fournisseur de rôles est une classe qui dérive de la classe `System.Web.Security.RoleProvider`. ASP.NET 2.0 présente les classes `SqlRoleProvider`, `AuthorizationStoreRoleProvider` et `WindowsTokenRoleProvider`. Vous pouvez créer votre propre fournisseur de rôles au moyen d'une classe dérivant de `RoleProvider`.

De la même manière que l'on utilise les méthodes statiques de la classe `MembershipUser` pour manipuler programmatiquement les comptes utilisateurs, on utilise les méthodes statiques de la classe `System.Web.Security.Roles` pour manipuler programmatiquement les rôles. Ici aussi leurs signatures sont particulièrement éloquentes :

```
void AddUsersToRole(string[] usernames,string roleName) ;
void AddUsersToRoles(string[] usernames,string[] roleNames) ;
void AddUserToRole(string username,string roleName) ;
void AddUserToRoles(string username,string[] roleNames) ;
void CreateRole(string roleName) ;
bool DeleteRole(string roleName) ;
bool DeleteRole(string roleName,bool throwOnPopulatedRole) ;
string[] FindUsersInRole(string roleName,string usernameToMatch) ;
string[] GetAllRoles() ;
string[] GetRolesForUser() ;
string[] GetRolesForUser(string username) ;
string[] GetUsersInRole(string roleName) ;
bool IsUserInRole(string roleName) ;
bool IsUserInRole(string username,string roleName) ;
void RemoveUserFromRole(string username,string roleName) ;
void RemoveUserFromRoles(string username,string[] roleNames) ;
void RemoveUsersFromRole(string[] usernames,string roleName) ;
void RemoveUsersFromRoles(string[] usernames,string[] roleNames) ;
bool RoleExists(string roleName) ;
```

Enfin, remarquez que l'on peut définir des règles d'accès en se basant sur ce type de rôles dans l'élément de configuration `<authorization>` :

*Exemple :* *Web.Config*

```
<?xml version="1.0"?>
<configuration
 xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <system.web>
      <authorization>
         <allow roles="admin"/>
         <deny  roles="simpleUser"/>
      </authorization>
...
```

## *Contrôles serveurs spécialisés dans l'authentification*

ASP.NET 2.0 présente plusieurs contrôles serveurs évolués qui implémentent les besoins récurrents concernant la gestion de comptes utilisateurs d'un site :

- Le contrôle `CreateUserWizard` prend en charge la création d'utilisateurs avec la vérification du mot de passe et la possibilité de fournir un email et un couple question/réponse utilisés en cas de perte de mot de passe. Ce contrôle est hautement paramétrable avec une expression régulière pour le mot de passe, une autre pour le email, des messages d'erreurs personnalisables pour tous types d'erreurs (utilisateur déjà existant, champs non remplis etc.).
- Le contrôle `Login` permet à un utilisateur de s'authentifier en entrant un nom de compte et un mot de passe. C'est en général la page qui contient ce contrôle qui doit être spécifiée dans l'attribut `loginUrl`.
- Le contrôle `LoginStatus` permet d'insérer dans la page HTML en cours de construction un hyperlien. Cet hyperlien varie selon que l'utilisateur courant est authentifié ou non. En général on l'utilise pour afficher *Login* dans le cas où l'utilisateur n'est pas authentifié et *Logout* dans le cas contraire.
- Le contrôle `LoginView` est similaire au contrôle `LoginStatus` à ceci près qu'il permet d'afficher un ensemble de contrôles différents selon que l'utilisateur courant est authentifié ou non.
- Le contrôle `LoginName` permet d'insérer dans la page HTML en cours de construction une chaîne de caractères contenant le nom de l'utilisateur couramment authentifié.
- Le contrôle `PasswordRecovery` permet d'envoyer un email contenant le mot de passe perdu à l'adresse associée avec le compte utilisateur. Il se base sur le système question/réponse. Si le mot de passe était encrypté d'une manière irréversible, ASP.NET fabrique un nouveau mot de passe.
- Le contrôle `ChangePassword` permet à un utilisateur de changer son mot de passe.

Les contrôles tels que `CreateUserWizard`, `ChangePassword` ou `Login` dont les implémentations exploitent directement le fournisseur d'utilisateurs présentent un attribut `MembershipProvider` pour le définir.

Signalons que l'interface graphique web de ASP.NET 2.0 présente un onglet sécurité qui vous permet d'administrer l'ensemble des utilisateurs, des rôles et des règles d'accès d'une application web.

# *Personnalisation*

## *Fournisseur de personnalisation et gestion des données personnelles*

ASP.NET 2.0 présente un *framework* permettant de stocker et de manipuler d'une manière standard les données personnelles de chaque utilisateur. Ici encore, pour préciser le mode de stockage de ces données ASP.NET exploite le *design pattern provider*. ASP.NET 2.0 ne présente qu'une classe de *fournisseur de personnalisation*, la classe `SqlProfileProvider`. Cette classe exploite une base de données qui a été préparée au préalable par l'outil `aspnet_regsql.exe` . Bien entendu, vous pouvez construire votre propre fournisseur de personnalisation au moyen d'une classe dérivée de `System.Web.Profile.ProfileProvider`.

*Exemple :* *Web.Config*

```
<?xml version="1.0"?>
<configuration
     xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <connectionStrings>
      <add name="MyLocalSqlServer"
           connectionString="Data Source=localhost;Integrated
                      Security=SSPI ;Initial Catalog=MyWebSiteUsers;" />
   </connectionStrings>
   <system.web>
      <profile enabled="true" defaultProvider="MyProfileSqlProvider">
         <providers>
            <add    name="MyProfileSqlProvider"
                    type="System.Web.Profile.SqlProfileProvider,
                          System.Web, Version=2.0.0.0, Culture=neutral,
                          PublicKeyToken=b03f5f7f11d50a3a"
                    connectionStringName="MyLocalSqlServer" />
         </providers>
         <properties>
            <add name="CompanyName" type="System.String" />
            <add name="Birthday" type="System.DateTime" />
            <add name="RequestCount" type="System.Int32"
                 defaultValue="0" />
         </properties>
      </profile>
...
```

La section `<profile>` est consacrée à la personnalisation. La sous-section `<properties>` permet de définir les paramètres personnels. Chacun de ces paramètres est nommé, typé et peut avoir une valeur par défaut.

Ces paramètres peuvent être exploités programmatiquement au moyen de la classe `Profile`. Le compilateur fait en sorte que la classe `ProfileCommon` ait une propriété publique d'instance accessible en écriture et en lecture pour chacun des paramètres précisés. Pour accéder à cette propriété, il suffit d'écrire `Profile.NomDeLaPropriété` :

```
...
   void Page_Load(Object sender, EventArgs e) {
      Profile.RequestCount += 1 ;
      LabelRequestCount.Text = Profile.RequestCount.ToString() ;
   }
...
```

Le compilateur ASP.NET interprète ceci comme un accès à une instance cachée de la classe `ProfileCommon`. Cette instance cachée contient les données de l'utilisateur couramment logué. S'il n'y a pas d'utilisateur couramment authentifié, une exception est levée.

Les données personnelles d'un utilisateur ne sont chargées que lors du premier accès à une donnée durant le traitement d'une requête. Le cas échéant, toutes les données sont chargées. Elles seront sauvegardées à la fin du traitement. Cela signifie que vous n'avez à écrire absolument aucune ligne de code pour prévoir le chargement ou la sauvegarde des données.

Cependant, il peut être efficace de ne pas charger toutes les données personnelles à chaque requête par exemple parce qu'elles représentent un gros volume et que l'on ne fait qu'incrémenter un compteur `RequestCount` à chaque requête. Aussi, ASP.NET 2.0 vous permet de partitionner vos données dans des groupes. Lorsque l'on accède à une donnée d'un groupe, seules les données de celui-ci sont chargées en mémoire :

*Exemple :* *Web.Config*

```
...
      <properties>
         <add name="RequestCount" type="System.Int32" defaultValue="0"/>
         <group name="ProfessionalInfo">
            <add name="CompanyName" type="System.String"/>
            <add name="ProfessionalEmail" type="System.String"/>
         </group>
      </properties>
...
```

Un groupe ne peut contenir d'autres groupes. L'accès à une donnée d'un groupe se fait par l'intermédiaire d'une propriété portant le nom du groupe :

```
...
   void Page_Load(Object sender, EventArgs e) {
      LblCompanyName.Text = Profile.ProfessionalInfo.CompanyName ;
   }
...
```

Vous pouvez typer une donnée utilisateur avec n'importe quel type sérialisable en XML ou en binaire, y compris des types collections tels que `System.Collections.Specialized.StringCollection`. Il est intéressant d'aller examiner le contenu des tables créées par `aspnet_regsql.exe` pour se rendre compte que les données personnelles sont sauvées sous une forme sérialisée. Cependant, vous ne pouvez pas utiliser de types génériques.

## *Personnalisation et utilisateurs anonymes*

Il peut être souhaitable de pouvoir sauvegarder des données relatives à un utilisateur anonyme. Par exemple, la plupart des sites en ligne de commerce vous autorisent à commencer vos achats avant même de vous authentifier. Le processus d'authentification n'a alors lieu en général qu'en fin d'achat, lorsque vous devez impérativement fournir vos paramètres personnels tels que votre numéro de carte bancaire et votre adresse de livraison pour procéder à l'achat.

ASP.NET 2.0 offre des facilités pour implémenter ce scénario. Il faut préciser que vous souhaiter pouvoir identifier un utilisateur anonyme au moyen de la section `<anonymousIdentification>`. L'idée est qu'un nouvel utilisateur est automatiquement créé pour chaque utilisateur anonyme qui effectue une requête. ASP.NET lui donne pour nom un nouvel identificateur unique. Cet identificateur unique est connu du navigateur client et est réutilisé à chaque requête. Cela se fait soit au moyen d'un cookie soit en utilisant un paramètre d'URL. L'élément `<anonymousIdentification>` accepte un attribut `cookieless` dont l'ensemble des valeurs possibles est expliqué en page 901.

Chacun des paramètres personnels susceptibles d'être sauvegardé pour un utilisateur anonyme doit être marqué avec l'attribut `allowAnonymous` positionné à `true`. Voici à quoi peut alors ressembler notre fichier de configuration :

*Exemple :* *Web.Config*

```
<?xml version="1.0"?>
<configuration
     xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <connectionStrings>...</connectionStrings>
   <anonymousIdentification enabled="true" cookieless="AutoDetect"/>
   <system.web>
      <profile enabled="true" defaultProvider="MyProfileSqlProvider">
         <providers>...</providers>
         <properties>
            <add name="RequestCount" type="System.Int32"
                 defaultValue="0" allowAnonymous="true" />
            ...
         </properties>
      </profile>
...
```

Pour implémenter correctement le scénario de l'utilisateur anonyme qui va potentiellement finir par s'authentifier, il faut prévoir la migration des données du compte utilisateur anonyme vers le compte utilisateur réel. Cette migration doit être réalisée au moment de l'authentification. Pour cela, il suffit d'insérer votre code de migration dans la méthode `Profile_MigrationAnonymous()` dans le fichier `Global.asax` :

*Exemple 23-88 :* *Global.asax*

```
<%@ Application Language="C#" %>
<script runat="server">
protected void Profile_MigrateAnonymous(object sender,
                                        ProfileMigrateEventArgs e) {
   ProfileCommon anonymousProfile = Profile.GetProfile(e.AnonymousId);
   if (anonymousProfile != null) {
      Profile.LoginCount += anonymousProfile.LoginCount;
   }
}
</script>
```

## *Personnalisation vs. Session*

Vous l'aurez remarqué, les concepts de sessions et de personnalisation sont proches puisque dans chaque cas il s'agit de stocker des données relatives à un utilisateur de manière à ce qu'elles « survivent » entre chaque requête. Aussi, il est important de souligner la différence sémantique entre session et personnalisation :

- Le mécanisme de personnalisation est fait pour stocker **des données intrinsèques à un utilisateur** telles que leur date d'anniversaire, leur adresse ou éventuellement leur numéro de carte bancaire.
- Le mécanisme de session est fait pour stocker **des données temporaires relatives à un utilisateur** telles que la liste de ses achats courants.

En outre, le mécanisme de personnalisation est mieux pensé. Il est donc plus simple à exploiter et plus efficace. Notamment :

- Les noms et types des paramètres sont vérifiés à la compilation et non à l'exécution.
- Les données personnelles à un utilisateur sont chargées à la demande contrairement aux données de sessions qui sont chargées puis sauvées à chaque requête.

# Styles, Thèmes et Skins

## Style CSS, contrôles HTML et contrôles serveur

La plupart des navigateurs prennent en compte la propriété `style` d'un contrôle HTML pour modifier sont apparence. Par exemple, la page HTML suivante contient une zone d'édition de texte stylée :

```
<html><body>
  <input type="text" style="font: 18pt times ; background-color:yellow;
                            border-style:dashed ; width:500 ;
                            border-color:blue;"
         value="Entrez votre texte !" />
</body></html>
```

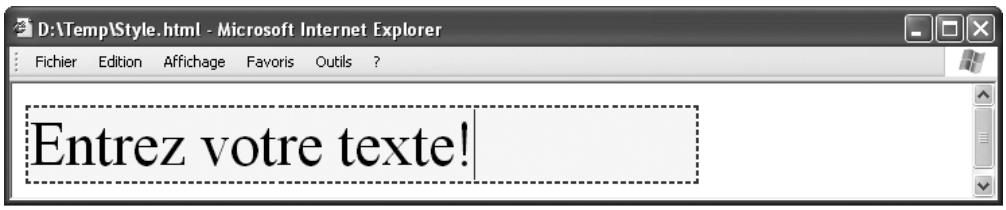


*Figure 23-28 : Style appliqué à une zone d'édition de texte*

La propriété `style` contient une liste d'attributs *CSS* (*Cascading Style Sheet*) séparés par des points virgules. Une autre syntaxe peut être utilisée pour factoriser un même style sur plusieurs contrôles :

```
<html><head><style>
   .inputstyle  { font: 34pt times ; background-color:yellow;
                  border-style:dashed ; width:500 ; border-color:blue ; }
</style></head>
<body>
  <input type="text" class="inputstyle" value="Entrez votre texte !" />
</body></html>
```

La classe `HtmlControl`, et par conséquent tous les contrôles serveurs HTML, présentent la propriété `CssStyleCollection Style{get}` qui permet de stocker un dictionnaire où les clés sont les noms des attributs CSS.

La classe de base commune à tous les contrôles serveurs web, à savoir la classe `WebControl`, présente la propriété `string CssClass{get;set;}` qui permet de spécifier le nom d'un style

à appliquer (tel que `inputstyle`). Elle présente aussi quelques attributs fortement typés commun à tous les contrôles serveurs web tels que `BackColor` ou `BorderWidth`. L'espace de noms `System.Web.UI.Controls` ainsi que ses sous espaces de noms contiennent plusieurs classes de définition de style qui dérivent de la classe `System.Web.UI.Controls.Style`. Ces classes telles que `TableStyle`, `PanelStyle`, `TitleStyle` ou `TreeNodeStyle` sont spécifiques au style de chaque contrôle web ou de chaque partie d'un contrôle web. Les propriétés de ces styles peuvent être initialisées dans la définition du contrôle comme ceci :

```
<asp:Calendar ...>
    <TitleStyle BorderColor="green" BorderWidth="3" ... />
 </asp:Calendar>
```

...ou comme cela :

```
<asp:Calendar ...
   TitleStyle-BorderColor="green"
   TitleStyle-BorderWidth="3" ... />
```

## *Notion de thème*

Malgré la possibilité de factoriser les styles dans un élément `<header>`, la nécessité d'un mécanisme puissant permettant de configurer simplement l'apparence de tout un site à pousser les concepteurs d'ASP.NET 2.0 à créer la notion de *thème*. Un thème est la définition d'un ensemble de styles qui peuvent être appliqués statiquement ou dynamiquement aux contrôles des pages d'un site. La modification des styles d'un thème entraîne automatiquement la modification de l'apparence des pages qui l'exploitent.

Un thème est matérialisé par le contenu d'un répertoire. Le nom du thème est le nom du répertoire. Un thème peut être local à une application ou partagé par toutes les applications d'une machine. Dans le premier cas, le répertoire du thème doit être un sous répertoire du répertoire `/App_Theme` de l'application. Dans le second cas le répertoire du thème est soit un sous répertoire du répertoire d'installation d'ASP.NET (à savoir `%WINDIR%\Microsoft.NET\Framework\<version>\ASP.NETClientFiles\Themes`) soit un sous répertoire du répertoire `Inetpub\wwwroot\aspnet_client\system_web\<version>\Themes` dans le cas d'application web hébergées par IIS.

Il y a plusieurs façons d'appliquer un thème aux pages d'une application :

- En utilisant l'attribut `theme` de la section `<pages>` dans le fichier de configuration. Dans ce cas le thème est appliqué à toutes les pages impactées par ce fichier de configuration :

  *Exemple :* *`Web.Config`*

  ```
  <?xml version="1.0"?>
  <configuration
       xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
     <system.web>
        <pages theme="NomDuTheme" >
  ...
  ```

- En utilisant la sous directive `Theme` de la directive `<@ Page>` des pages `.aspx` concernées. Cette possibilité permet de réécrire pour une page un thème global spécifié dans la section de configuration `<page>` :

*Exemple :* *PagesXXX.aspx*

```
<%@ Page Theme="NomDuTheme"  %>
...
```

- En utilisant la sous directive `StyleSheetTheme` de la directive `<@ Page>` des pages `.aspx` concernées. Il faut savoir que pour les contrôles d'une page, les styles sont choisis dans cet ordre croissant de priorité : styles définis par le thème spécifié avec la sous directive `StyleSheetTheme` ; styles définis directement dans les déclarations des contrôles ; styles définis dans l'élément `<page>` du fichier de configuration ; styles définis par le thème spécifié avec la sous directive `Theme`.

*Exemple :* *PagesXXX.aspx*

```
<%@ Page StyleSheetTheme="NomDuTheme"  %>
...
```

- Dynamiquement en spécifiant le thème dans la méthode abonnée à l'évènement `PreInit` de la page. Typiquement, cette possibilité est utilisée lorsque l'on souhaite permettre aux utilisateurs authentifiés de personnaliser le thème définissant l'apparence du site :

*Exemple :* *PagesXXX.aspx*

```
<%@ Page Language="C#" %>
<script runat="server">
  protected void Page_PreInit(){
    if (Profile.IsAnonymous == false)
       Page.Theme = Profile.Theme;
  }
</script>
...
```

Enfin, vous pouvez désactiver l'application des styles d'un thème pour un contrôle simplement en positionnant sa propriété `EnableTheming` à `false`. Cette propriété est initialement présentée par la classe `Control`.

## *Notion de Skin*

Un répertoire d'un thème contient un ou plusieurs fichiers d'extension `.skin`, zéro, un ou plusieurs fichiers d'extension `.css` et éventuellement des sous répertoires contenant des ressources référencées par les styles, telles que des images. Les *stylesheets* d'un thème, définis dans ses fichiers d'extension `.css` seront appliqués aux pages concernées.

Les fichiers d'extension `.skin` contiennent les définitions des *skins des types de contrôles* (que l'on pourrait traduire par peau/apparence d'un type de contrôle). La définition d'un skin ressemble à la définition d'un contrôle mis à par que seules certaines propriétés peuvent être positionnées. Ce sont les propriétés qui se rapportent au style du type de contrôle. Vous pouvez les reconnaître en regardant la définition d'un contrôle car elles sont marquées avec l'attribut `System.Web.UI.ThemeableAttribute(true)`. Voici par exemple la définition d'un fichier `.skin` :

*Exemple 23-89 :* *BigText.skin*

```
<asp:Label Font-Bold="true" Font-Size="20" runat="server" />
<asp:TextBox Font-Bold="true" Font-Size="20" runat="server" />
```

Voici une page sur laquelle nous appliquons le thème `ThemeBig` qui ne contient que le fichier `BigText.skin` :

*Exemple 23-90 :* *`Default.aspx`*

```
<%@ Page Language="C#" Theme="ThemeBig" %>
<html xmlns="http://www.w3.org/1999/xhtml">
<body>
   <form id="form1" runat="server">
      <asp:Label ID="Lbl1" runat="server" Text="Label 1" /><br/>
      <asp:Label ID="Lbl2" runat="server" Text="Label 2"
                 Font-Size="10"/><br/>
      <asp:Label ID="Lbl3" runat="server" Text="Label 3"
                 EnableTheming=false  /><br/>
      <asp:TextBox ID="TextBox1" runat="server" Text="TextBox"/>
   </form>
</body>
</html>
```

Enfin, voici une copie d'écran de cette page. On voit bien que seul le `Label` numéro 3 n'a pas été impacté par le thème du fait que l'on a positionné la propriété `EnableTheming` à `false` :

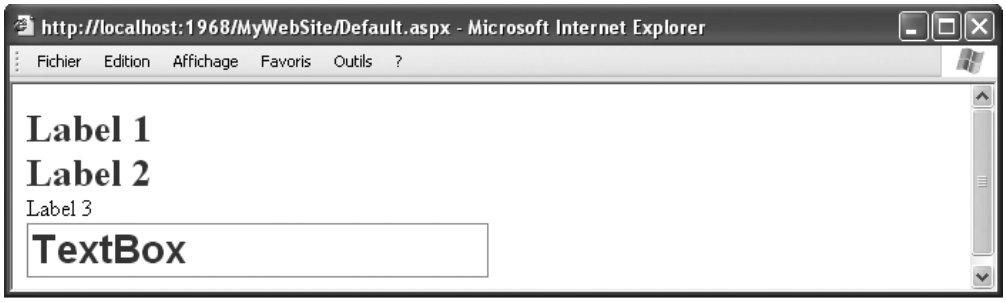


*Figure 23-29 : Page à laquelle on applique un thème*

## *Skin nommé*

Lors de la définition d'un skin, vous avez la possibilité de le nommer avec l'attribut `SkinId`. On peut ainsi définir plusieurs skins relatifs à un même type de contrôle dans un thème. Bien entendu, un thème ne peut contenir des skins homonymes ni plus d'un skin anonyme.

Comme l'on peut sans douter, on peut affecter un skin nommé dans la définition d'un contrôle avec l'attribut `SkinId`. Si à l'exécution un skin nommé est utilisé par un contrôle mais n'existe pas dans le thème courant, ASP.NET n'applique aucun skin au contrôle concerné. Aussi, il est préférable de redéfinir la totalité de vos skins nommés dans la totalité de vos thèmes.

# *WebParts*

ASP.NET 2.0 présente un *framework* dédié à la création de *webParts*. Cette notion de webParts permet à un utilisateur de personnaliser l'ensemble des services que peut lui rendre une page.

Par exemple, on pourrait imaginer que l'utilisateur ait le choix entre les services : dernière nouvelles concernant le milieu des télécommunications ; les valeurs du jour de la bourse ; la météo locale ; le programme TV du soir etc. Chacun de ces services est matérialisé sur la page par un contrôle particulier que l'on nomme une webPart. Une application web qui supporte un tel niveau de personnalisation est nommé *portail web*. Jusqu'ici, dans le monde *Microsoft* seule la technologie *SharePoint* était dédiée à la création de portails. Les webParts d'ASP.NET 2.0 permettent essentiellement :

- De présenter une ou plusieurs zones sur une même page qui contiennent chacune une ou plusieurs webParts. Chacune de ces zones est en fait un contrôle serveurs de type `WebPartZone`. Les webParts contenues dans une zone peuvent être alignées verticalement ou horizontalement.

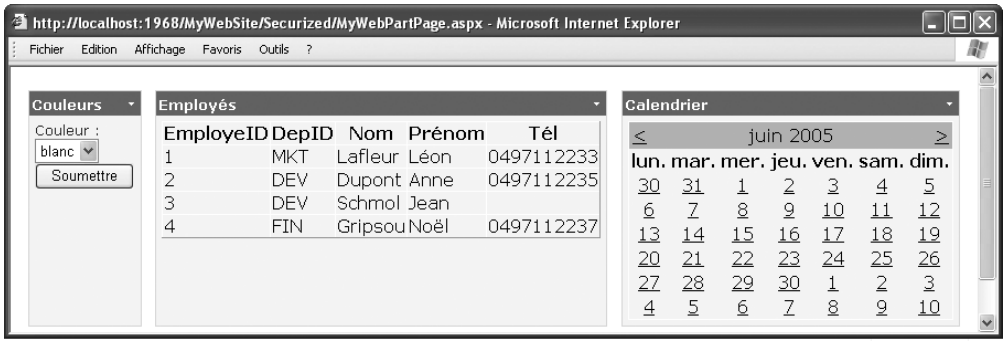


*Figure 23-30 : Une page avec trois webParts*

- De définir l'ordre dans lequel sont affichées les webParts d'une zone.

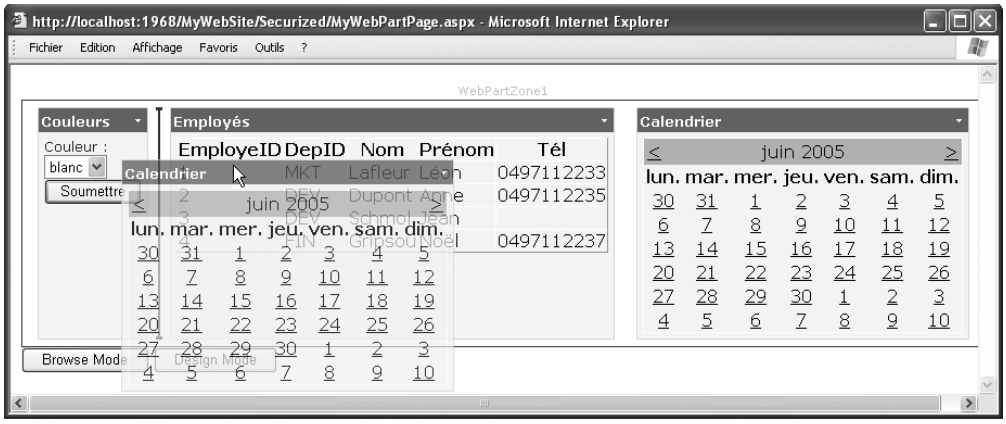


*Figure 23-31 : Changement de l'ordre des webParts dans une zone.*

- De choisir les webParts qui sont contenues dans une zone au moyen d'un catalogue de webParts. Un catalogue par défaut peut être fourni par le site mais l'utilisateur peut importer ses propres webParts stockées dans des fichiers XML sur son disque dur.


*Figure 23-32 : Sélection de webParts dans le catalogue*

- Le contenu d'une webPart est affiché dans une fenêtre spéciale que l'on nomme *chrome*. Pour chaque webPart, l'utilisateur peut définir l'apparence du chrome, les actions présentées dans le chrome (minimise, close etc.), le titre écrit dans le chrome etc. Notez que les actions présentées par un chrome sont nommées *verbes*.

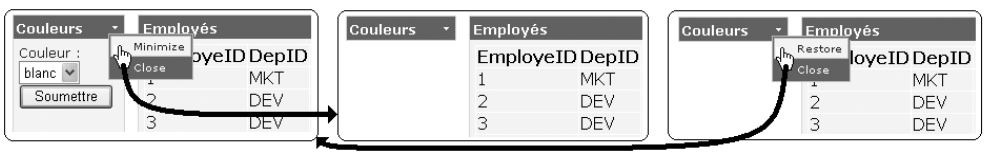


*Figure 23-33 : Menu du chrome*

- De connecter deux webParts. Dans ce cas une webPart joue le rôle du producteur d'information un peu comme une source de données tandis que l'autre joue le rôle de consommateur.

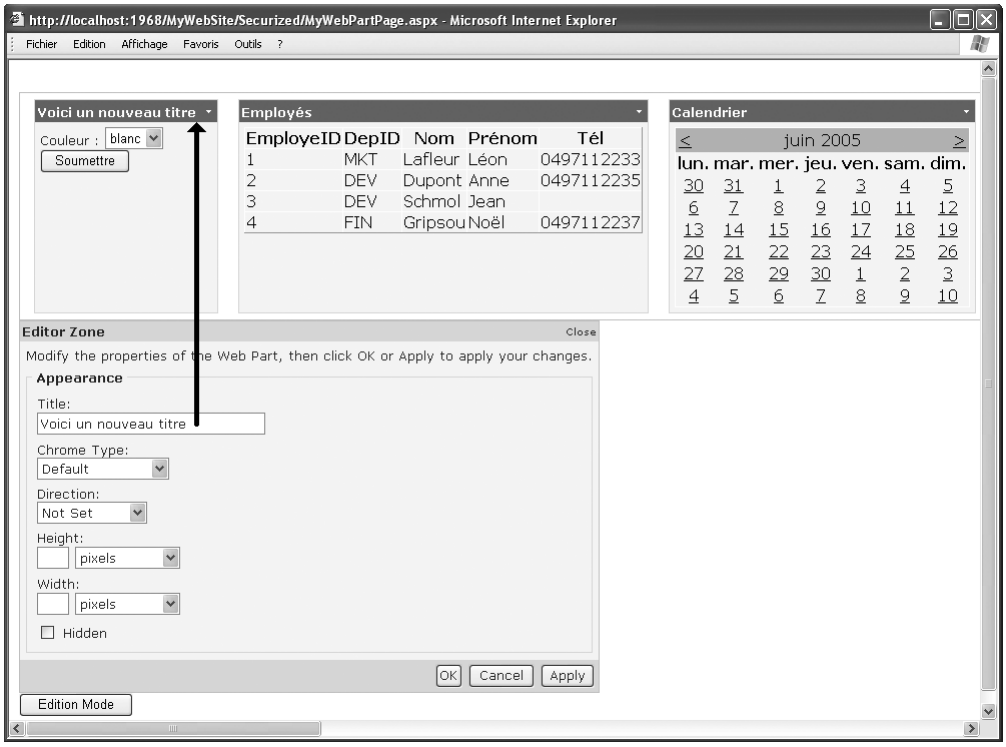


*Figure 23-34 : Édition du chrome*


*Figure 23-35 : Connexion entre deux webParts*

Un utilisateur ne peut modifier les webParts d'une page que s'il est couramment authentifié. En effet, chaque modification entraîne une requête POST, qui à son tour entraîne du côté serveur la sauvegarde de l'état courant des webParts pour l'utilisateur couramment authentifié.

## *Création d'une page avec des webParts*

Une webPart peut contenir un contrôle serveur standard ou un contrôle serveur utilisateur. Une webPart est matérialisée du côté serveur par une instance d'une classe qui dérive de `System.Web.UI.WebControls.WebParts.WebPart`. ASP.NET 2.0 utilise automatiquement et implicitement une instance de la classe `GenericWebPart` pour contenir un contrôle standard ou un contrôle utilisateur défini dans un fichier `.ascx`. Un contrôle utilisateur défini directement dans un fichier C# (comme celui de l'Exemple 23-31) doit lui-même être une webPart. En conséquence, pour pouvoir être utilisé comme webPart sa classe doit dériver de la classe `WebPart` plutôt que de la classe `Control`.

Comme nous l'avons vu, pour pouvoir modifier les webParts sur une page il faut qu'un utilisateur soit authentifié. Cela ne suffit pas cependant pour sauver l'état des webParts. Il faut prévoir un fournisseur de webParts dans le fichier de configuration comme ceci :

*Exemple :* *Web.Config*

```
<?xml version="1.0"?>
<configuration
     xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <connectionStrings>
      <add name="MyLocalSqlServer"
           connectionString="Data Source=localhost;Integrated
                     Security=SSPI ;Initial Catalog=MyWebSiteUsers;" />
   </connectionStrings>
   <system.web>
      <webParts>
         <personalization defaultProvider="MyWebPartsSqlProvider">
            <providers>
               <add    name=" MyWebPartsSqlProvider "
       type="System.Web.UI.WebControls.WebParts.SqlPersonalizationProvider,
                     System.Web, Version=2.0.0.0, Culture=neutral,
                     PublicKeyToken=b03f5f7f11d50a3a"
                     connectionStringName="MyLocalSqlServer" />
            </providers>
         </personalization>
      </webParts>
...
```

Puisque le *design pattern provider* est utilisé vous pouvez bien entendu prévoir vos propres fournisseurs de webParts avec des classes qui dérivent de la classe `System.Web.UI.WebControls.WebParts.PersonalizationProvider`. Ainsi les données relatives aux états des webParts des pages accessibles à un utilisateur donné peuvent être sauvées à un autre endroit que les données personnelles de cet utilisateur.

Pour qu'une page puisse présenter des webParts, il faut qu'elle contienne un contrôle serveur de type `WebPartManager` avant la déclaration de tous contrôles serveurs relatifs aux webParts. Vient alors la notion de *webPartsZone*. Une webPartsZone est un contrôle serveur qui à la particularité de pouvoir contenir des webParts. Précisons qu'une webParts est forcément contenu dans une webPartsZone. La page suivante contient une webPartsZone qui elle-même contient trois webParts : une webPart qui contient un contrôle utilisateur (celui défini par l'Exemple 23-33),

une webPart qui contient un contrôle serveur de type `GridView` et une webPart qui contient un contrôle serveur de type `Calendar` :

*Exemple 23-91 :*

```
<%@ Page Language="C#" %>
<%@ Register TagPrefix="PRATIQUE" Src="~/Securized/MyUserCtrl.ascx"
             TagName=UserCtrl %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
   <form id="Form1" runat="server">
      <asp:SqlDataSource ID="DataSrc" runat="server" ConnectionString=
         "server = localhost ; uid=sa ; pwd= ; database = ORGANISATION"
       SelectCommand="SELECT * FROM EMPLOYES" />
      <asp:WebPartManager ID="WebPartManager1" runat="server"/>
      <asp:WebPartZone ID="WebPartZone1" runat="server"
                       LayoutOrientation="Horizontal">
        <ZoneTemplate>
          <PRATIQUE:UserCtrl ID="UserCtrl1" runat="server" />
          <asp:GridView ID="Grid" DataSourceID="DataSrc" runat="server"/>
          <asp:Calendar ID="Calendar1" runat="server"></asp:Calendar>
        </ZoneTemplate>
      </asp:WebPartZone>
   </form>
</body>
</html>
```

La Figure 23-30 représente une copie d'écran de cette page. Notez que nous avons choisi un style professionnel pour afficher les webParts. Voici les changements qu'un tel style implique au code de notre page. Pour plus de clarté, nous omettrons ce code nécessaire aux styles dans nos prochains listings. En outre, vous aurez rarement le besoin d'accéder au code des pages qui contiennent des webParts puisque le mode *design* de *Visual Studio* est particulièrement efficace :

*Exemple 23-92 :*

```
<%@ Page Language="C#" %>
...
<body>
   ...
      <asp:WebPartZone ID="WebPartZone1" runat="server"
                LayoutOrientation="Horizontal" BorderColor="#CCCCCC"
                Font-Names="Verdana" Padding="6">
        <ZoneTemplate> ... </ZoneTemplate>
        <PartChromeStyle BackColor="#F7F6F3" BorderColor="#E2DED6"
                         Font-Names="Verdana" ForeColor="White" />
        <MenuLabelHoverStyle ForeColor="#E2DED6" />
        <EmptyZoneTextStyle Font-Size="0.8em" />
        <MenuLabelStyle ForeColor="White" />
        <MenuVerbHoverStyle BackColor="#F7F6F3" BorderColor="#CCCCCC"
           BorderStyle="Solid" ForeColor="#333333" BorderWidth="1px" />
        <HeaderStyle Font-Size="0.7em" ForeColor="#CCCCCC"
```

```
                    HorizontalAlign="Center" />
        <MenuVerbStyle BorderColor="#5D7B9D" BorderStyle="Solid"
                       BorderWidth="1px" ForeColor="White" />
        <PartStyle Font-Size="0.8em" ForeColor="#333333" />
        <TitleBarVerbStyle Font-Size="0.6em" Font-Underline="False"
                           ForeColor="White" />
        <MenuPopupStyle BackColor="#5D7B9D" BorderColor="#CCCCCC"
                        BorderWidth="1px" Font-Names="Verdana"
            Font-Size="0.6em" />
        <PartTitleStyle BackColor="#5D7B9D" Font-Bold="True"
                        Font-Size="0.8em" ForeColor="White" />
      </asp:WebPartZone>
...
</body>
```

La copie d'écran de la Figure 23-33 montre qu'à ce stade on peut déjà minimiser/restaurer ou fermer une webPart. L'état du chrome de chaque webPart (minimisé/restauré, ouvert/fermé) est sauvé pour l'utilisateur authentifié dans la base de données spécifiée par le fournisseur de webParts. À son retour sur cette page, l'utilisateur retrouvera les chromes des webParts dans l'état dans lesquels il les a laissé. Comprenez bien que seul l'état d'une webPart est sauvé. Pour vous en convaincre, modifier l'état d'un contrôle et vérifier que cet état est perdu lorsque vous vous ré authentifiez.

Une question se pose : comment faire pour définir les attributs d'une webPart tels que son titre ou la description qui doit être affichée dans le tooltip ? En effet, dans notre page `.aspx` nos trois contrôles sont implicitement contenus dans des contrôles de type `GenericWebPart`. Puisque nous n'avons pas accès à ces contrôles, nous ne pouvons pas les paramétrer. Deux solutions existent à ce problème :

- Dans le cas de contrôles standard ou `.aspx` nous sommes obligé de fixer des attributs qui seront exploités par le `GenericWebPart` par exemple comme ceci :

  ```
  ...
  <ZoneTemplate>
     <PRATIQUE:UserCtrl title="Couleurs" ID="UserCtrl1" runat="server" />
     <asp:GridView title="Employés" ID="Grid" DataSourceID="DataSrc"
                   runat="server"/>
      <asp:Calendar title="Calendrier" ID="Calendar1"
                    runat="server"></asp:Calendar>
  </ZoneTemplate>
  ...
  ```

  On note que l'intellisense ne connaît pas ces attributs et que le compilateur génère un avertissement.
- Dans le cas d'un contrôle utilisateur vous pouvez implémenter l'interface `IWebPart` qui définit les attributs sous forme de propriétés.

## *Le mode design*

Le mode d'une page contenant des webParts est défini par la propriété `DisplayMode{get;set;}` du contrôle `WebPartManager` de la page. Par défaut ce mode est `BrowseDisplayMode` et l'utilisateur ne peut que minimiser/restaurer et fermer les webParts. Si vous passez en mode

DesignDisplayMode, il devient possible de changer l'ordre des webParts (comme illustré par la Figure 23-31). Ici aussi, chaque changement génère un évènement *postback* qui permet de sauver le nouvel état des webParts.

*Exemple 23-93 :*

```
<%@ Page Language="C#" %>
...
<script runat="server">
   protected void ButtonBrowse_Click(object sender, EventArgs e) {
      WebPartManager1.DisplayMode = WebPartManager.BrowseDisplayMode;
   }
   protected void ButtonDesign_Click(object sender, EventArgs e) {
      WebPartManager1.DisplayMode = WebPartManager.DesignDisplayMode;
   }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
   <form id="Form1" runat="server">
      ...
      <asp:Button ID="ButtonBrowse" runat="server"
                  OnClick="ButtonBrowse_Click" Text="Browse Mode" />
      <asp:Button ID="ButtonDesign" runat="server"
                  OnClick="ButtonDesign_Click" Text="Design Mode" />
   </form>
</body>
</html>
```

## *Catalogue de webParts*

Nous avons vu que le chrome permet de rendre invisible une webPart après avoir sélectionné le verbe *Close*. Vous pouvez vous servir d'un contrôle de type PageCatalogPart pour rendre visible une ou plusieurs webParts qui ont été fermées. Un tel contrôle affiche la liste des webParts fermées que contient la page courante. Il doit être contenu dans un contrôle de type CatalogZone. Un CatalogZone n'est affiché que lorsque l'on est en mode CatalogDisplayMode :

*Exemple 23-94 :*

```
<%@ Page Language="C#" %>
...
<script runat="server">
   protected void ButtonCatalog_Click(object sender, EventArgs e) {
      WebPartManager1.DisplayMode = WebPartManager.CatalogDisplayMode;
   }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
   <form id="Form1" runat="server">
      ...
      <asp:CatalogZone ID="CatalogZone1" runat="server">
```

```
         <ZoneTemplate>
            <asp:PageCatalogPart ID="part1" runat="server"/>
         </ZoneTemplate>
      </asp:CatalogZone>
      <asp:Button ID="ButtonCatalog" runat="server"
                  OnClick="ButtonCatalog_Click" Text="Catalog Mode" />
   </form>
</body>
</html>
```

La notion de catalogue va beaucoup plus loin que la simple réapparition des webParts fermées. Vous pouvez déclarer un catalogue de contrôles que l'utilisateur peut potentiellement ajouter dans votre page au moyen d'un contrôle de type `DeclarativeCatalogPart`. Ce type de contrôle contient des définitions de webParts, un peu comme un `WebPartZone`.

Grâce au type de contrôles `ImportCatalogPart`, vous pouvez même autoriser un utilisateur à importer des webParts définies dans un document XML. Veuillez vous référer à la documentation de ce contrôle dans les **MSDN** pour connaître le format d'un tel fichier.

Ces deux contrôles permettent donc d'ajouter dynamiquement des webParts. Notons qu'en interne, du côté serveur, un identificateur unique est assigné à chaque webPart. Les webParts ajoutées dynamiquement présentent le verbe `Delete` qui permet de les détruire. Comme le montre la page suivante, il est aussi possible d'ajouter dynamiquement des webParts à partir du code :

*Exemple 23-95 :*

```
<%@ Page Language="C#" %>
<script runat="server">
   private static int calendarID = 0 ;
   protected void ButtonAdd_Click(object sender, EventArgs e) {
      Calendar calendar = new Calendar();
      calendar.ID = "Calendar_" + calendarID++;
      GenericWebPart wrapper = WebPartManager1.CreateWebPart(calendar);
      WebPartManager1.AddWebPart(wrapper, WebPartZone1, 0);
   }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
   <form id="Form2" runat="server">
      <asp:SqlDataSource ID="DataSrc" runat="server" ConnectionString=
         "server = localhost ; uid=sa ; pwd= ; database = ORGANISATION"
       SelectCommand="SELECT * FROM EMPLOYES" />
      <asp:WebPartManager ID="WebPartManager1" runat="server"/>
      <asp:WebPartZone ID="WebPartZone1" runat="server"
                       LayoutOrientation="Horizontal"/>
      <asp:Button id="ButtonAdd" runat="server"
            Text="Ajout d'un calendrier" OnClick="ButtonAdd_Click"/>
   </form>
</body>
</html>
```

## *Le mode édition*

Le mode d'édition permet d'éditer tous les attributs des webParts d'une page. Pour cela il faut passer en mode `EditDisplayMode`. Les contrôles contenus dans les contrôles de type `EditorZone` sont alors visibles. Ils peuvent être de trois types :

- `AppearanceEditorPart` (illustré par la Figure 23-34) : Permet d'éditer les propriétés visuelles d'une webPart telles que sont titre ou sa taille.
- `BehaviorEditorPart` : Permet d'éditer les propriétés comportementales d'une webPart telles que les verbes disponibles à partir de son chrome.
- `LayoutEditorPart` : Permet d'éditer l'organisation des webParts comme par exemple l'ordre d'affichage au sein d'une zone. Cet éditeur est utile car certains navigateurs ne présentent pas de facilités de type *glisser-déposer* comparables à celles de IE pour organiser les webParts.

La page suivante montre comment déclarer un contrôle de type `EditorZone` :

*Exemple 23-96 :*

```
<%@ Page Language="C#" %>
...
<script runat="server">
   protected void ButtonEdit_Click(object sender, EventArgs e) {
      WebPartManager1.DisplayMode = WebPartManager.EditDisplayMode;
   }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
   <form id="Form1" runat="server">
      ...
      <asp:EditorZone ID="EditorZone1" runat="server">
         <ZoneTemplate>
            <asp:AppearanceEditorPart ID="AppearanceEditorPart1"
                                      runat="server" />
         </ZoneTemplate>
      </asp:EditorZone>
      <asp:Button ID="ButtonEdit" runat="server"
                  OnClick="ButtonEdit_Click" Text="Edition Mode" />
   </form>
</body>
</html>
```

## *Connexion entre webParts*

Vous avez la possibilité de créer une connexion entre deux webParts. Ce type de connexion est dissymétrique : tandis qu'une webPart joue le rôle de producteur l'autre joue le rôle de consommateur. Illustrons cette possibilité à l'aide d'une page qui utilise un contrôle serveur utilisateur producteur permettant de choisir une couleur et un contrôle serveur utilisateur consommateur qui affiche la couleur sélectionnée. Cette page est illustrée par la Figure 23-35 :

*Exemple 23-97 :* *MyProviderCtrl.ascx*

```
<%@ Control Language=C#  CodeFile="~/Securized/MyProviderCtrl.ascx.cs"
                Inherits="MyUserCtrl" %>
Couleur : <asp:dropdownlist id="Couleur" runat="server">
            <asp:listitem>blanc</asp:listitem>
            <asp:listitem>noir</asp:listitem>
          </asp:dropdownlist>
<asp:button id="Button1" text="Soumettre" OnClick="Btn_Click"
   runat="server"/>
```

*Exemple 23-98 :* *MyProviderCtrl.ascx.cs*

```
using System.Web.UI ;
using System.Web.UI.WebControls.WebParts ;
public interface IColor { string SelectedColor { get;set ; } }
public partial class MyProviderCtrl : UserControl, IColor {
    private string m_SelectedColor ;
    public string SelectedColor {
        get { return m_SelectedColor ; }
        set { m_SelectedColor = value ; }
    }
    protected void Btn_Click(System.Object sender, System.EventArgs e){
        m_SelectedColor = Couleur.SelectedItem.Value;
    }
    [ConnectionProvider("TestProviderConsumer")]
    public IColor ProvideColor() { return this ;  }
}
```

Le contrôle serveur qui représente la webPart producteur (en l'occurrence `MyProviderCtrl`) doit supporter une interface connue du contrôle serveur qui représente le consommateur (en l'occurrence `MyProviderCtrl`). L'interface `IColor` joue ici ce rôle d'intermédiaire :

*Exemple 23-99 :* *MyConsumerCtrl.ascx*

```
<%@ Control Language="C#"  ClassName="MyConsumerCtrl" %>
<script runat="server">
    [ConnectionConsumer("TestProviderConsumer")]
    public void Consume(IColor colorProvider){
        Msg.Text = "Vous avez sélectioné : " + color.SelectedColor ;
    }
</script>
<asp:Label ID="Msg" runat="server" />
```

On voit qu'il faut utiliser l'attribut `ConnectionProvider` pour indiquer la méthode qu'ASP.NET doit utiliser pour obtenir l'objet représentant la webPart producteur. De même, on indique la méthode du contrôle consommateur qu'ASP.NET doit invoquer au moyen de l'attribut `ConnectionConsumer`. À ce stade, si vous construisez une page avec ces deux contrôles contenus dans deux webParts rien ne se passe. En effet, pour que deux webParts puissent communiquer, il faut créer une connexion entre elles. Vous pouvez créer une telle connexion au moyen d'un contrôle `WebPartConnection` comme ceci :

*Exemple 23-100 :*

```
<%@ Page Language="C#" %>
<%@ Register TagPrefix="PRATIQUE" Src="~/Securized/MyProviderCtrl.ascx"
             TagName="ProviderCtrl" %>
<%@ Register TagPrefix="PRATIQUE" Src="~/Securized/MyConsumerCtrl.ascx"
             TagName="ConsumerCtrl" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
   <form id="Form1" runat="server">
      <asp:WebPartManager ID="WebPartManager1" runat="server">
         <StaticConnections>
            <asp:WebPartConnection ID="MyCnx" ConsumerID="ConsumerCtrl1"
                                              ProviderID="ProviderCtrl1" />
         </StaticConnections>
      </asp:WebPartManager>
      <asp:WebPartZone ID="WebPartZone1" runat="server"
                       LayoutOrientation="Horizontal">
        <ZoneTemplate>
          <PRATIQUE:ProviderCtrl ID="ProviderCtrl1" runat="server" />
          <PRATIQUE:ConsumerCtrl ID="ConsumerCtrl1" runat="server" />
        </ZoneTemplate>
      </asp:WebPartZone>
   </form>
</body>
</html>
```

Cette page illustre une connexion construite statiquement. Vous pouvez aussi permettre à vos utilisateurs de créer dynamiquement des connexions. Pour cela, il faut passer en mode `ConnectDisplayMode` et fournir un contrôle de type `ConnectionZone` :

*Exemple 23-101 :*

```
<%@ Page Language="C#" %>
...
<script runat="server">
   protected void ButtonBrowse_Click(object sender, EventArgs e) {
      WebPartManager1.DisplayMode = WebPartManager.BrowseDisplayMode ;
   }
   protected void ButtonCnx_Click(object sender, EventArgs e) {
      WebPartManager1.DisplayMode = WebPartManager.ConnectDisplayMode ;
   }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
   <form id="Form1" runat="server">
      <asp:WebPartManager ID="WebPartManager1" runat="server"/>
      ...
      <asp:ConnectionsZone ID="ConnectionsZone1" runat="server"/>
      <asp:Button ID="ButtonBrowse" runat="server"
                  OnClick="ButtonBrowse_Click" Text="Browse Mode" />
```

```
      <asp:Button ID="ButtonDesign" runat="server"
                  OnClick="ButtonCnx_Click" Text="Connexion Mode" />
   </form>
</body>
</html>
```

Si l'utilisateur souhaite créer une connexion impliquant une webPart, il lui suffit de sélectionner le verbe `Connect` sur cette webPart. Du fait de la présence d'un des attributs `ConnectionProvider` ou `ConnectionConsumer`, ASP.NET sait reconnaître si la webPart est de type producteur ou consommateur. Le contrôle `ConnectionZone` est alors affiché et propose à l'utilisateur de sélectionner la webPart avec laquelle notre première webPart va être connectée. Ici encore, une grande part de l'intérêt de cette technique réside dans le fait que l'existence de connexions créées dynamiquement est sauvée du côté serveur pour chaque utilisateur.

# 24

# *Introduction au développement de Services Web avec .NET*

## *Introduction*

### *SOA : Architecture Orientée Service*

La programmation et l'architecture orientée service (*SOA* pour *Service Oriented Architecture*) est un complément récent de la POO (Programmation Orientée Objet). La notion de *service* est simple : un service est une application avec laquelle on interagit avec des messages. Un service a donc des clients qui lui envoient des messages et auxquels il peut retourner des messages. Un service peut aussi être le client d'un autre service. SOA est fondé sur quatre principes clés :

- Les **frontières** d'un service sont **explicites** :
  Chaque message envoyé à ou reçu d'un service représente un coût en terme de performance puisque cela nécessite l'utilisation d'un réseaux. Les différentes technologies d'objets distribués (telle que .NET Remoting ou Java RMI) cachent au développeur le fait qu'un objet est distant en convertissant implicitement un appel à une de ses méthodes en un message envoyé sur le réseau. L'approche SOA est diamétralement opposée. Il est nécessaire que le développeur soit conscient de l'envoi ou de la réception d'un message. Nous verrons que cette approche permet notamment d'autres modèles d'échange de messages que le traditionnel requête/réponse synchrone utilisé pour simuler un appel de méthode sur un objet distant.
- Un service est une **entité autonome** :
  La notion d'autonomie d'un service découle de règles radicalement différentes de ce qui se pratique traditionnellement en POO. Le **déploiement** et l'évolution d'un service se font indépendamment de ses clients. Il est de la responsabilité des clients de s'adapter à une

nouvelle **version** d'un service. De cette façon la topologie d'un ensemble de services interagissant s'adapte naturellement aux nouveaux besoins. Cela permet aussi de mettre en évidence l'importance de la **gestion des erreurs** puisque celles-ci surviendront certainement. De même, on ne peut pas se passer d'une gestion sérieuse de la **sécurité** puisque les services sont en général accessibles à partir d'un réseau public et donc peu sûr, tel que *Internet*.

- La **structure d'utilisation** d'un service est **définie sans ambiguïté** par un **unique contrat présenté par le service à ses clients** :

  L'évolution des langages objets illustre l'importance croissante de la dissociation entre abstraction et implémentation. En effet, la notion d'interface est maintenant une composante privilégiée des langages modernes tels que C# ou Java, ce qui n'est pas le cas par exemple en C++. SOA va plus loin en formalisant la structure d'utilisation d'un service dans un unique contrat matérialisé par un document XML. Ce contrat spécifie entre autres la structure des informations contenues dans les messages entrant et sortant au moyen de schémas XSD. Ce contrat peut contenir des commentaires et ainsi renseigner un humain sur la structure d'utilisation du service. Ce contrat peut aussi être utilisé par un programme pour générer par exemple, des classes qui seront utilisées côté client pour fabriquer les messages et les envoyer au service. Le fait que la réception d'un message côté service déclenche en interne l'appel à une certaine méthode d'une certaine classe est un détail d'implémentation du service et ne fait pas partie de son contrat.

- La **sémantique d'utilisation** d'un service, aussi appelé ***stratégie*** *(policy* en anglais*)* est **définie sans ambiguïté** dans le contrat présenté par le service à ses clients :

  La plupart des langages objets ne présentent pas de mécanismes simples pour signifier aux clients d'une interface les préconditions et les postconditions d'utilisation. Ces conditions sont vérifiées au sein de l'implémentation de l'interface et si elles ne sont pas rigoureusement documentées, le client n'a aucun moyen de les connaître avec exactitude. En SOA, cette sémantique d'utilisation fait partie intégrante du contrat présenté par le service à ses clients. Par exemple un service peut déclarer avec une stratégie qu'il n'accepte de traiter un message entrant qu'à la condition que celui-ci soit encrypté d'une certaine manière. Un service peut déclarer avec une stratégie qu'il n'est accessible que de 8h à 22h GMT ou qu'il supporte un certain protocole transactionnel. À l'instar de la structure d'utilisation, les stratégies sont rédigées dans un langage XML non ambiguë. Elles peuvent donc être consommées par un *framework* pour éviter au développeur de coder la logique spécifiée (par exemple un message sera automatiquement crypté d'une certaine manière côté client par un tel *framework* si telle est la stratégie du service ciblé).

## *Services Web : les langages SOAP et WSDL*

Les *services web* représentent la manière privilégiée à l'heure actuelle pour implémenter la notion de service telle que nous venons de la définir. Les services web ont deux caractéristiques essentielles :

- **Indépendance de la plateforme d'exécution du client et du service web** : Tout a été fait pour que les messages échangés entre un service web et ses clients soient complètement indépendants des plateformes d'exécutions impliquées. Les services web constituent le moyen privilégié pour faire circuler l'information dans un environnement hétérogène. Concrètement, un service web implémenté en Java fonctionne de la même manière avec un client écrit en C# ou un client écrit en Java. On parle d'**interopérabilité**.

- **Indépendance du protocole de transport des messages** : Concrètement, un service web se comportera de la même façon indépendamment du fait que les messages soient acheminés par un des protocoles HTTP, HTTPS, TCP/IP, UDP/IP ou SMTP. En conséquence, les services web ne font aucune hypothèse sur les caractéristiques propres du protocole sous-jacent (comme l'encryptions des données avec HTTPS ou la fiabilité avec TCP). Nous verrons comment la plupart des caractéristiques de transport peuvent être obtenues au niveau de l'échange de message, indépendamment du protocole de transport sous-jacent. Enfin, ne vous y trompez pas, l'expression *service web* contient le mot *web* parce que le protocole HTTP (i.e le protocole du web), est le plus utilisé au monde. Le fait de pouvoir se placer dans la continuité du succès phénoménal du web constitue certainement un atout décisif pour l'adoption des services web.

Pour pouvoir obtenir l'interopérabilité, les services web reposent d'abord sur deux langages XML simples et normalisés. En effet, en page 759 nous expliquons qu'un atout majeur de XML est de pouvoir coder des données indépendamment d'une plateforme d'exécution :

- Le langage XML ***SOAP*** (*Simple Object Access Protocol*) est utilisé pour la rédaction des **messages** échangés entre services. Des spécifications existent pour chaque protocole réseaux pour décrire comment acheminer un message SOAP.
- Le langage XML ***WSDL*** (*Web Service Definition Language*) est utilisé pour rédiger les **contrats** présentés par les services à leurs clients. Nous verrons que ce langage permet aussi de définir les liens entre le contrat d'un service et son implémentation (autrement dit, il indique quelle méthode de quelle classe doit être invoquée pour traiter tel type de message). Ainsi, le contrat que présente un service à ses clients ne constitue qu'une partie d'un document WSDL puisque nous avons vu que ce dernier type d'information n'a pas à être consommé par les clients.

## *WS-I Basic profiles*

La caractéristique principale des langages SOAP et de WSDL est qu'ils sont extensibles. Concrètement, les plus gros acteurs du marché tel que *Microsoft*, *IBM* ou *BEA* se sont réunis au sein d'une organisation nommée *WS-I* (*Web Service Interoperabilityy*) pour produire un ensemble de spécifications qui étendent ces langages. Cet ensemble est nommé *WS-** (prononcé *WS Star*) car les noms de ces spécifications commencent par *WS-* puis se termine par une expression illustrant le dessein. Par exemple, on comprend que la spécification *WS-Security* permet d'introduire un certain niveau de sécurité dans l'échange des messages. Voici les grands domaines couverts à l'heure actuels (fin 2005) par les spécifications *WS-** :

- **Sécurité** : Plusieurs spécifications WS-* permettent d'implémenter différentes facettes de la sécurité telles que l'authentification, la confidentialité des données ou les sessions sécurisées d'échange de messages.
- **Description** : Le langage WSDL est étendu par plusieurs spécifications WS-* qui permettent de rédiger des contrats plus précis. Par exemple, la spécification *WS-SecurityPolicy* permet au contrat d'un service de spécifier qu'il n'accepte de traiter un message reçu que si ce dernier a été crypté d'une certaine manière.
- **Découverte** : Des spécifications décrivent des implémentations de mécanismes de recherche selon des critères, de services web au travers d'un réseau. Ainsi un client peut découvrir l'ensemble des services qui répondent à ses besoins. Lorsque que le client décide

d'exploiter un service, il n'a plus qu'à examiner son contrat afin de comprendre comment l'utiliser.

- **Livraison de messages** : Des spécifications WS-* permettent de fournir certaines garanties quant à la livraison des messages. Cela permet de pallier le fait que le protocole réseau de transport des messages sous-jacent n'est pas nécessairement fiable. Par exemple le protocole UDP ne permet pas à une source de savoir qu'un destinataire a bien reçu un message. Le domaine de la livraison de messages couvre aussi les problématiques liées à un envoi de messages vers plusieurs destinataires (*broadcasting*) ainsi que les problématiques liées à la consommation d'un même message par plusieurs services qui sont destinés à travailler « à la chaîne » (en *pipeline*).
- **Coordination et transaction** : Des spécifications WS-* permettent de coordonner les activités de plusieurs services afin d'obtenir un certain comportement global. La notion de transaction distribuée entre plusieurs services constitue un exemple d'une telle coordination.

Un aspect très intéressant des spécifications WS-* est qu'elles permettent de composer les protocoles. Si deux protocoles ont des desseins orthogonaux, alors ils peuvent être utilisés conjointement d'une manière indépendante. Par exemple, un protocole de sécurité destiné à rendre les messages confidentiels en les cryptant peu optionnellement être utilisé conjointement avec un protocole de livraison de message permettant de garantir au client que les messages ont bien été reçus par le service.

Cependant comprenez bien que l'indépendance dans la composition des protocoles se fait dans la mesure du possible. Les spécifications WS-* agissent à différents niveaux de l'architecture. Il n'est pas rare qu'une spécification WS-* compte sur une autre spécification d'un niveau inférieur. Par exemple, les algorithmes de transaction distribuée sur plusieurs services présupposent que la livraison des messages se fait dans l'ordre dans lequel ils sont envoyés.

## *Modèle d'échange de messages*

Du fait qu'ils permettent l'interopérabilité entre plateformes et s'affranchissent des caractéristiques des protocoles réseaux sous-jacents, nous avons vu que les services web représentent une évolution majeure par rapport aux autres technologies d'échange d'information entre application. Contrairement à la plupart des ces technologies, la notion de service rend possible aussi différents modèles d'échanges de messages (en anglais *Message Exchange Pattern* ou *MEP*). En effet, jusqu'ici les développeurs étaient contraints par ces technologies à utiliser principalement le modèle d'échange requête/réponse synchrone. Ce modèle a été popularisé par la technologie RPC (*Remote Procedure Call*). Il consiste à simuler du côté client l'appel d'une procédure en local en encapsulant les données entrantes dans un message requête envoyé au travers du réseaux, puis en attendant un message réponse contenant les données sortantes. Les langages SOAP et WSDL des services web permettent d'implémenter tous les modèles d'échanges de messages imaginables tels que :

- Le modèle style RPC **requête/réponse synchrone** que nous venons de décrire.
- Le modèle **requête/réponse asynchrone**. La différence avec le modèle précédent est que le thread du client responsable de l'envoi du message requête reprend sa course directement après l'envoi. Le client doit mobiliser des ressources afin d'assurer la réception du message réponse. Ce modèle est adapté lorsque le traitement du message requête par le service peut prendre un certain temps.

- Le modèle **requête/polling de la réponse**. La différence avec le modèle précédent est qu'après l'envoi du message requête, le client demande périodiquement la réponse au service au moyen d'un échange de message requête/réponse jusqu'à ce qu'il obtienne la réponse ou une erreur.
- Le modèle **oneway**. Dans ce modèle, le client n'attend aucune information du service. Il ne fait qu'envoyer son message. Ce modèle est adapté à l'envoi d'information non cruciale telles que des informations de log.
- Le modèle **multi-diffusion** (dit aussi **broadcast**) permet d'envoyer un même message simultanément à plusieurs services. Des variantes de ce modèle permettent au client de s'attendre à une ou plusieurs réponses de la part des services contactés.
- Le modèle **abonnement/notification** (dit aussi d'**évènement**) permet à un client de s'abonner à un service. Il recevra alors les messages fabriqués par le service qui correspondent à ses besoins. Plusieurs variantes de ce modèle existent pour spécifier comment le service obtient la notification de désabonnement du client.

Ces modèles représentent la plupart des échanges de messages réels mais bien d'autres modèles d'échange plus ou moins compliqués sont imaginables et implémentables.

# *Développement d'un service web simple*

## *Développement d'un service web simple sans utiliser Visual Studio .NET*

Un service web simple, est un service web qui ne supporte que le modèle d'échange de messages style RPC au dessus du protocole HTTP et qui n'exploite aucune des spécifications WS-*. Il n'est pas nécessaire d'avoir recours à la plateforme de développement WSE pour fabriquer de tels services web. Il est préférable d'avoir assimilé les principales notions du développement ASP .NET avant de continuer.

En .NET, le code source d'un service web est un fichier d'extension `.asmx` stocké dans un répertoire hébergeant le service. Pour notre exemple ce répertoire se nommera `localizationcorp`. Le code source contenu dans une page `asmx` sera automatiquement compilé avant son exécution, grâce à ASP.NET. Un assemblage sera produit dans le sous répertoire `bin`.

Dans ce chapitre nous nous baserons sur un service web de géo localisation qui présente une opération unique : la possibilité d'obtenir un couple ville/pays à partir d'un couple de coordonnées latitude/longitude. Cette opération n'a clairement besoin que du modèle d'échanges de messages style RPC synchrone. Aussi, le concept de méthode d'une classe convient parfaitement à son implémentation. Une méthode qui représente l'implémentation d'une telle opération d'un service est en général qualifiée de *web méthode*. Comprenez bien que les implémentations des autres modèles d'échanges de messages ne peuvent pas être aussi simples.

Par souci de simplicité nous ne nous limiterons qu'à la localisation de la ville de Nice. Voici le code de ce service web :

*Exemple 24-1 :* *`Localizer.asmx`*

```
<%@ WebService language="C#" class="LocalizationCorp.Localizer" %>
```

```
using System ;
using System.Web.Services ;
using System.Xml.Serialization ;
namespace LocalizationCorp {
   public class Localizer : WebService {
      [XmlRoot(Namespace="http://localizationcorp.com/documents/data/")]
      public class Town {
         public string Name ;
         public string Country ;
      }
      [WebMethod]
      public Town GetTownFromLatLon(double lat, double lon) {
         Town town = new Town() ;
         if (lat < 43.44 && lat > 43.39 && lon < 7.18 && lon > 7.10) {
            town.Name = "Nice" ; town.Country = "France" ;
         } else {
            town.Name = "Unkown" ; town.Country = "Unkown" ;
         }
         return town ;
      }
   }
}
```

La première ligne indique à ASP.NET que ce fichier décrit un service web. À l'exécution, un service web est en fait une instance de la classe spécifiée en première ligne avec l'attribut `class`, en l'occurrence, la classe `LocalizationCorp.Localizer`. Une telle instance est créée par ASP.NET pour traiter chaque requête. On parle d'environnement sans état puisqu'un même objet ne peut servir plusieurs requêtes. Comprenez bien que selon vos besoin vous pouvez toujours stocker des états persistant entre les requêtes du côté serveur, par exemple au moyen d'une base de données.

Dans cette première ligne, nous indiquons à ASP.NET le langage .NET utilisé pour rédiger cette classe, avec l'attribut `Language`, en l'occurrence le langage C#. Tous comme une page ASPX, on peut séparer le code source C# des pages ASMX. Nous expliquons ceci en page 868.

On remarque que notre classe dérive de la classe `System.Web.Services.WebService`. Une classe représentant un service web ne dérive pas nécessairement de la classe `WebService`. Cependant, cela permet l'accès à de nombreuses fonctionnalités, comme la gestion des états d'une page sur l'autre. Une classe représentant un service web doit satisfaire les deux contraintes suivantes :

- La classe doit être publique et doit avoir un constructeur par défaut public.
- Toutes les web méthodes doivent être marquées avec l'attribut `System.Web.Services.WebMethod`. Les propriétés de cet attribut configurent les fonctionnalités accessibles à cette méthode, comme son comportement relatif aux sessions. La propriété `Description` permet de fournir une description pour la méthode. Veuillez consulter l'article **WebMethodAttribute Members** des **MSDN** pour plus de détails.

La propriété `Namespace` de la classe `WebService` permet de nommer d'une manière unique votre service web. En général, on fournit une URL pour cette propriété mais ceci n'est pas une obligation. Si on ne fournit pas explicitement d'URL, l'URL *http://tempuri.org/* est prise par défaut.

Dans tous les cas, la ressource localisée à l'URL (si elle existe) ne sera jamais accédée. Le fait d'utiliser une URL qui vous appartient permet de garantir qu'aucun autre service web au monde ne portera le même nom que le vôtre. Voici comment nous aurions pu écrire notre service web pour utiliser cette propriété :

*Exemple 24-2 :*

```
...
namespace LocalizationCorp {
   [WebService(Namespace="http://localizationcorp.com/localizer")]
   public class Localizer : WebService {
      ...
```

Vous pouvez remarquer qu'il n'y a pas de fichier d'extension WSDL présent dans le répertoire hébergeant le service web. Pour obtenir ce document, il suffit de taper l'URL de notre service web dans un navigateur, suivie de `?wsdl`. Dans notre exemple cela donne : `http://localhost/localizationcorp/localizer.asmx?wsdl`.

En fin de chapitre nous consacrons une section à la description des documents WSDL. Vous pouvez déjà remarquer que notre classe de données `LocalizationCorp.Town` a été convertie en un schéma XSD inclus dans le document WSDL et défini avec l'espace de noms `http://localizationcorp.com/documents/data/`.

## *Développement d'un service web simple avec Visual Studio*

*Visual Studio* présente de nombreuses facilités pour construire un service web ASP.NET simple.

Pour créer un service web avec *Visual Studio* il suffit de faire : *Fichier* ▶ *Nouveau* ▶ *Site Web* ▶ *Service Web ASP.NET* ▶ *Emplacement* = `http://localhost/localizationcorp` ▶ *OK*. Notre service web est hébergé sur la machine locale puisque nous avons précisé `localhost`.

Vous pouvez maintenant renommer le ficher `Service1.asmx` en `localizer.asmx` et recopier le code C# de l'Exemple 24-1 dans le fichier `localizer.asmx.cs` (accessible par *click droit sur le fichier* `localizer.asmx` ▶ *Afficher le code*).

# *Tester et déboguer un service web*

## *Tester un service web*

Grâce à ASP.NET, nous pouvons tester un service web sans avoir à écrire un client. Il suffit d'ouvrir un navigateur et d'accéder au fichier `asmx` avec l'URL `http://localhost/localizationcorp/localizer.asmx`. Une page HTML fabriquée automatiquement à partir du document WSDL présente les opérations du service. Vous pouvez sélectionner l'opération `GetTownFromLatLon`. Une nouvelle page HTML apparaît et vous permet de saisir les informations consommées par l'opération, en l'occurrence la latitude et la longitude. Si vous tapez des coordonnées géographiques correspondants à la ville de Nice (telles que latitude=43.42 et longitude=7.15) vous obtiendrez une page HTML contenant le document XML suivant :

```
<?xml version="1.0" encoding="utf-8" ?>
<Town xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
      xmlns="http://localizationcorp.com/documents/data/">
  <Name>Nice</Name>
  <Country>France</Country>
</Town>
```

Cette possibilité de test utilise la méthode `Documentation` pour générer les pages HTML. Si vous désactivez cette méthode vous n'avez plus accès aux tests à partir d'un navigateur. Cette remarque se révèlera pertinente lorsque l'on expliquera que la plateforme WSE compte sur la méthode `HttpSoap` pour effectuer son travail. Ainsi, les traitements de WSE ne peuvent être testés à partir de la méthode `Documentation`.

En production, pour plus de sécurité il est conseillé de désactiver les méthodes autres que `HttpSoap`. Cette désactivation peut s'effectuer en ajoutant les lignes suivantes dans le fichier `web.config` de votre service ou même mieux, dans votre fichier `machine.config` (qui peut être trouvé dans votre répertoire d'installation de .NET `C:\WINDOWS\Microsoft.NET\Framework\v2.0.XXXXX\CONFIG`) :

```
<configuration>
  <system.web>
    <webServices>
      <protocols>
        <add name="HttpSoap1.2" />
        <add name="HttpSoap" />
        <remove name="HttpPost" />
        <remove name="HttpGet" />
        <remove name="HttpPostLocalhost"/>
        <remove name="Documentation"/>
...
```

## *Déboguer un service web*

*Visual Studio* vous permet de déboguer un service web comme n'importe qu'elle application .NET. Si vous faite *déboguer* ▶ *démarrer* vous obtiendrez la page HTML de test. Vous pouvez maintenant déboguer votre service en lui envoyant des messages soit en utilisant les pages HTML de test, soit en utilisant n'importe quel autre client.

# *Créer un client .NET d'un service web*

## *Créer un client .NET d'un service web sans Visual Studio*

En général, pour développer un client d'un service web dont les opérations sont modélisables par un échange de messages style RPC, on se sert d'une classe appelée classe *proxy* du service. Cette classe est directement accédée par le code source du client. Pour chaque web méthode du service web, la classe proxy présente une méthode de même nom et qui accepte et retourne les mêmes arguments. La classe proxy s'occupe complètement de la construction et de l'envoi du message SOAP, et de la réception de la réponse.

Si votre client est développé avec .NET, vous vous servirez certainement de l'outil `wsdl.exe`, fourni avec *Visual Studio* pour générer votre classe proxy. Cet outil est capable de générer la classe

proxy dans un fichier source en langage C#, VB.NET ou JScript, directement à partir du contrat WSDL du service. Par exemple cette commande génère ce fichier :

```
wsdl.exe /out:LocalizerProxy.cs /language:C#
    http://localhost/localizationcorp/Localizer.asmx?wsdl
```

*Exemple :* *LocalizerProxy.cs*

```
//------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:2.0.XXXXX
//
//     Changes to this file may cause incorrect behavior and will be lost
//     if the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------
using System ;
using System.ComponentModel ;
using System.Diagnostics ;
using System.Web.Services ;
using System.Web.Services.Protocols ;
using System.Xml.Serialization ;

[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Web.Services.WebServiceBindingAttribute(
 Name="LocalizerSoap", Namespace="http://tempuri.org/")]
public partial class Localizer :
   System.Web.Services.Protocols.SoapHttpClientProtocol {
   ...
   public Localizer() {
      this.Url = "http://localhost/LocalizationCorp/Localizer.asmx" ;
   }

   [System.Web.Services.Protocols.SoapDocumentMethodAttribute(
      "http://tempuri.org/GetTownFromLatLon",
      ...)]
   [return: System.Xml.Serialization.XmlElementAttribute(
      Namespace="http://localizationcorp.com/documents/data/",
      IsNullable=true)]
   public Town GetTownFromLatLon(double lat, double lon) {
      object[] results = this.Invoke("GetTownFromLatLon", new object[] {
               lat,
               lon}) ;
      return ((Town)(results[0])) ;
   }

   public System.IAsyncResult BeginGetTownFromLatLon(
              double lat, double lon,
```

```
            System.AsyncCallback callback, object asyncState) {
      return this.BeginInvoke("GetTownFromLatLon", new object[] {
               lat, lon}, callback, asyncState) ;
   }

   public Town EndGetTownFromLatLon(System.IAsyncResult asyncResult) {
      object[] results = this.EndInvoke(asyncResult) ;
      return ((Town)(results[0])) ;
   }

   public void GetTownFromLatLonAsync(double lat, double lon) {
      this.GetTownFromLatLonAsync(lat, lon, null) ;
   }

   public void GetTownFromLatLonAsync(double lat, double lon,
                              object userState) {
      if ((this.GetTownFromLatLonOperationCompleted == null)) {
         this.GetTownFromLatLonOperationCompleted =
            new System.Threading.SendOrPostCallback(
              this.OnGetTownFromLatLonOperationCompleted) ;
      }
      this.InvokeAsync("GetTownFromLatLon", new object[] {
               lat,
               lon}, this.GetTownFromLatLonOperationCompleted,
               userState) ;
   }

   private void OnGetTownFromLatLonOperationCompleted(object arg) {
      if ((this.GetTownFromLatLonCompleted != null)) {
         System.Web.Services.Protocols.InvokeCompletedEventArgs
            invokeArgs =
             ((System.Web.Services.Protocols.InvokeCompletedEventArgs)
             (arg)) ;
         this.GetTownFromLatLonCompleted(this,
            new GetTownFromLatLonCompletedEventArgs(
                 invokeArgs.Results, invokeArgs.Error,
                 invokeArgs.Cancelled, invokeArgs.UserState)) ;
      }
   }

   public new void CancelAsync(object userState) {
      base.CancelAsync(userState) ;
   }

   public event GetTownFromLatLonCompletedEventHandler
      GetTownFromLatLonCompleted;

}
```

```
[System.SerializableAttribute()]
[System.Xml.Serialization.XmlTypeAttribute(
   Namespace="http://localizationcorp.com/documents/data/")]
public partial class Town {
    private string nameField ;
    private string countryField ;
    public string Name {
       get { returnthis.nameField;} set {this.nameField = value;}
    }
    public string Country {
       get {return this.countryField;} set {this.countryField = value;}
    }
}
...
```

Il est intéressant de remarquer que la classe `Town` a été régénérée à partir du schéma XSD présent dans le fichier WSDL du service. Certaine documentions conseillent de supprimer cette définition du fichier généré et d'encapsuler ces classes de données dans des bibliothèques référencées à la fois par le service et ses clients. Nous ne sommes pas d'accord avec cette pratique qui va complètement à l'encontre d'un des principes fondamentaux de l'approche SOA : **un service et ses clients ne partagent qu'un contrat**. Aussi dans la suite nous nous contenterons parfaitement de cette classe `Town` générée.

Notez que la classe proxy dérive de la classe `SoapHttpClientProtocol`. Cela lui confère un certains nombre de possibilités comme la configuration des méthodes d'authentification à utiliser.

Vous pouvez maintenant créer un assemblage qui référence l'assemblage `System.Web.Services.dll` et qui contient le fichier source `LocalizerProxy.cs` ainsi que le fichier source suivant :

*Exemple 24-3 :*

```
using System ;
class Program {
   static void Main() {
      Localizer proxy = new Localizer();
      Town town = proxy.GetTownFromLatLon(43.42, 7.15);
      Console.WriteLine("Ville:" + town.Name + " Pays:" + town.Country) ;
   }
}
```

## *Créer un client .NET d'un service web avec Visual Studio*

Pour utiliser un service web à partir d'une application développée avec *Visual Studio*, il suffit d'ajouter une référence web au projet vers le service web concerné. Pour cela, il faut cliquer droit sur le projet et sélectionner *Add Web Reference*.... Une fenêtre de recherche de service apparaît. À partir de cette fenêtre vous pouvez sélectionner un service web distant ou local. Lorsque la référence est ajoutée, *Visual Studio* utilise automatiquement et implicitement l'outil `wsdl.exe` pour construire le fichier source C# contenant la classe proxy. Ce nouveau fichier source vous est caché au niveau de l'explorateur de solution pour vous évitez de le modifier. Vous pouvez vérifier cependant qu'il est bien présent dans le répertoire du projet du client.

Notez que *Visual Studio* utilise la technologie UDDI pour rechercher les services web distants. Nous détaillons cette technologie un peu plus loin dans ce chapitre.

# *Appels asynchrones et modèle d'échange de message*

Remarquez que pour chaque web méthode web qui peut être invoquée, la classe proxy présente aussi une méthode pour les appels asynchrones (nommée `Begin[method]`) et une méthode (nommée `End[method]`) pour récupérer les arguments de retour d'un appel asynchrone à la web méthode concernée. Ce type d'appel asynchrone est décrit page 171.

Pour chaque web méthode, un autre modèle d'appel asynchrone est fourni grâce aux méthodes `[method]Async()`, `On[method]OperationCompleted()` et à l'évènement `[method]CompletedEventHandler`. Ce modèle supporte notamment l'annulation de toutes les opérations asynchrones de ce type en cours au moyen de la méthode `CancelAsync()`.

Comprenez bien que cette façon de programmer ne correspond pas à l'*esprit SOA*. En SOA, le paradigme client/serveur est plutôt remplacé par l'idée d'association expéditeur/destinataire propre à chaque message. Aussi, dans un échange de messages type requête/réponse (synchrone ou asynchrone) les rôles d'expéditeur/destinataire sont inversés selon le point de vue que l'on prend :

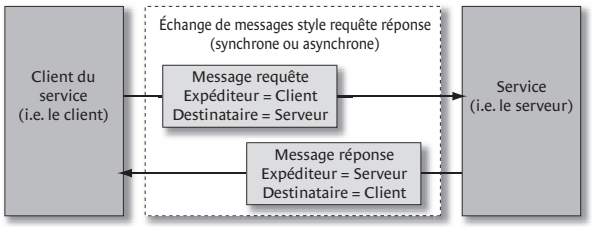


*Figure 24-1 : Client/Serveur vs. Expéditeur/Destinataire*

Autrement dit, pour être habilité à recevoir un message réponse, et plus généralement pour pouvoir supporter n'importe quel modèle d'échange de message, il faut que le client soit lui-même un service. La suite de notre exposé n'aura pas besoin de modèles d'échange de messages autres que le traditionnel style RPC.

# *Utiliser un service web à partir d'un client .NET Remoting*

Vous pouvez consommer un service web simple avec la technologie .NET Remoting. Pour cela, il suffit d'utiliser les métadonnées de type produites par l'outil `soapsuds.exe` (décrit page 804). Pour que `soapsuds.exe` puisse consommer un service web, il faut que celui-ci supporte le formatage RPC qui est déconseillé. Ce formatage ainsi que ses désavantages sont décrits dans la prochaine section consacrée à SOAP. Il faut alors utiliser l'attribut `SoapRpcService` sur la classe contenant les web méthodes de votre service. Le service web de l'Exemple 24-1 doit donc être réécrit comme ceci pour être consommable par `soapsuds.exe` :

*Exemple 24-4 :* *localizer.asmx*

```
...
namespace LocalizationCorp {
   [System.Web.Services.Protocols.SoapRpcService()]
   public class Localizer : WebService {
      ...
```

Vous pouvez maintenant obtenir des métadonnées exploitables par .NET Remoting dans l'assemblage `ProxyLocalizationCorp.dll` en tapant la ligne de commande suivante. Notez que pour exécuter cette commande sur un service web hébergé par IIS, il faut désactiver l'authentification intégrée de *Windows* sur le répertoire virtuel concerné. Si vous exécutez votre service web avec le serveur web de *Visual Studio 2005*, là aussi il faut désactiver l'authentification intégrée avec le menu *Propriété du projet* ▶ *Start Options* ▶ Décochez *NTLM Authentication* :

```
soapsuds.exe /url:http://localhost:80/LocalizationCorp/
Localizer.asmx?wsdl/oa:ProxyLocalizationCorp.dll
```

Le programme suivant, qui doit être compilé en référençant les assemblages `ProxyLocalizationCorp.dll` et `System.Runtime.Remoting.dll` utilise le service web à partir de .NET Remoting. Un client .NET Remoting n'a pas besoin que le service web supporte le formatage RPC. Vous pouvez donc maintenant enlever l'attribut `SoapRpcService` en revenant à l'Exemple 24-1. Notez que les classes `LocalizerSoap` et `Town` sont dans l'espace de noms `InteropNS` :

*Exemple 24-5 :*

```
using System ;
using System.Runtime.Remoting ;
using System.Runtime.Remoting.Channels ;
using System.Runtime.Remoting.Channels.Http ;
class Program {
   static void Main() {
      HttpChannel canalHttp = new HttpChannel(0) ;
      ChannelServices.RegisterChannel( canalHttp, false ) ;

      MarshalByRefObject obj = (MarshalByRefObject)
         RemotingServices.Connect(
         typeof(InteropNS.LocalizerSoap),
         "http://localhost:80/LocalizationCorp/Localizer.asmx") ;
      InteropNS.LocalizerSoap proxy = obj as InteropNS.LocalizerSoap;

      InteropNS.Town town = proxy.GetTownFromLatLon(43.42, 7.15) ;
      Console.WriteLine("Ville:" + town.Name + " Pays:" + town.Country) ;
   }
}
```

Après cette incursion dans le monde de la pratique il est temps d'apporter un peu plus de théorie quant aux langages SOAP et WSDL.

# Encoder les messages au format SOAP

Avant d'aborder cette section, rappelons que les trois caractéristiques centrales de SOAP vues dans l'introduction sont :

- La possibilité d'extension du langage SOAP pour implémenter des protocoles de communications (d'encryption, d'authentification, transactionnel etc).
- Les messages codés en langage SOAP peuvent être acheminés par tous types de protocoles réseaux tels que HTTP, TCP, UDP ou SMTP.
- SOAP permet tous types de modèle d'échange de messages.

## Introduction à SOAP

La version 1.0 de SOAP était destinée à être utilisée par des technologies d'objets distribués. La version 1.1 de SOAP est utilisée depuis plusieurs années principalement par les services web. Le mot *Object* de l'acronyme SOAP n'est donc plus significatif. Aussi depuis la version 1.2 de SOAP, le terme *SOAP* n'est plus considéré comme un acronyme.

WSE 3.0 travaille maintenant par défaut avec la version 1.2. Dans la suite nous nous baserons donc sur la version 1.2. Vous pouvez obtenir une liste exhaustive des évolutions de SOAP 1.1 vers SOAP 1.2 dans cet article `http://www.idealliance.org/papers/xmle02/dx_xmle02/papers/02-02-02/02-02-02.html`. La spécification complète de SOAP 1.2 est divisée en trois documents accessibles aux URL `http://www.w3.org/TR/soap12-part0/`, `http://www.w3.org/TR/soap12-part1/` et `http://www.w3.org/TR/soap12-part2/`.

Un message SOAP contient un élément racine `<Envelope>`. Cet élément contient optionnellement un élément `<Header>` au début et obligatoirement un élément `<Body>` en dernière position. Donc un message SOAP ressemble à ceci :

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Header> <!-- optionel -->
      <!-- Contient les données relatives au transport du message. -->
   </soap:Header>
   <soap:Body> <!-- obligatoire -->
      <!-- Contient les données métiers. -->
   </soap:Body>
</soap:Envelope>
```

L'élément `<body>` contient les données utiles du message codées en XML. Par exemple, voici à quoi ressemble un message SOAP requête envoyé par notre client au service `localizer` :

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <GetTownFromLatLon xmlns="http://tempuri.org/">
         <lat>43.42</lat>
         <lon>7.15</lon>
      </GetTownFromLatLon>
   </soap:Body>
</soap:Envelope>
```

L'élément <Header> contient des données codées en XML, fabriquées et consommées par des extensions de SOAP telles que les implémentations des spécifications WS-*. Le langage SOAP ne définit pas de sous éléments de l'élément <Header>. Seules les spécifications WS-* définissent des sous éléments de l'élément <Header>. Chaque spécification impliquée dans l'acheminement d'un message ajoute ses sous éléments. C'est grâce à ce mécanisme que l'on obtient à la fois l'extensibilité du langage SOAP et la possibilité de composer les protocoles.

Voici un message SOAP qui contient un jeton de sécurité sous la forme utilisateur/mot de passe ajouté par une implémentation de la spécification WS-Security. Il contient aussi des informations sur la source du message, qui ont été ajoutées par une implémentation de la spécification WS-Addressing :

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext/">
   <soap:Header>
      <!-- Les données relatives à WS-Addressing. -->
      <wsa:From>
         <wsa:Address>http://smacchia.com/client</wsa:Address>
      </wsa:From>
      ...
      <!-- Les données relatives à WS-Security. -->
      <wsse:Security soap:mustUnderstand="1">
         <wsse:UserName>psmacchia</wsse:UserName>
         <wsse:Password wsse:Type="wsse:PasswordDigest">
            yH/*kiGGdsdujg5%16?LHRVhbg...
         </wsse:Password>
         ...
      </wsse:Security>
      ...
   </soap:Header>
   <soap:Body>
      ...
   </soap:Body>
</soap:Envelope>
```

## *Traitement de l'entête d'un message SOAP*

Bien que le langage SOAP ne définisse pas de sous éléments de l'élément <header>, il définit un ensemble de règles pour traiter ces éléments de l'entête. Les spécifications WS-* doivent ainsi suivre ces règles. Pour expliquer ces règles, il faut d'abord aller plus loin dans la terminologie SOAP.

Un *nœud SOAP* est un agent logiciel qui émet ou (non exclusif) reçoit des messages SOAP. Pour un message SOAP donné, on distingue le nœud d'origine (qualifié aussi de nœud expéditeur, c'est le client du service) et le nœud final (qualifié aussi de nœud destinataire, c'est le service). Comme illustré par la figure suivante, entre ces deux nœuds, un même message peut être reçu puis renvoyé par plusieurs nœuds intermédiaires. Les nœuds intermédiaires ainsi que le nœud final sont qualifiés de nœuds du chemin. Naturellement toutes les topologies de nœuds SOAP ne sont pas nécessairement aussi sophistiquées et n'utilisent pas forcément plusieurs protocoles réseaux entre les nœuds :

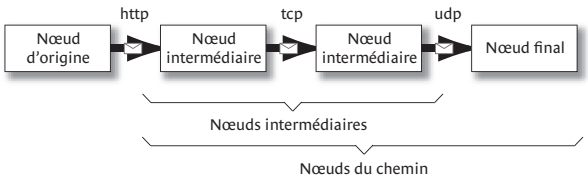


*Figure 24-2 : Une topologie sophistiquée de nœuds SOAP*

Seul le nœud final est habilité à traiter les données de l'élément <body>. En revanche un élément de l'entête peut être traité par n'importe quel nœud du chemin. Ces traitements sont définis par les spécifications WS-*. Cependant SOAP fournit un mécanisme permettant de déterminer quel nœud du chemin doit traiter quel élément d'entête au moyen des trois attributs XML `role`, `mustUnderstand` et `relay`.

Chaque nœud du chemin joue un ou plusieurs *rôles SOAP*. Un élément d'entête informe un nœud SOAP si il peut le traiter au moyen de l'attribut `role`. Si l'URI précisée par cet attribut définit un des rôles joué par le nœud SOAP alors ce dernier peut potentiellement le traiter. En fait, le nœud SOAP n'est obligé de le traiter que si l'élément de l'entête contient aussi l'attribut `mustUnderstand="1"` (ou `mustUnderstand='true'`). Si un nœud intermédiaire traite un ou plusieurs éléments de l'entête, il devra les enlever du message avant de le renvoyer vers le prochain noeud. Un élément de l'entête qui correspond au rôle d'un nœud intermédiaire mais qui ne doit pas être obligatoirement traité peut être soit enlevé du message, soit laissé si il contient l'attribut `relay="1"` (ou `relay='true'`).

SOAP 1.2 ne définit que trois rôles SOAP : le rôle `Next` est joué par tous les nœuds intermédiaires tandis que le rôle `UltimateReceiver` n'est joué que par le nœud final. Le rôle `None` n'est joué par aucun nœud. Les données d'un élément d'entête qui a ce rôle sont donc purement informatives. Ces trois rôles sont représentés respectivement par les URIs `http://www.w3.org/2003/05/soap-envelope/role/next`, `http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver` et `http://www.w3.org/2003/05/soap-envelope/role/none`. Les spécifications WS-* sont libres d'étendre la langage SOAP en définissant de nouveaux rôles.

Il est intéressant de noter que seuls les éléments de l'entête qui doivent être obligatoirement traités (i.e ceux avec un attribut `mustUnderstand` positif) peuvent entraîner une condition de rupture. Autrement dit, si un nœud du chemin rencontre un problème lors du traitement d'un tel élément de l'entête, il est obligé de produire une erreur qui empêchera le traitement des données du corps du message par le nœud final. Cette erreur peut éventuellement être acheminée jusqu'au nœud d'origine par l'intermédiaire d'un nouveau message SOAP d'erreur.

## *Encodage du corps d'un message SOAP*

Les données contenues dans le corps d'un message SOAP (i.e dans l'élément <body>) se présentent en général sous la forme d'un document XML dont le schéma est précisé dans le contrat du service concerné. Cette façon de procéder est connue sous le nom de *formatage « document » / encodage « littéral »* puisque les données sont formatées dans un *document* XML qui satisfait *littéralement* à un schéma XSD.

L'historique « orientée objet » du langage SOAP que nous avons mentionnée dans les pages précédentes fait qu'il existe une alternative pour présenter les données du corps d'un message

SOAP. Cette autre façon de faire est connue sous le nom de *formatage « RPC » / encodage « encoded »*. Cette technique est adaptée aux échanges de messages requête/réponse synchrone modélisant un appel de méthode (d'où *RPC*). L'encodage *encoded* fait partie de la spécification de SOAP et décrit comment sérialiser en XML des arguments entrant et sortant d'une méthode en se basant sur un ensemble de règles. Ces règles spécifient la façon dont les objets, les tableaux, les structures et les graphes d'objets sont sérialisés.

SOAP est maintenant principalement utilisé pour coder des messages échangés entre services. Comme nous l'avons mentionné le modèle d'échange de messages RPC style requête/réponse synchrone n'est plus qu'un modèle d'échange parmi d'autres. Il n'y a donc plus de raison de le privilégier. Aussi les différentes implémentations des services web telles que ASP.NET/WSE utilisent par défaut le modèle d'encodage *document/literal* plutôt que le modèle *RPC/encoded*.

### *Messages d'erreur SOAP*

Nous avons vu que le traitement d'un élément d'entête par un nœud SOAP du chemin peut engendrer une erreur qui peut être retournée au nœud d'origine par l'intermédiaire d'un nouveau message SOAP. Ceci est possible car le langage SOAP permet de définir des messages SOAP d'erreur. Un message SOAP d'erreur est un message SOAP dont le corps ne contient qu'un élément `<Fault>`. La spécification du langage SOAP explique comment encoder au sein de cet élément la raison de l'erreur, le rôle SOAP qui a provoqué l'erreur, le nœud qui a provoqué l'erreur etc. Elle spécifie aussi certains codes d'erreur tel que le code `MustUnderstand` qui signifie qu'un élément de l'entête avec un attribut `mustUnderstand` positif n'a pas pu être traité.

### *Lien entre SOAP et les protocoles de transport sous-jacents*

Pour chaque protocole réseau pouvant être utilisé pour transporter un message SOAP il existe une spécification expliquant en détail comment le message est acheminé. En général cela ne pose pas de problèmes puisque la plupart des protocoles utilisés ont une notion de ce qu'est au sens large un message.

La spécification SOAP 1.2 ne décrit que le lien avec le protocole HTTP. Elle établit notamment une correspondance naturelle avec le modèle d'échange de messages style requête/réponse synchrone de HTTP.

## *Définir des contrats avec le langage WSDL*

Nous avons introduit *WSDL* (*Web Service Description Language* prononcé *Wizdil*) comme un langage XML extensible servant à rédiger les contrats des services. Nous allons maintenant nous intéresser en détail à ce langage.

### *Domaine de couverture de WSDL*

Le langage WSDL est constitué d'un élément racine `<definitions>` et de sept éléments XML que l'on peut classer en deux catégories :

- Les éléments `<types>`, `<message>`, `<operation>` et `<portType>` utilisés pour décrire le contrat que présente un service web.

- Les éléments `<binding>`, `<port>` et `<service>` utilisés pour préciser les liens entre un contrat et une implémentation. En plus de lier les web méthodes aux opérations du service, ces liens décrivent d'autres aspects de l'implémentation tels que le protocole réseau ou l'encodage des données dans les messages SOAP.

Ainsi, certaines informations décrites par la seconde catégorie d'éléments ne sont pas utiles aux clients des services web. Elles ne font donc pas partie du contrat.

La spécification WS-Policy décrit comment étendre WSDL pour rédiger des stratégies. Les stratégies sont des clauses du contrat permettant d'augmenter sa précision. Par exemple, le langage WSDL ne spécifie pas comment expliciter le fait que les messages envoyés à un service doivent être encryptés. Il faut pour cela ajouter des stratégies de sécurité dans le contrat. Ainsi, seule une partie du contrat est effectivement rédigée en WSDL :

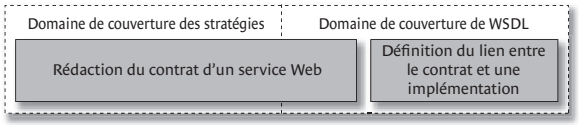


*Figure 24-3 : WSDL et contrat*

## *Analyse d'un document WSDL*

Voici le document WSDL de notre service `localizer` :

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions
   xmlns:s1="http://localizationcorp.com/documents/data/"
   xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:s="http://www.w3.org/2001/XMLSchema"
   xmlns:s0="http://tempuri.org/"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
   xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
   targetNamespace="http://tempuri.org/"
   xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <s:schema elementFormDefault="qualified"
              targetNamespace="http://tempuri.org/">
      <s:import namespace="http://localizationcorp.com/documents/data/" />
      <s:element name="GetTownFromLatLon">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="lat"
                       type="s:double" />
            <s:element minOccurs="1" maxOccurs="1" name="lon"
                       type="s:double" />
          </s:sequence>
```

```
        </s:complexType>
      </s:element>
      <s:element name="GetTownFromLatLonResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1"
             ref="s1:GetTownFromLatLonResult" />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
    <s:schema elementFormDefault="qualified"
          targetNamespace="http://localizationcorp.com/documents/data/">
      <s:element name="GetTownFromLatLonResult" nillable="true"
                 type="s1:Town" />
      <s:complexType name="Town">
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1" name="Name"
                     type="s:string" />
          <s:element minOccurs="0" maxOccurs="1" name="Country"
                     type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:schema>
  </types>
  <message name="GetTownFromLatLonSoapIn">
    <part name="parameters" element="s0:GetTownFromLatLon" />
  </message>
  <message name="GetTownFromLatLonSoapOut">
    <part name="parameters" element="s0:GetTownFromLatLonResponse" />
  </message>
  <portType name="LocalizerSoap">
    <operation name="GetTownFromLatLon">
      <input message="s0:GetTownFromLatLonSoapIn" />
      <output message="s0:GetTownFromLatLonSoapOut" />
    </operation>
  </portType>
  <binding name="LocalizerSoap" type="s0:LocalizerSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
     style="document" />
    <operation name="GetTownFromLatLon">
      <soap:operation soapAction="http://tempuri.org/GetTownFromLatLon"
       style="document" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
```

```
    </operation>
  </binding>
  <service name="Localizer">
    <port name="LocalizerSoap" binding="s0:LocalizerSoap">
      <soap:address
       location="http://localhost/localizationcorp/localizer.asmx" />
    </port>
  </service>
</definitions>
```

### *L'élément <definitions>*

L'élément `<definitions>` est l'élément racine de tous document WSDL. Cet élément inclut les espaces de noms standard utilisés (SOAP, XSD etc) et éventuellement des espaces de noms propriétaires.

### *L'élément <Types>*

On voit que l'élément `<types>` contient les schémas XSD des données qui sont contenues dans la partie `<Body>` des messages SOAP échangés. Notez que l'élément `<types>` est obligatoire mais qu'il peut être vide si vous importez les schémas XSD dans un élément racine `<import>` :

```
<?xml version="1.0" encoding="utf-8" ?>
<import namespace="http://localizationcorp.com/documents/data/"
        location="http://localizationcorp.com/documents/data/schema.xsd" >
<definitions>
  <types />
  ...
</definitions>
```

### *L'élément <message>*

Chaque élément `<message>` définit un message SOAP entrant, sortant ou d'erreur. Chaque élément `<part>` d'un élément `<message>` référence un schéma de données. Puisqu'un message SOAP peut contenir plusieurs parties, un élément `<message>` peut contenir plusieurs éléments `<part>`.

### *L'élément <operation>*

Chaque élément `<operation>` définit les messages SOAP supportés par chaque opération du service web. Dans notre service web simple `localizer`, la seule opération est constituée d'un échange de message synchrone style RPC aussi elle présente un message entrant et un message sortant. Une opération supportant un autre modèle d'échange de message pourrait ne définir qu'un message d'entrée ou de sortie. Une opération a aussi la possibilité de présenter un message d'erreur en contenant un élément `<fault>`.

Le nom d'une opération définie dans l'élément `soap:operation` est préfixé par défaut par l'espace de nom `http://tempuri.org/`. L'Exemple 24-2 montre comment personnaliser cet espace de nom.

### *L'élément <portType>*

Un élément <portType> définit l'ensemble des opérations présentées par un service. À l'instar de note exemple les opérations ont la possibilité d'être définies directement dans l'élément <portType>. Elles peuvent être aussi définies hors de cet élément puis référencées comme ceci :

```
...
   <operation name="GetTownFromLatLon">
      <input message="s0:GetTownFromLatLonSoapIn" />
      <output message="s0:GetTownFromLatLonSoapOut" />
  </operation>
  <portType name="LocalizerSoap">
    <operation name="GetTownFromLatLon" />
  </portType>
...
```

Si l'on raisonne en termes de langage objet les types de l'élément <types> s'apparentent aux structures/classes de données, les éléments <message> s'apparentent aux signatures des méthodes, les éléments <operation> s'apparentent aux méthodes et les éléments <portType> s'apparentent aux interfaces. Intéressons nous maintenant aux éléments WSDL permettant d'établir un lien entre un contrat et une implémentation.

### *L'élément <binding>*

L'élément <binding> permet de préciser des informations sur l'implémentation d'un service. Ces informations sont essentiellement :

- Les protocoles d'encodage des messages SOAP avec les éléments <soap:body>.
- Les noms des web méthodes à utiliser pour chaque opération avec les éléments <soap:operation>.
- Le protocole réseaux sous-jacent avec l'élément <soap:binding>.

Comprenez bien que par exemple les éléments <soap:operation> et <operation> sont différents puisqu'ils ne font pas parties du même espace de noms. En outre, bien que certaines de ces informations soient utiles aux clients pour consommer le service, elles ne contiennent pas d'information relative à la sémantique du service.

### *L'élément <port>*

L'élément <port> permet de faire le lien entre un élément <binding> et une implémentation d'un service au moyen d'un élément <soap:address>. Il est intéressant que cette information ne fasse pas partie de l'élément <binding> car cela donne un degré de liberté supplémentaire pour rediriger les messages SOAP entrant vers une autre implémentation.

### *L'élément <service>*

L'élément <service> est une collection d'élément <port>.

## *Squelette d'un document WSDL*

En résumé, voici le squelette d'un document WSDL :

```
<definitions>
   <types />
   <message />
   <operation>
      <message />
   </operation>
   <portType>
      <operation />
   </portType>
   <binding>
      <operation />
   </binding>
   <service>
      <port>
         <binding />
      </port>
   </service>
</definitions>
```

# *Introduction à WSE et aux spécifications WS-**

## *Introduction à WSE*

Les facilités présentées par *Visual Studio* et ASP.NET pour le développement d'un service web ne permettent pas d'implémenter des possibilités basiques telles que l'encryptions des messages ou l'utilisation d'un protocole de transport autre que HTTP. D'un autre coté seules certaines spécifications WS-* sont finalisées. Depuis 2003, pour permettrent aux développeurs .NET de construire des services web exploitants ces spécifications finalisées *Microsoft* fournit gratuitement la plateforme de développement nommé *WSE* (*Web Service Enhancement* prononcez *Wizi*).

Concrètement WSE est principalement constitué d'un éditeur de configuration intégré à *Visual Studio* et de l'assemblage `Microsoft.Web.ServiceXX.dll` (`XX` désigne la version de WSE). L'éditeur de configuration WSE permet de fixer simplement les paramètres d'un projet qui exploite WSE. Cet éditeur travaille en parsant et en mettrant à jour le code du fichier de configuration de l'application (`web.config` dans le cas d'un service, `app.config` dans le cas d'un client d'un service). Bien entendu vous pouvez décider de ne pas utiliser cet éditeur et préférer modifier vos fichiers à la main.

L'assemblage `Microsoft.Web.ServiceXX.dll` doit être référencé par chaque assemblage qui exploite WSE et distribué avec chaque application basée sur WSE. Il contient du code qui modifit les messages SOAP envoyés ou reçus selon les spécifications WS-* supportées. Cet assemblage peut être utilisé du côté du client d'un service et du côté du service.

La plateforme .NET 2.0 n'est compatible qu'avec les versions 3.0+ de WSE. Il faut savoir que les versions majeures de WSE (1.0, 2.0, 3.0 etc) ne sont pas compatibles entre elles. En outre, les livraisons de WSE ne sont pas calquées sur les livraisons du *framework*. Ces deux libertés prisent par *Microsoft* confèrent une certaine souplesse à l'évolution de WSE.

Vous pouvez être réticent à investir dans une technologie qui n'aura pas de compatibilité ascendante. Cependant sachez que WSE constitue l'alternative la plus efficace pour mettre des services

en production avec .NET dés aujourd'hui. Il est clair que vous bénéficierez de vos acquis lorsque vous devrez aborder les futures plateformes de développement plus évoluées telles que *Windows Communication Foundation*.

## *Les spécifications supportées par WSE 3.0*

WSE 3.0 supporte principalement les spécifications suivantes :

- *WS-Policy* : pour le développement de stratégies.
- *WS-Security* pour l'authentification, et l'encryptions des messages.
- *WS-SecureConversation* pour établir des sessions d'échange de messages sécurisés (dans le même esprit que le protocole SSL).
- *WS-SecurityPolicy* pour la description des stratégies de sécurité.
- *WS-Trust* pour l'obtention de jeton de sécurité.
- *WS-Addressing* pour supporter des modèles d'échange de messages autres que le style RPC.
- MTOM (*Message Transmission Optimization Mechanism*) pour l'envoi de données binaires volumineuses.

En plus de HTTP, WSE 3.0 permet d'exploiter le protocole RPC ou un mode de transport optimisé lorsqu'un service et son client sont dans le même processus. Dans ces deux cas il n'est bien évidemment pas utile d'exploiter la technologie ASP.NET ni un serveur web tel que IIS.

## *L'installation de WSE*

WSE est téléchargeable gratuitement sur le site de *Microsoft*. En plus de l'éditeur de configuration WSE et de l'assemblage `Microsoft.Web.ServiceXX.dll` vous trouverez dans le répertoire d'installation des outils, de la documentation et des exemples de projets exploitant WSE.

La technologie de certification X509 (décrite dans la section page 230) est largement exploitée dans le domaine de la sécurité des services développés avec WSE. L'outil *X509 Certificate Tool* fourni avec WSE permet de faciliter la manipulation des certificats X509. WSE est livré avec deux certificats X509 que vous pouvez utiliser pour tester les exemples fournis ainsi que vos propres prototypes.

WSE est aussi livré avec l'outil *Policy Wizard* qui permet de générer plus simplement vos stratégies.

Une fois que WSE est installé sur votre machine de développement vous pouvez l'activer très simplement sur vos projets *Visual Studio* (ASP.NET ou non). Il vous suffit de cliquer droit sur votre projet dans l'explorateur de solution et de sélectionner *WSE Settings 3.0...*. L'éditeur de WSE s'affiche alors et vous n'avez plus qu'à sélectionner *Enable this project for Web Services Enhancements*. Dans le cas d'un projet ASP.NET il faut aussi sélectionner *Enable Microsoft Web Services Enhancements Soap Extensions*.

## *Comment WSE est implémenté grâce aux extensions SOAP ?*

Dans le cas d'un projet ASP.NET, WSE exploite les *extensions SOAP*. Une extension SOAP est une classe capable d'avoir accès aux messages sortants avant qu'ils soient envoyés et aux messages entrants avant qu'ils soient exploités par le service. Pour pouvoir être reconnue comme une extension SOAP, une classe doit dériver de la classe abstraite `System.Web.Services.SoapExtension`.

La description détaillée des extensions SOAP dépasse le cadre de cet ouvrage. Néanmoins vous pouvez vous référer à la documentation des MSDN concernant la classe `SoapExtension`. En effet, celle-ci contient un exemple d'extension SOAP particulièrement instructif qui permet de sauver dans un fichier de log les messages SOAP entrants et sortants. Pour faire fonctionner cet exemple vous devez prendre trois précautions :

- Faire en sorte que l'utilisateur *Windows* qui exécute le processus d'ASP.NET (par défaut ASPNET) ait tous les droits d'accès sur le répertoire dans lequel l'application web est stockée afin de pouvoir créer et modifier les fichiers de log.
- Utiliser un client développé avec une classe proxy. En effet, si vous utilisez un navigateur pour tester votre service vous n'utiliserez pas la méthode `HttpSoap` et dans ce cas les extensions SOAP ne sont pas utilisées.
- Ajoutez ceci dans votre fichier `web.config` afin d'activer votre extension SOAP.

```
<configuration>
   <system.web>
      <webServices>
         <soapExtensionTypes>
            <add type="NomDeLaClasseAvecEspaceDeNoms,
                 NomDeLAssemblageContenantLaClasse",
                 priority="1" group="0" />
          </soapExtensionTypes>
      ...
```

- Notez que vous pouvez décider de n'appliquer une extension SOAP que sur certaines web méthodes. Dans ce cas il ne faut pas rajouter ceci à votre fichier `web.config` mais il faut marquer chaque méthode concernée avec un attribut d'extension SOAP. Plus d'informations à ce sujet sont disponibles dans la documentation des MSDN concernant la classe `SoapExtensionAttribute`.

Dans le cas d'un projet non ASP.NET l'éditeur de configuration de WSE crée une nouvelle classe proxy dont le nom se termine par `Wse`. Dans notre exemple il y a donc deux classes proxy : `Localizer` et `LocalizerWse`. La classe `Localizer` dérive de `System.Web.Services.Protocols.SoapHttpClientProtocol` tandis que la classe `LocalizerWse` dérive de `Microsoft.Web.Services2.WebServicesClientProtocol`. Bien entendu vous n'obtenez les services de WSE que si vous utilisez la classe proxy `LocalizerWse` pour accéder à votre service.

## *Un test de WSE avec les diagnostics*

Vous pouvez très simplement tester WSE sur un projet (ASP.NET ou non) en activant le log des messages SOAP dans l'onglet *Diagnostics* de l'éditeur de configuration WSE Il est instructif que vous regardiez les modifications qui sont alors apportées au fichier `web.config` (ou `app.config` dans le cas d'un projet non ASP.NET). Ici aussi, le test ne s'exécute correctement que si vous prenez les précautions de la section précédente (droits d'accès de l'utilisateur *Windows* ASPNET + client avec une classe proxy).

# *Les spécifications WS-* non encore supportées par WSE*

## *WS-PolicyAttachment et WS-MetadataExchange*

Il n'est pas encore possible qu'un client obtienne automatiquement les expressions *WS-Policy* qui s'appliquent sur les opérations d'un service développé avec WSE. Le responsable du service peut toujours communiquer le fichier contenant les expressions d'une manière ou d'une autre aux clients mais ce type de solution n'est pas satisfaisant car non standardisé. Aussi, il existe la spécification *WS-PolicyAttachment* qui décrit comment assigner des expressions aux opérations d'un service directement dans le contrat WSDL. Il existe aussi la spécification *WS-MetadataExchange* qui décrit comment un client peut récupérer les expressions référencées par WS-PolicyAttachment.

## *WS-ReliableMessage*

La spécification *WS-ReliableMessage* permet de pallier l'absence de garanties de certains protocoles réseaux utilisés pour acheminer les messages SOAP. Par exemple, le protocole réseau UDP ne fournit ni mécanisme d'accusé réception pour informer l'envoyeur d'un message qu'il a bien été reçu par son destinataire, ni mécanisme permettant de garantir l'ordre de réception d'une séquence de messages par un destinataire, ni mécanisme pour permettre au destinataire d'éliminer des messages dupliqués. Bien entendu la spécification WS-ReliableMessage se place au niveau le plus bas dans l'empilement des spécifications WS-* et sera gérée implicitement par les environnements d'exécutions.

## *UDDI et WS-Discovery*

Les spécifications *UDDI* (*Universal Description Discovery and Integration*) et *WS-Discovery* permettent de découvrir des services web. Tandis que UDDI définit un système centralisé basé sur des inscriptions et des recherches dans des annuaires, WS-Discovery exploite des algorithmes types multi diffusion où l'on peut se passer d'annuaires centralisés.

Les annuaires UDDI peuvent être publiques, inter entreprise ou intra entreprise. Chaque entrée dans un annuaire représente un fournisseur de service. Une partie de cette entrée est consacrée à la présentation du fournisseur de service (domaines d'activité, contacts, tarifs etc). Une deuxième partie est dédiée à la description abstraite des services web présentés (éléments `<types>`, `<import>`, `<message>`, `<portType>` et `<binding>` du contrat WSDL). Les documents représentant les descriptions abstraites des services sont nommés *Types Models* (*tModel*). La recherche d'un service se fait essentiellement en demandant à l'annuaire quels sont les services qui supportent un certains tModel. Enfin chaque entrée de l'annuaire contient une troisième partie qui stocke les liens vers les implémentations des services (éléments `<import>` et `<service>` du contrat WSDL).

WS-Discovery définit deux messages *'hello'* et *'bye'* permettant à un service de se déclarer ou de se retirer lorsqu'il se connecte ou se déconnecte à un réseau. Ces messages sont relayés par multi diffusion pour atteindre les clients potentiels du service. Ces derniers ont aussi à leur disposition deux messages *'probe'* et *'resolve'* pour lancer une recherche d'un service sur le réseau. Pour limiter la consommation excessive de bande passante des protocoles multi diffusion, WS-Discovery permet d'implémenter des services spéciaux dit *proxy de découverte* pour s'appuyer sur une certaine centralisation de la liste des services disponibles sur le réseau.

## *WS-Federation*

La spécification *WS-Federation* étend la spécification WS-Trust pour affiner le modèle d'authentification d'identités entre différents domaines de confiance. WS-Federation permet à un même utilisateur de naviguer entre différents domaines fédérés, sans avoir à s'authentifier explicitement à chaque fois. WS-Federation définit notamment les messages SOAP que s'échangent les domaines pour rendre une telle navigation possible. D'un point de vue pratique, cette spécification vise à pallier plusieurs problèmes couramment rencontrés dans les organisations et les entreprises tels que :

- La difficulté pour obtenir les autorisations qu'un agent d'une organisation avec laquelle on collabore a dans le cadre d'une opération donnée.
- La difficulté pour authentifier une même personne au travers de ses différentes adresses e-mail (professionnelles, personnelles etc).
- La difficulté pour définir le degré de sensibilité des données privées d'une identité en fonction des contextes des opérations qu'elle effectue.

## *WS-Coordination*

La spécification *WS-Coordination* permet à plusieurs services web de coordonner leurs opérations dans le cadre d'un travail qu'ils ont à réaliser en commun. Chaque travail est identifié par un identificateur unique nommé contexte de coordination. Cet identificateur se retrouve dans des entêtes SOAP de chaque message SOAP échangé dans le cadre de ce travail. WS-Coordination définit ces entêtes ainsi que leurs significations (requête pour demander d'effectuer une opération, refus ou incapacité de faire une opération, terminaison d'une opération, time out d'une opération etc). Remarquez que certains travaux en commun tels qu'une transaction 2PC nécessitent la présence d'un service qui joue le rôle particulier de coordinateur. D'autres travaux tels que l'élection d'un service parmi plusieurs services n'ont pas besoin de coordinateurs. WS-Coordination gère ces deux types de travaux et l'ensemble des travaux possibles est ouvert. Cette extensibilité se traduit par d'autres spécifications WS-* telles que WS-AtomicTransaction qui se basent sur les possibilités de WS-Coordination pour définir des travaux précis.

## *WS-AtomicTransaction et WS-BusinessActivity*

La spécification *WS-AtomicTransaction* se sert des possibilités de WS-Coordination pour permettre d'effectuer des transactions ACID volatiles ou durables, distribuées sur plusieurs services. Pour cela WS-AtomicTransaction exploite les algorithmes styles 2PC. Cette spécification sera implémentée par le DTC (Distributed Transaction Coordinator) de *Windows Vista*. Elle supporte notamment les notions de transactions volatiles et durables.

La spécification *WS-BusinessActivity* se sert des possibilités de WS-Coordination pour permettre d'effectuer des transactions longues distribuées sur plusieurs services. Les transactions longues viennent pallier l'impossibilité de verrouiller certaines ressources pendant la durée d'une transaction ACID. Aussi, une telle ressource est mise à jour dés le début de la transaction longue. S'il s'avère que la transaction échoue, le coordinateur en informe chaque participant. Libre à eux alors d'effectuer une action compensatoire pour remettre leurs ressources dans leurs états initiales. Une conséquence de cette façon de procéder est que l'on n'a pas de notion d'isolation entres plusieurs transactions longues. Ainsi, il faut être conscient qu'un participant ne peut pas forcément remettre une ressource dans son état initial puisque cette dernière a pu être modifiée entre temps dans le cadre d'une autre transaction.

## *WS-Enumeration*

La spécification *WS-Enumeration* permet de demander une grosse quantité d'information à un service. Une telle demande se traduit par l'établissement d'une session WS-Enumeration qui se charge de gérer la séquence des messages envoyés du service vers le demandeur. Le suivit de l'avancement du curseur sur les informations à transmettre peut se faire aussi bien du côté demandeur que du côté service. Ce suivi peut même éventuellement être basculé durant une même session, par exemple du serveur vers le demandeur pour libérer le serveur de la gestion de l'avancement lorsqu'il détecte que le temps de latence entre les messages devient prohibitif.

## *WS-Eventing*

La spécification *WS-Enumeration* permet d'implémenter d'une manière standard le modèle d'échange de message abonnement/notification décrit en page 991.

## *WS-Management*

La spécification *WS-Management* a pour but d'uniformiser la supervision des systèmes d'information. Le terme système d'information doit ici être pris au sens large et englobe à la fois le *software* (tels qu'un serveur IIS ou un système d'exploitation) et le *hardware* (tels qu'un *SmartPhone* ou un *Pocket PC*). WS-Management définit des messages SOAP modélisant les opérations qui sont exigées par toutes les solutions de supervision. Parmi ces opérations, citons la découverte de la présence de ressources de supervision, le paramétrage de ces ressources et l'obtention de renseignements sur l'état courant du système. WS-Management est supportée par les acteurs majeurs du marché tels que *Microsoft*, *Intel*, *AMD*, *Dell* ou Sun.

# *Introduction à WCF (Windows Communication Framework)*

*WCF* (*Windows Communication Framework* anciennement nommé *Indigo*) est un *framework* qui sera une partie intégrante de *Windows Vista*. Il sera aussi disponible pour *Windows XP*. Ce *framework* vise à unifier les différentes technologies de communication proposées par *Microsoft* jusqu'ici à savoir : ASMX, .NET Remoting, COM+/Enterprise Services, System.Messaging, MSMQ et WSE. La compatibilité avec ces technologies permettra à chaque entreprise de migrer à son rythme vers WCF.

WCF permet d'établir des communications entre des *endpoints* WCF (que l'on pourrait traduire par point de terminaison). Chaque endpoint est caractérisé par trois composantes :

- Son adresse qui permet de le localiser.
- Ses informations de liaison (*binding*) qui définissent le type de protocole avec lequel il sait communiquer.
- Ses contrats au sens où ce que l'on a pu voir dans le présent chapitre (structure et sémantique d'utilisation).

L'architecture de WCF fait en sorte de bien séparer chacune de ces composantes. En tant que développeur, vous serez surtout concerné par la notion de contrat puisque les notions d'adresse et d'information de liaison ne sont significatives que durant le déploiement et la maintenance.

WCF supportera la plupart des spécifications WS-* non encore implémentées par WSE que nous venons de présenter. La conception de WCF fait en sorte de masquer la complexité de ces spécifications aux développeurs. Aussi, vous aurez accès à toutes ces fonctionnalités au travers de simples paramètres de fichiers de configuration et il est probable que vous n'entendrez parler des spécifications WS-* sous jacente que très rarement.

# A

# *Les mots-clés du langage C♯ 2.0*

Voici la liste des mots-clés :

- Les mots-clés en gras n'existent pas dans le langage C++.
- Les mots-clés en italique existent dans le langage C++ mais ont une utilisation ou une signification différente.
- Le terme *C♯2* est précisé pour un mot-clé introduit avec cette version du langage. Très peu de mots-clés ont été ajoutés en comparaison du nombre de nouvelles possibilités.

Remarquez que plusieurs mots-clés ont deux ou trois significations (`using`, `delegate`, `new`, `fixed` etc).

Un mot-clé ne peut pas être utilisé comme identificateur. En revanche un mot-clé préfixé du caractère @ est un identificateur, mais il vaut mieux éviter cette pratique.

| Mot-clé | Référence |
|---|---|
| **`abstract`** | La solution ! les classes abstraites et les méthodes abstraites, page 453 |
| **`as`** | L'opérateur as, page 465 |
| **`base`** | L'héritage d'implémentation, page 445 |
| `bool` | Le type booléen, page 356 |
| `break` | Les instructions break et continue, page 332. L'instruction switch, page 329 |
| **`byte`** | Les types concernant la représentation des nombres entiers, page 354 |

| | |
|---|---|
| `case` | L'instruction switch, page 329 |
| `catch` | Principe de la gestion des exceptions, page 509 |
| *`char`* | Le type représentant un caractère, page 357 |
| **`checked`** | Gestion des dépassements de capacité, page 359 |
| `class` | Définition d'une classe, page 396. La contrainte type valeur/type référence, page 477 |
| `const` | Champs constants, page 398 |
| `continue` | Les instructions break et continue, page 332 |
| **`decimal`** | Les types concernant la représentation des nombres réels, page 355 |
| `default` | L'instruction switch, page 329. L'opérateur default, page 484 |
| **`delegate`** | Les délégations et les délégués, page 378. Introduction aux méthodes anonymes de C♯2, page 524 |
| `do` | Les boucles de type while while (mot-clé), et do/while, page 332 |
| `double` | Les types concernant la représentation des nombres réels, page 355 |
| `else` | Utilisation de if/else, page 327 |
| `enum` | Les énumérations, page 366 |
| **`event`** | Les événements, page 411 |
| `explicit` | Opérateurs de conversion de type (de transtypage), page 436 |
| `extern` | L'attribut DllImport, page 265 |
| **`extern alias C♯2`** | Les alias externes, page 322 |
| `false` | Le type booléen, page 356 |
| **`finally`** | Le gestionnaire d'exceptions et la clause finally, page 515 |
| **`fixed`** | Nécessité d'épingler les objets en mémoire, page 505. Les tableaux fixés de taille fixe, page 507 |
| `float` | Les types concernant la représentation des nombres réels, page 355 |
| `for` | Les boucles for, page 332 |
| **`foreach`** | Parcours des éléments d'une collection avec 'foreach' et 'in', page 563 |
| **`get`** | Les propriétés, page 407 |

| | |
|---|---|
| public | Les niveaux de visibilité des membres, page 417 |
| **readonly** | Champs constants, page 398 |
| **ref** | La possibilité de forcer le passage d'argument par référence, page 401 |
| return | Récupération d'information à partir d'une méthode (les paramètres out), page 404 |
| **sbyte** | Les types concernant la représentation des nombres entiers, page 354 |
| **sealed** | Les classes dont on ne peut dériver (sealed), page 448 |
| **set** | Les propriétés, page 407 |
| short | Les types concernant la représentation des nombres entiers, page 354 |
| sizeof | L'opérateur sizeof, page 504 |
| **stackalloc** | Réservation de mémoire sur la pile avec stackalloc, page 508 |
| static | Les membres statiques, page 430 |
| string | Les chaînes de caractères, page 370 |
| struct | Les structures, page 364. La contrainte type valeur/type référence, page 477 |
| switch | L'instruction switch, page 329 |
| this | Les indexeurs, page 409. Le mot-clé this, page 420 |
| throw | Principe de la gestion des exceptions, page 509 |
| true | Le type booléen, page 356 |
| try | Principe de la gestion des exceptions, page 509 |
| **typeof** | Préciser un type, page 240 |
| **uint** | Les types concernant la représentation des nombres entiers, page 354 |
| **ulong** | Les types concernant la représentation des nombres entiers, page 354 |
| **unchecked** | Gestion des dépassements de capacité, page 359 |
| **unsafe** | Déclaration d'une zone de code non vérifiable, page 502 |

| | |
|---|---|
| **ushort** | Les types concernant la représentation des nombres entiers, page 354 |
| using | Utilisation des ressources contenue dans un espace de noms, page 310. L'interface IDisposable et sa méthode Dispose(), page 425. Alias sur les espaces de noms et sur les types, page 320 |
| **value** | Accesseur set, page 408 |
| virtual | La solution ! les méthodes virtuelles et le polymorphisme, page 449 |
| volatile | Les champs volatiles, page 145 |
| void | Casting de pointeurs, page 505 |
| where | Contraintes de dérivation, page 475 |
| while | Les boucles de type while while (mot-clé), et do/while page 332 |
| **yield break C#2** | Le mot-clé yield break, page 546 |
| **yield return C#2** | Un premier exemple avec le mot-clé yield return, page 542 |

# B

# *Nouveautés .NET 2.0*

## *Assemblage*

L'utilisation de l'attribut `AssemblyKeyFile` pour signer un assemblage est maintenant à éviter. Il est préférable d'utiliser les options `/keycontainer` et `/keyfile` du compilateur `csc.exe` ou les nouvelles propriétés du projet de *Visual Studio 2005* (page 30).

Le nouvel attribut `System.Runtime.CompilerServices.InternalsVisibleToAttribute` permet de spécifier des assemblages qui ont accès aux types non publics de l'assemblage sur lequel il s'applique. On parle alors d'assemblages amis (page 27).

L'outil `ildasm.exe` 2.0 présente par défaut la possibilité d'obtenir des statistiques quant à la taille en octets de chaque section d'un assemblage et quant à l'affichage des informations sur les métadonnées. Avec `ildasm.exe` 1.x il fallait utiliser le commutateur `/adv` en ligne de commande afin de les obtenir (page 24).

## *L'internationalisation des applications*

L'outil `resgen.exe` peut maintenant générer le code source de classes C# ou VB.NET qui encapsulent l'accès aux ressources d'une manière fortement typée (page 35).

## *Construction des applications*

La plateforme .NET 2.0 est livrée avec un nouvel outil nommé `msbuild.exe`. Cet outil sert à construire les applications .NET. Cet outil est notamment exploité par *Visual Studio 2005* mais vous pouvez l'utiliser pour lancer vos propres scripts de construction (page 49).

## *Configuration des applications*

La plateforme .NET 2.0 présente une nouvelle gestion fortement typée de vos paramètres de configuration (page 60). *Visual Studio 2005* contient un éditeur de paramètres de configuration qui prend complètement en charge la génération des lignes de code nécessaires pour exploiter cette technique (page 62).

## *Déploiement des applications*

La nouvelle technologie de déploiement d'application nommée *ClickOnce* permet une gestion fine de la sécurité ainsi que des mises à jour et une gestion du déploiement d'une application en plusieurs temps (page 76). En outre, *Visual Studio 2005* présente des facilités très pratiques pour utiliser cette technologie (page 81).

## *CLR*

Un bug majeur du CLR version 1.x rendant possible la modification d'un assemblage signé numériquement a été corrigé en version 2.0 (page 28).

La classe `System.GC` présente les nouvelles méthodes `AddMemoryPressure()` et `RemoveMemoryPressure()` permettant de fournir au GC une indication quant au volume des ressources non gérées détenues (page 123). Une autre méthode `CollectionCount(int generation)` permet d'obtenir le nombre de collectes effectuées pour une génération donnée (page 123).

De nouvelles possibilités ont été ajoutées à l'outil `ngen.exe` pour supporter les assemblages exploitant la réflexion et pour automatiser la mise à jour de la version compilée d'un assemblage lorsqu'une de ses dépendances évolue (page 113).

L'interface `ICLRRuntimeHost` utilisé à partir du code non géré pour héberger le CLR vient remplacer l'interface `ICorRuntimeHost`. Elle permet d'avoir accès à une nouvelle API permettant au CLR de déléguer un certains nombre de responsabilités cruciales telles que le chargement des assemblages, la gestion des threads ou la gestion des allocations mémoire. Cette API n'est pour l'instant qu'utilisée par l'hôte du moteur d'exécution de *SQL Server 2005* (page 100).

Trois nouveaux mécanismes dénommés *région d'exécution contrainte* (CER), *finaliseur critique* et *région critique* (CR) permettent d'accroître la fiabilité des applications telles que *SQL Server 2005* susceptibles de faire face à des pénuries de ressources (page 125, page 129 et page 129).

Un mécanisme dit de portail de mémoire est exploitable pour évaluer avant un traitement si l'on disposera d'assez de mémoire (page 127).

Vous pouvez maintenant terminer rapidement un processus en appelant la méthode statique `FailFast()` de la classe `System.Environment`. Cette méthode ne prend pas de précautions telles que l'exécution des finaliseurs ou des blocs `finally` en attente (page 136).

## *Délégué*

Un délégué peut référencer une méthode générique ou une méthode d'un type générique. On voit alors apparaître la notion de délégué générique (page 491).

Grâce aux nouvelles surcharges de la méthode `Delegate.CreateDelegate(Type, Object, MethodInfo)` il est possible de référencer une méthode statique et son premier argument à partir d'un délégué. Les appels aux délégués n'ont alors pas besoin de ce premier argument et cette pratique s'apparente à un appel de méthode d'instance (page 538).

L'invocation de méthodes au travers de délégués est maintenant beaucoup plus performante.

## *Synchronisation*

Vous pouvez passer simplement des informations à un thread que vous créez grâce à la nouvelle délégation `ParametrizedThreadStart`. En outre, de nouveaux constructeurs de la classe `Thread` vous permettent de fixer la taille maximale en octets de la pile d'un thread (page 140).

La classe `Interlocked` présente de nouvelles méthodes et permet de traiter d'autres types tels que `IntPtr` ou `double` (page 145).

La classe `WaitHandle` présente une nouvelle méthode statique `SignalAndWait()`. En outre, toutes les classes dérivant de `WaitHandle` présentent une nouvelle méthode statique `OpenExisting()` (page 153).

La classe `EventWaitHandle` permet de se passer des deux classes `AutoResetEvent` et `ManualResetEvent` qui en dérivent. De plus, elle permet de nommer un évènement et donc, de le partager entre plusieurs processus (page 155).

La nouvelle classe `Semaphore` permet d'exploiter des sémaphores *Windows* à partir de votre code géré (page 157).

La nouvelle méthode `SetMaxThreads()` de la classe `ThreadPool` permet de modifier le nombre de threads du pool de threads à partir du code géré (page 167).

Le *framework* .NET 2. 0 propose des nouvelles classes qui permettent de capturer et de propager le contexte d'exécution du thread courant à un autre thread (page 180).

## *Sécurité*

La classe `System.Security.Policy.Gac` permet de représenter un nouveau type de preuve basé sur la présence d'un assemblage dans le répertoire GAC. (page 188).

Les nouvelles classes de permissions suivantes ont été ajoutées : `System.Security.Permissions.KeyContainerPermission`, `System.Net.NetworkInformation.NetworkInformationPermission`, `System.Security.Permissions.DataProtectionPermission`, `System.Net.Mail.SmtpPermission`, `System.Data.SqlClient.SqlNotificationPermission`, `System.Security.Permissions.StorePermission`, `System.Configuration.UserSettingsPermission`, `System.Transactions.DistributedTransactionPermission` et `System.Security.Permissions.GacIdentityPermission`.

La classe `IsolatedStorageFile` présente les nouvelles méthodes suivantes : `GetUserStoreForApplication()`, `GetMachineStoreForAssembly()`, `GetMachineStoreForDomain()` et `GetMachineStoreForApplication()` (page 205).

Le *framework* .NET 2005 permet de lancer un processus fils dans un contexte de sécurité différent de celui de son processus parent (page 135).

Le *framework* .NET 2005 présente de nouveaux types dans l'espace de nom `System.Security.Principal` permettant de représenter et de manipuler des identificateurs de sécurité (page 209).

Le *framework* .NET 2005 présente de nouveaux types dans l'espace de nom `System.Security.AccessControl` pour manipuler les droits d'accès *Windows* (page 211).

Le *framework* .NET 2005 présente de nouvelles implémentations d'algorithme de hachage disponibles dans l'espace de noms `System.Security.Cryptography`.

Le *framework* .NET 2005 présente plusieurs classes permettant d'avoir accès aux fonctionnalités présentées par l'API de protection de données (DPAPI) de *Windows* (page 225).

La classe `System.Configuration.Configuration` permet de gérer très simplement le fichier de configuration d'une application. Notamment, vous pouvez l'exploiter pour encrypter vos données de configuration (page 718).

Le *framework* .NET 2005 présente de nouveaux types dans les espaces de noms `System.Security.Cryptography.X509Certificates` et `System.Security.Cryptography.Pkcs` qui sont

spécialisés dans la manipulation des standards de certification *X.509* et *CMS/Pkcs7* ainsi que dans la manipulation des listes de certificats (page 230).

Le nouvel espace de noms `System.Net.Security` présente les nouvelles classes `SslStream` et `NegociateStream` qui permettent d'utiliser les protocoles SSL, NTLM et Kerberos de sécurisation de flots de données (page 660).

### *Réflexion/Attribut*

Vous avez maintenant la possibilité de charger un assemblage en mode *reflection only* (page 237). D'ailleurs, la classe `AppDomain` présente le nouvel évènement `ReflectionOnlyPreBindAssemblyResolve` déclenché juste avant le chargement d'un assemblage destiné à être utilisé par le mécanisme de réflexion (page 92).

Le *framework* .NET 2005 introduit la notion d'attribut conditionnel. Un tel attribut a la particularité d'être pris en compte par le compilateur C♯2 que si un certain symbole est défini (page 255).

### *Interopérabilité*

Les notions de pointeur sur fonction et de délégué sont maintenant interchangeables grâce aux nouvelles méthodes `GetDelegateForFunctionPointer()` et `GetFunctionPointerForDelegate()` de la classe `Marshal` (page 272).

La classe `HandleCollector` permet de fournir au ramasse-miettes une estimation de la quantité de ressources non gérées couramment détenues (page 277).

Les nouvelles classes `SafeHandle` et `CriticalHandle` permettent d'exploiter les *handles Windows* plus efficacement qu'avec la classe `IntPtr` (page 278).

Les outils `tlbimp.exe` et `tlbexp.exe` présentent la nouvelle option `/tlbreference` permettant de fournir explicitement une libraire de type sans passer par la base des registres. Cela permet la création d'environnements de compilation moins fragiles.

*Visual Studio 2005* présente des facilités pour exploiter à partir d'une application .NET la technologie *reg free COM* de *Windows XP* qui permet d'utiliser une classe COM sans l'enregistrer dans la base des registres (page 287).

Les structures relatives à la technologie COM telles que `BINDPTR`, `ELEMDESC` ou `STATDATA` ont été déplacées de l'espace de noms `System.Runtime.InteropServices` vers le nouvel espace de noms `System.Runtime.InteropServices.ComTypes`. En outre, cet espace de noms contient de nouvelles interfaces qui redéfinissent certaines interfaces standard COM telles que `IAdviseSink` ou `IConnectionPoint`.

L'espace de noms `System.Runtime.InteropServices` contient de nouvelles interfaces telles que `_Type`, `_MemberInfo` ou `_ConstructorInfo` qui permettent au code non géré d'avoir accès au mécanisme de réflexion sur les éléments du code géré. Bien entendu, les classes gérées concernées (`Type`, `MemberInfo`, `ConstructorInfo` etc) implémentent ces interfaces.

### *C♯2.0*

Le chapitre 13 est consacré à la fonctionnalité phare de .NET 2.0 : la généricité.

Le compilateur `csc.exe` présente les nouvelles options `/keycontainer`, `/keyfile`, `/delaysign` (page 33), `/errorreport` et `/langversion`.

C#2.0 apporte les notions de qualificateur d'alias d'espaces de noms (page 320) de qualificateur `global::` (page 321) et d'alias externe (page 322) pour éviter certains problèmes de conflits d'identificateurs.

C#2.0 présente les nouvelles directives de compilation `#pragma warning disable` et `#pragma warning restore` (page 316).

Le compilateur C#2.0 est capable d'inférer la délégation adéquate lors de la création d'un délégué (page 380). Ceci rend le code plus lisible.

Le *framework* .NET 2005 introduit la notion de type nullable exploitable à partir d'une syntaxe spéciale en C#2 (page 385).

C#2.0 permet d'étaler la définition d'un même type sur plusieurs fichiers sources d'un même module (page 392).

C#2.0 permet d'assigner une visibilité différente aux accesseurs d'une propriété ou d'un indexeur (page 419).

C#2.0 permet la définition de classes statiques (page 432).

C#2.0 permet de déclarer un champ de type tableau de taille fixe d'éléments de types primitifs au sein d'une structure (page 507).

L'intellisense de *Visual Studio 2005* vous propose les balises XML des commentaires de types `///` (page 324).

## *Exceptions*

La classe `SecurityException` et *Visual Studio 2005* ont été améliorés de façon à vous permettre de tester et de déboguer plus facilement votre code mobile (page 205).

*Visual Studio 2005* présente un assistant très pratique pour obtenir la totalité des données relatives à une exception qui survient durant le déboguage (page 510).

*Visual Studio 2005* vous permet d'être informé lorsqu'un évènement problématique connu du CLR survient. Ces évènements se traduisent parfois par une exception gérée (page 522).

## *Collection*

L'ensemble des types de collection du *framework* .NET a été complètement revu afin de tenir compte des bénéfices du support de la généricité. Ces types font l'objet du chapitre 15. En page 595 nous exposons un tableau de correspondance entre les types des deux espaces de noms `System.Collections` et `System.Collections.Generic`.

## *Débogage*

L'espace de noms `System.Diagnostics` présente les nouveaux attributs `DebuggerDisplayAttribute`, `DebuggerBrowsable`, `DebuggerTypeProxyAttribute` et `DebuggerVisualizerAttribute` qui vous permettent de personnaliser l'affichage des états de vos objets durant le débogage (page 617).

Vous avez maintenant accès à la fonctionnalité *Edit And Continue* permettant de modifier le code en cours de débogage.

## Classes de bases

Les types primitifs (entier, booléen, nombre à virgule flottante, décimaux) présentent la méthode `TryParse()` qui permet de parser une valeur dans une chaîne de caractères sans lancer d'exception en cas d'échec (page 358).

Le *framework* .NET 2005 présente plusieurs implémentations dérivées de la classe abstraite `System.StringComparer` qui permettent de comparer des chaînes de caractères en tenant compte ou non de la culture et de la casse (page 585).

La nouvelle classe `DriveInfo` permet de représenter et de manipuler des volumes (page 607).

Le *framework* .NET 2005 introduit la notion de source de traçage permettant un paramétrage plus fin des traces (page 622). En outre, les nouvelles classes suivantes de listeners ont été ajoutées : `ConsoleTraceListener`, `DelimitedListTraceListener`, `XmlWriterTraceListener` et `WebPageTraceListener`. (page 621).

De nombreuses fonctionnalités ont été ajoutées à la classe `System.Console` afin de rendre l'affichage des données plus convivial (page 629).

## Entrées/Sorties

Le *framework* .NET 2005 présente la nouvelle classe `System.Net.HttpListener` qui permet d'exploiter la couche HTTP.SYS de *Windows XP SP2* et *Windows Server 2003* pour développer un serveur HTTP (page 654).

En .NET 2005, les classes de l'espace de noms `System.Web.Mail` sont maintenant obsolètes. Pour envoyer des mails, il faut utiliser les classes de l'espace de noms `System.Net.Mail`. En outre, le nouvel espace de noms `System.Net.Mime` contient des classes pour le support de la norme MIME (page 656).

Des nouvelles méthodes permettent de lire ou d'écrire dans un fichier en un seul appel (page 636).

De nouvelles classes sont disponibles pour compresser/décompresser un flot de données (page 659).

Une nouvelle version non gérée `System.IO.UnmanagedMemoryStream` de la classe `MemoryStream` permet d'éviter la copie des données sur le tas des objets du CLR. Elle est donc plus efficace (page 659).

La nouvelle classe `System.Net.FtpWebRequest` implémente un client FTP (page 653).

Le nouvel espace de noms `System.Net.NetworkInformation` contient des types qui permettent de connaître les interfaces réseaux disponibles sur la machine, d'interroger leurs états, d'être averti des changements d'états et d'obtenir des statistiques sur le trafic (page 649).

Des services de cache de ressources web sont disponibles dans le nouvel espace de noms `System.Net.Cache` (page 653).

La nouvelle classe `System.IO.Ports.SerialPort` permet d'exploiter un port série d'une manière synchrone ou par évènement (page 660).

## Windows Forms 2.0

*Visual Studio 2005* exploite la notion de classe partielle au niveau de la gestion des formulaires *Windows Forms*. Ainsi il ne mélange pas le code qu'il génère et votre propre code dans le même fichier (page 669).

*Windows Forms 2.0* présente la classe `BackgroundWorker` qui permet de standardiser le développement d'opération asynchrone au sein d'un formulaire (page 676).

L'apparence (i.e le style visuel) des contrôles est mieux gérée avec *Windows Forms 2.0* car il n'y a plus besoin d'avoir recours à la DLL `comctl32.dll` pour obtenir un style à la *Windows XP* (page 679).

Un rapide aperçu des nouveaux contrôles Windows Forms 2.0 est disponible en page 681.

*Windows Forms 2.0* et *Visual Studio 2005* contiennent un *framework* et des outils de développement rapide de fenêtres de présentation et d'édition de données (page 688).

*Windows Forms 2.0* présente les nouvelles classes `BufferedGraphicsContext` et `BufferedGraphics` qui permettent de gérer finement un mécanisme de double buffering (page 703).

### *ADO.NET 2.0*

ASO.NET 2.0 présente de nouvelles classes abstraites telles que `DbConnection` ou `DbCommand` dans le nouvel espace de noms `System.Data.Common` qui implémentent les interfaces `IDbConnection` ou `IDbCommand`. L'utilisation de ces nouvelles classes pour s'abstraire d'un fournisseur de données sous-jacent est maintenant à préférer à l'utilisation des interfaces (page 712).

ADO.NET 2.0 présente une architecture de classes évoluées de type *fabrique abstraite* qui permet de produire du code d'accès aux données complètement indépendant du fournisseur de données sous-jacent (page 712).

Les quatre fournisseurs de données fournis avec le *framework* présentent de nouvelles facilités permettant de construire une chaîne de connexion indépendamment du fournisseur de données sous-jacent (page 716).

ADO.NET 2.0 présente un *framework* permettant de naviguer programmatiquement dans le schéma d'un SGBD (page 718).

Le moteur d'indexation exploité en interne par le *framework* lorsque vous utilisez des instances des classes `DataSet` et `DataTable` a été refait de façon à être plus performant lors du chargement et de la manipulation des données.

Les instances des classes `DataSet` et `DataTable` sont maintenant sérialisables au format binaire grâce à la nouvelle propriété `SerializationFormat RemotingFormat{get;set;}`. Vous pouvez vous attendre à un gain d'un facteur de 3 à 8 par rapport à la sérialisation XML.

La classe `DataTable` est moins dépendante de la classe `DataSet` car les possibilités XML de cette dernière (décrites en page 775) lui ont été ajoutées.

La nouvelle méthode `DataTable DataView.ToTable()` permet de construire une `DataTable` ayant une copie du contenu d'une vue (page 730).

ADO.NET 2.0 offre un pont entre le mode connecté et le mode déconnecté en permettant aux classes `DataSet`/`DataTable` et `DataReader` de travailler ensembles (page 735).

Les `DataSet` typés de ADO.NET 2.0 prennent en compte directement la notion de relation entre tables. En outre, grâce aux types partiels, le code généré et votre propre code sont séparés dans deux fichiers distincts (page 731). Enfin, vous pouvez coder des requêtes SQL typées avec la nouvelle notion de `TableAdapter` (page 733).

ADO.NET 2.0 offre la possibilité d'optimiser la sauvegarde dans une base des modifications effectuées sur un cache de données en permettant la mise à jour par lot des modifications (page 727).

## ADO.NET 2.0 : Fournisseur de données de SQL Server

Vous avez la possibilité d'énumérer les sources de données de type *SQL Server* (page 716).

Vous avez la possibilité d'obtenir programmatiquement un certain contrôle sur la façon dont le pooling de connexions s'effectue (page 718).

Le fournisseur de données `SqlClient` de ADO.NET 2.0 permet d'effectuer des commandes d'une manière asynchrone (page 738).

Vous pouvez utiliser les services de copie en masse de l'outil *SQL Server* `bcp.exe` grâce à la nouvelle classe `SqlBulkCopy` (page 739).

Vous pouvez obtenir des statistiques concernant l'activité d'une connexion (page 740).

Il existe une version simplifiée, gratuite et librement distribuable de *SQL Server 2005* qui présente de nombreux avantages par rapport aux anciens produits *MSDE* et *Jet* (page 740).

## Transaction

Le nouvel espace de noms `System.Transactions` (contenu dans la DLL `Systems.Transactions.dll`) propose à la fois un modèle de programmation transactionnelle unifié et un nouveau moteur transactionnel dont la particularité est d'être très performant sur certains types de transactions légères (page 741).

## XML

Les performances de toutes les implémentations utilisées pour manipuler des données stockées dans un format XML ont été améliorées significativement (d'un facteur 2 à 4 lors des scénarios d'utilisation classiques selon *Microsoft*).

La nouvelle classe `System.Xml.XmlReaderSettings` permet de préciser les vérifications qui doivent être faites lorsque l'on lit des données XML au moyen d'une classe dérivée de `XmlReader` (page 766).

Il est possible de valider partiellement un arbre DOM chargé dans une instance de `XmlDocument` (page 771).

Il est maintenant possible de modifier un arbre DOM stocké dans un `XmlDocument` au moyen de l'API type curseur de `XPathNavigator` (page 774).

La classe `XslCompiledTransform` remplace la classe `XslTransform` maintenant devenue obsolète. Son atout principal et de compiler les programmes XSLT en code MSIL avant d'appliquer une transformation. Selon *Microsoft*, cette nouvelle implémentation améliore les performances d'un facteur de 3 à 4 (page 774). En outre *Visual Studio 2005* permet maintenant de déboguer un programme XSLT (page 784).

Le support de la classe `DataSet` pour XML a été amélioré. Vous pouvez maintenant charger des schémas XSD avec des noms répétés dans différents espaces de noms et charger du XML avec plusieurs schémas en ligne. En outre, les méthodes de chargement et de sauvegarde des données stockées en XML ont été ajoutées à la classe `DataTable` (page 775).

La version 2005 de *SQL Server* apporte des nouvelles facilités quant à l'intégration de données XML dans une base de données relationnelle (page 779).

La sérialisation XML permet maintenant de sérialiser les informations de type nullable et les instances de types génériques. En outre, il existe un nouvel outil `sgen.exe` permettant de pré générer un assemblage contenant le code pour sérialiser un type (page 779).

## .NET Remoting

Le nouveau canal `IpcChannel` est dédié à la communication entre processus tournants sur la même machine. Son implémentation est basée sur la notion de pipe nommé *Windows* (page 830).

Si vous utilisez un canal de type TCP, vous avez maintenant la possibilité d'utiliser les protocoles NTLM et Kerberos pour authentifier l'utilisateur *Windows* sous lequel s'exécute le client, pour crypter les données échangées et pour impersonifier vos requêtes (page 816).

De nouveaux attributs de l'espace de noms `System.Runtime.Serialization` permettent de gérer les problèmes inhérents à l'évolution d'une classe dont les instances sont sérialisées (page 791).

Il est possible de consommer une instance d'un type générique fermé avec la technologie .NET Remoting, que vous soyez en mode CAO ou WKO (page 499).

## ASP.NET 2.0

*Visual Studio .NET 2005* est fourni avec un serveur web permettant de tester et de déboguer vos applications web en cours de développement (page 862).

Il est aisé de se servir de la couche HTTP.SYS pour construire un serveur web qui héberge ASP.NET sans aucunement avoir recours à IIS (page 865).

ASP.NET 2.0 présente un nouveau modèle de construction des classes représentant les pages. Ce modèle est basé sur les classes partielles et est différent de celui d'ASP.NET 1.x (page 869).

La directive `CodeBehind` d'ASP.NET v1.x n'est plus supportée (page 871).

En ASP.NET 2.0, le modèle de compilation dynamique a considérablement évolué et se base maintenant sur plusieurs nouveaux répertoires standard (page 871). De plus, ASP.NET 2.0 présente deux nouveaux modes de pré compilation (page 872 et page 872).

Pour pallier les effets des *viewstate* volumineux de ASP.NET 1.x, ASP.NET 2.0 stocke les informations dans la chaîne base64 plus efficacement et introduit la notion de *controlstate* (page 879).

ASP.NET 2.0 présente une nouvelle technique pour permettre à une page d'envoyer un évènement *postback* à une autre page (page 880).

Certains évènements ont été rajoutés au cycle de vie d'une page (page 883).

ASP.NET 2.0 présente une infrastructure pour permettre d'effectuer le traitement d'une même requête sur plusieurs threads du pool. Cela permet notamment d'éviter la pénurie de threads du pool qui survient lorsque plusieurs traitements asynchrones longs sont réalisés simultanément (page 886).

De nouveaux évènements ont été ajoutés à la classe `HttpApplication` (voir les MSDN).

La manipulation des fichiers de configuration est simplifiée grâce à l'intellisense de *Visual Studio 2005*, une nouvelle interface web, une nouvelle interface graphique intégrée dans la console de configuration d'IIS et grâce à des nouvelles classes de base permettant de manipuler programmatiquement et surtout, d'une manière fortement typée, les données XML de ces fichiers (page 890).

ASP.NET 2.0 présente un *framework* permettant de gérer d'une manière standard les évènements qui surviennent dans la vie d'une application web (page 906).

Vous pouvez configurer ASP.NET 2.0 pour qu'il détecte s'il peut stocker l'identifiant de session dans un *cookie* coté client et pour se mettre automatiquement en mode URI si le navigateur du client ne permet pas les *cookies* (page 901).

ASP.NET 2.0 vous permet de fournir votre propre mécanisme de gestion de sessions ou de gestion d'IDs de sessions (page 902).

Le moteur de cache d'ASP.NET 2.0 présente plusieurs nouvelles possibilités intéressantes. Vous pouvez maintenant utiliser la sous directive `VaryByControl` dans vos pages (page 920). Vous pouvez substituer des fragments de page dynamiques dans vos pages cachées (page 923). Vous pouvez associer à des données cachées des dépendances vers des tables d'un SGBD type *SQL Server* (page 926). Enfin, vous pouvez créer vos propres types de dépendances (page 928).

ASP.NET 2.0 présente plusieurs nouveaux contrôles serveur permettant de se lier déclarativement à une source de donnée (page 930).

ASP.NET 2.0 présente une nouvelle hiérarchie de contrôles serveurs de présentation et d'édition de données. Ces contrôles ont tous la particularité de pouvoir exploiter un contrôle de source de données pour l'accès en lecture et en écriture (page 935).

ASP.NET 2.0 présente une syntaxe simplifiée pour les templates (page 939).

ASP.NET 2.0 présente la notion de *master page* qui permet de réutiliser simplement un modèle de design de page sur les pages d'un site (page . 947).

ASP.NET 2.0 présente une infrastructure extensible permettant d'insérer dans vos pages des contrôles d'aide à la navigation dans un site (page 954).

Avec ASP.NET 2.0 vous pouvez utiliser le mode d'authentification Forms sans avoir recours aux *cookies* (page 962).

ASP.NET 2.0 permet de gérer les données d'authentification des utilisateurs ainsi que les rôles auxquels ils peuvent appartenir d'une manière standard au moyen d'une base de données (pages 962 et 964). En outre, plusieurs nouveaux contrôles serveurs viennent grandement simplifier le développement d'application ASP.NET supportant l'authentification (page 965).

ASP.NET 2.0 présente un *framework* permettant de stocker et de manipuler d'une manière standard les données personnelles à chaque utilisateur (page 966).

ASP.NET 2.0 présente un *framework* permettant de faciliter la maintenance de l'apparence d'un site (page 971).

ASP.NET 2.0 présente un *framework* permettant de créer des portails webs au moyen de ce que l'on nomme les *webParts* (page (page 973).

ASP.NET 2.0 présente un *framework* permettant de modifier le rendu d'une page HTML si la requête HTTP initiatrice a été émise à partir d'un système avec un écran de petite taille, type système mobile. Concrètement, le rendu de la plupart des contrôles serveurs est modifié pour tenir moins de place. Cette modification se fait grâce à des objets que l'on appelle adaptateurs. Les adaptateurs sont sollicités automatiquement et implicitement par ASP.NET au moment de la phase du rendu. L'article **Inside the ASP.NET Mobile Controls** des **MSDN** constitue un bon point de départ pour aborder cette facette d'ASP.NET 2.0.

## *Web Service*

Les classes proxys générées avec l'outil `wsdl.exe` présentent un nouveau modèle d'appel asynchrone annulable (page 998).

# C

# *Introduction aux design patterns*

Les *design patterns* objets sont des motifs d'interaction entre classes, des motifs d'interaction entre objets, et des motifs de définition de classes. La notion de *pattern* est apparue dans les années 1970, dans le domaine de l'architecture classique (bâtiment etc). Elle a été popularisée dans le domaine de la programmation orientée objet à la fin des années 90 grâce à cet ouvrage de référence :

**Design Patterns, Elements of Reusable Object-Oriented Software**
ADDISON-WESLEY 1994
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
ISBN : 0-201-63361-2

La notion de *pattern* s'est alors invitée dans d'autres domaines de la conception logicielle, notamment en architecture des applications distribuées. L'ouvrage suivant, qui lui aussi fait référence, met en évidence des motifs récurrents (i.e des patterns) concernant la gestion de la persistance et l'échange de données entres les tiers d'une application distribuée :

**Patterns of Enterprise Application Architecture**
ADDISON-WESLEY 2002
Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, Randy Stafford
ISBN : 0-321-12742-0

*Microsoft* s'intéresse et communique depuis quelques années sur les *patterns* de conception logicielle avec ses technologies au travers de son groupe de travail *Pattern & Practices* (*PAG*). Plus d'informations à ce sujet sont disponibles à l'URL `http://msdn.microsoft.com/practices/`.

Les initiés qui pensent en terme de *patterns* parviennent à concevoir rapidement des architectures de meilleure qualité. De plus leurs capacités de communication et d'abstraction sont bonifiées grâce aux noms des *patterns*. Le développeur a tout intérêt à les assimiler et à les utiliser. Chaque *pattern* définit un comportement plus ou moins souple, qui répond à une ou plusieurs problématiques récurrentes de conception logiciel. Voici quelques exemples de problématiques introduites dans le présent ouvrage suivies du nom du *pattern* utilisé pour les résoudre :

- La création des objets sans spécifier explicitement leur classe concrète (***Fabrique abstraite***). Ce pattern est exploité par la technologie *ADO.NET* notamment pour éviter à une application de se coupler avec un fournisseur de données particulier (voir page 715). Il est aussi utilisé par la technologie .NET Remoting pour éviter d'exposer les implémentations des objets serveurs aux clients (voir page 802).
- L'interception des appels à un objet (***Proxy***). Plus de détails sont disponibles page 817.
- La définition d'une relation *one-to-many* entre un objet sujet et des objets de façon à ce que chacun des objets dépendants soient notifiés des changements d'état de l'objet sujet (***Observateur***). La notion d'évènement de C# constitue une excellente alternative à l'implémentation de ce *pattern* (voir page 411).
- La possibilité de pouvoir choisir l'implémentation d'une fonctionnalité après que le code ait été compilé grâce à un fichier de configuration (***Provider***). Ce *pattern* est très exploité par *ASP.NET*, notamment pour permettre de fournir vos propres extensions aux fonctionnalités de ce *framework* (voir page 902).
- Déléguer le choix d'une implémentation jusqu'au moment de l'exécution en ayant recours à des liens tardifs (***Plugin***). En page 247 nous exposons un exemple d'utilisation.
- La possibilité de pouvoir accéder aux éléments d'une collection d'une manière séquentielle sans se préoccuper de la représentation physique de ces éléments (***Itérateur***). C# 2.0 présente des facilités pour implémenter ce *pattern* (voir page 542).
- Permettre à des objets qui ne se connaissent pas d'interagir grâce à une classe qui encapsule ces interactions (***Médiateur***). Ce *pattern* est exploité par la technologie *Windows Form* pour faciliter le développement des formulaires (voir page 667).
- Le contrôle du nombre d'instance d'une classe (***Singleton***). Les constructeurs de classe de .NET permettent de faciliter l'implémentation de ce *pattern* (voir page 431).
- L'ajout de responsabilités à un objet durant l'exécution (***Décorateur***). Ce pattern est exploité par le *framework* .NET notamment pour ajouter des services à un flot de données (voir page 657).
- Créer à volonté des objets à partir d'un objet prototype connu par une référence de type une interface qui ne contient qu'une méthode de clonage (***Prototype***). L'interface du *framework* `System.ICloneable` présentée page 347 est dans une certaine mesure adaptée à ce rôle.
- Maintenir la liste de changements effectués durant une opération et coordonner la persistance de ces changements en gérant les éventuels problèmes dus à la concurrence (***Unité de travail***). Ce pattern est utilisé pour sauver les changements effectués sur les données d'un `DataSet` dans la technologie ADO.NET (voir page 725).
- Obtenir une implémentation *thread-safe* d'une classe (***SyncRoot***). Plus de détails sont disponibles en page 149.

Vous pouvez compléter cette lecture par l'article **Discover the Design Patterns You're Already Using in the .NET Framework** de *Rob Pierry* disponible dans le numéro de *Juillet 2005* de **MSDN Magazine** à l'URL : `http://msdn.microsoft.com/msdnmag/issues/05/07/DesignPatterns/default.aspx`.

# D

# *Les outils*

Vous pouvez consulter le site `http://sharptoolbox.com` qui maintient une liste très complète des outils disponibles autour du développement pour la plateforme .NET. Voici la liste des outils dont nous parlons dans le présent ouvrage :

### *Assemblages*

`ildasm.exe` page 21

Reflector page 24

`resgen.exe` page 35

`al.exe` page 37 et page 67

`ilasm.exe` page 41

`ILMerge` page 17

### *Construction/Déploiement*

`msbuild.exe` page 49

`GACUtil.exe` page 65

`mage.exe` page 77

`mageui.exe` page 77

`fuslogvw.exe` page 106

`ngen.exe` page 113

### *Profiling*

CLR Profiler page 119

### *Sécurité*

`caspol.exe` page 195

`mscorcfg.msc` page 195

`permview.exe` page 204

`certmgr.exe` page 231

`permcalc.exe` page 82

### *Interopérabilité*

`regsvr32.exe` page 292

`regasm.exe` page 292

`tlbimp.exe` page 279

`tlbexp.exe` page 289

`guidgen.exe` page 20

`regsvcs.exe` page 304

### *C♯*

`csc.exe` page 317

NDoc page 326

### *Windows Forms*

HTML Help WorkShop page 679

### *Sérialisation XML et datasets typés*

`xsd.exe` et les datasets typés page 731

`xsd.exe` et la sérialisation XML page 782

`sgen.exe` page 782

### *Mapping Objet/Relationnel*

Des liens vers les outils les plus populaires sont disponibles en page 737

### *Sql Server*

`bcp.exe` page 739

### *.NET Remoting*

`soapsuds.exe` page 804

### *ASP.NET 2.0*

`aspnet_regiis.exe` page 863

`aspnet_compiler.exe` page 872

`aspnet_regsql.exe` page 963

### *Service web*

`wsdl.exe` page 994

X509 Certificate Tool page 1009

Policy Wizard page 1009

### *Diagnostic*

`perfmon.exe` page 906 et page 116

### *Programmation Orientée Aspect*

AspectDNG page 466

Enfin, nous signalons l'existence de l'outil ***NDepend***. Cet outil analyse statiquement le code compilé de n'importe quelle application .NET et n'est aucunement intrusif. En plus de calculer la plupart des métriques de code à partir de cette analyse il sait détecter un certain nombre de problèmes de *design*.

# *Index*

Composition et mise en page :
*Atelier Fluxus Virus*, 11 rue Alsace-Lorraine, F-29200 Brest
http://www.fluxus-virus.com