Le langage C#

Christine Eberhardt

CAMPUSPRESS

CampusPress a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, CampusPress n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

CampusPress ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Publié par CampusPress
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00

Auteur : Christine Eberhardt

Mise en pages : Andassa

ISBN : 2-7440-1362-5

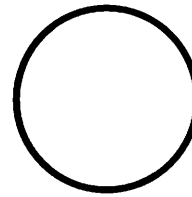
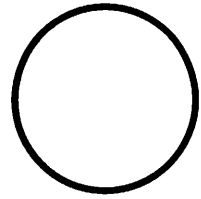


Table des matières



Introduction

Vous avez dit see sharp ?

C# (pour *see sharp*) est le nouveau langage de programmation orienté objet de Microsoft. Comme son nom l'indique, il a été créé en conservant le meilleur des langages C et C++ et apporte encore d'autres améliorations. A son apparition, beaucoup l'avaient surnommé J--, trouvant un peu trop rapidement, mais la critique est facile, que ce n'était qu'une traduction de Java à la sauce Microsoft.

Depuis près de vingt ans, le C et le C++ sont les langages les plus utilisés pour le développement de logiciels. La raison de ce succès tient principalement au contrôle puissant et précis que le programmeur peut avoir sur son code. Malheureusement, cette souplesse est obtenue au détriment de la vitesse de développement, en raison de la complexité et des délais liés à ces langages. Beaucoup de développeurs se sont alors tournés en 1994 vers le langage Java. Celui-ci, en plus d'une très grande simplicité, proposait des innovations majeures, notamment des programmes portables : le code créé peut s'exécuter sur n'importe quelle plate-forme, il suffit de le recompiler pour qu'il puisse être interprété par tout système d'exploitation possédant un interpréteur. En outre, Java est extrêmement sécurisé : ainsi, il est impossible de compiler un programme comprenant des erreurs, d'écrire en dehors des limites d'un tableau. De plus, ce langage évite les fuites de mémoire grâce au "garbage collector" (nettoyeur de mémoire), qui libère la mémoire occupée

par tous les objets dont la vie est terminée. Malheureusement, Java (principalement à cause de son mode de compilation) se révèle plus lent que d'autres langages objet comme le C++.

Beaucoup de programmeurs attendaient un langage offrant un meilleur équilibre entre puissance et souplesse. La solution idéale allierait à la rapidité de développement du langage Java, à sa simplicité, à sa conception objet et à sa sécurité la puissance et le contrôle du C et du C++. Le tout avec un accès à toutes les fonctionnalités de la plate-forme pour laquelle le programme est développé. Il fallait enfin un environnement de programmation en synchronisation totale avec les normes Web émergeantes, permettant une intégration aisée aux applications existantes. C'est là la définition de C#. Ce langage permet en particulier de convertir des composants en services Web qui pourront être appelés sur Internet à partir de n'importe quel langage s'exécutant sur n'importe quel système d'exploitation. Et surtout, C# est conçu pour apporter la rapidité de développement au programmeur C++, sans toutefois diminuer la puissance et le contrôle qui caractérisent depuis toujours le C et le C++. Du fait de cet héritage, C# reste très fidèle au C et au C++ sans pour autant s'éloigner de Java. Les développeurs habitués à ces langages peuvent rapidement devenir productifs en C#. Ils vont aussi adorer la simplicité de ce langage.

Il n'existe aucune restriction sur le type de programmes que vous pouvez créer. Vous pouvez envisager d'écrire un livre de recettes électronique pour votre grand-mère, un programme de surveillance des utilisateurs de votre propre ordinateur, un fouineur Web pour trouver toutes les pages consacrées à Michael Jackson, ou même votre propre compilateur de langage. La seule limite, c'est celle de votre imagination.

Les ingénieurs de Microsoft le décrivent tout simplement comme un langage simple, moderne et orienté objet.

Pour ceux d'entre vous qui n'ont jamais pratiqué les langages C++ ou Java, ne soyez pas traumatisés par les termes barbares de la section qui suit puisque, justement, ils vous seront épargnés dans cet ouvrage.

En effet, les notions les plus compliquées de l'orienté objet, comme les macros, les modèles, l'héritage multiple et les classes de base virtuelles, n'existent pas dans C#.

Une autre simplification concerne l'accès aux divers membres. Avec le C++, vous disposez des trois opérateurs `:`, `.` et `->`, et leur utilisation exige une certaine maîtrise du langage. Avec C#, réjouissez-vous, ils sont tous remplacés par l'unique opérateur "point".

Vous constaterez aussi une nette diminution des mots clés (voir annexe) et, nous avons gardé le meilleur pour la fin, l'utilisation des pointeurs a aussi été simplifiée.

Pour terminer, C# est une pièce de la stratégie .NET de Microsoft. C'est le langage le plus adapté pour développer des applications destinées au nouveau .NET Framework de Microsoft, présenté dans la section suivante.

Microsoft .NET Framework

Le langage de programmation C# fait partie des langages .NET et il ne peut être utilisé sans le .NET Framework.

L'objectif de la stratégie .NET de Microsoft, c'est de faire d'Internet une véritable plate-forme de programmation distribuée, permettant aux ordinateurs, systèmes et services de communiquer et de collaborer entre eux. Cela signifie que, dans cet environnement, un code Python pourra instancier un objet C# qui hérite d'une classe `Eiffel`. Si cette phrase n'a aucun sens pour vous maintenant, vous la comprendrez après avoir parcouru les cinq premiers chapitres. Retenez simplement qu'au sein de la plate-forme .NET, des programmes écrits dans différents langages pourront faire appel à des fonctionnalités développées dans d'autres langages.

Le .NET Framework a un rôle essentiel dans cette stratégie. Il se compose de trois grandes parties :

- Le CLR ou runtime (*Common Language Runtime*, moteur d'exécution en langage commun).

  Avec les langages .NET, vous ne compilez plus le code des programmes directement en code machine mais dans un langage intermédiaire nommé IL. Ce code est ensuite pris en charge par le runtime : celui-ci se charge de la compilation finale au moment de l'installation ou de la pre-

mière exécution du programme, gère la mémoire et la durée de vie des objets. Le .NET Framework n'étant pas sensible à un langage particulier, vous pouvez construire les programmes .NET dans de nombreux langages, y compris le C++, Microsoft Visual Basic.NET, Jscript, sans oublier C#. Un grand nombre de langages d'autres éditeurs, comme COBOL, Eiffel, Perl, Python ou Smalltalk, sont également utilisables.

- Les bibliothèques de classes, qui représentent une manne de fonctionnalités dans lesquelles les programmeurs des langages .NET peuvent puiser.
- ASP.NET, la nouvelle version d'ASP.

La version bêta 2 du .NET Framework peut être exécutée sur Windows XP, Windows 2000, Windows 98 ou Me, et Windows NT 4.0.

Pour terminer, quelques précisions sur le terme CLS que vous ne manquerez pas de rencontrer dans la documentation du .NET Framework. Si on peut considérer le CLR comme l'union de fonctionnalités de plusieurs langages de programmation, la spécification de langages communs (CLS, *Common Language Specification*) constitue un sous-ensemble de ces fonctionnalités. Ce n'est pas l'ensemble des fonctionnalités communes à tous les langages, mais plutôt un ensemble de fonctionnalités communes à plusieurs langages de programmation. Le CLS représente un *niveau de conformité* que la plupart des langages doivent pouvoir atteindre si les développeurs de compilateurs désirent qu'il y ait interopérabilité. Ce sous-ensemble de fonctionnalités correspond aussi à ce que de nombreux outils conformes au .NET Framework cherchent à atteindre.

Connaissances requises

Pour apprendre à programmer en C#, il n'est pas nécessaire de connaître le C, et encore moins le C++ ou Java. Cependant, pour ne pas rendre la lecture de cet ouvrage trop assommante pour ceux qui maîtrisent déjà les notions de base, nous les aborderons très rapidement et signalerons au lecteur comment et où trouver des informations complémentaires.

Comment démarrer

La première étape consiste bien sûr à vous procurer le .NET Framework et à l'installer sur votre machine. La version bêta 2 du SDK .NET Framework est disponible sur **http://www.microsoft.com/france/msdn/** (saisissez .NET Framework dans la fenêtre *Recherche sur l'espace*). Vous pouvez télécharger cette version bêta ou encore commander le CD sur MS Developer Store (**http://developerstore.com/devstore/**).

Si vous disposez déjà de la version bêta 2 de Visual Studio.NET, vous n'avez pas besoin d'installer le SDK .NET Framework. Visual Studio.NET inclut déjà le SDK.

Nous n'allons pas traiter l'étape d'analyse du problème qui doit précéder toute programmation, mais gardez tout de même en mémoire que cette phase est primordiale. Elle permet d'obtenir des programmes mieux conçus, donc plus performants et plus fiables.

Une fois que vous avez bien identifié les objectifs du programme, vous devez commencer par en écrire le code source, histoire de bien expliquer à l'ordinateur ce qu'il doit faire. Vous pouvez utiliser l'éditeur le plus simple comme le Bloc-notes ou WordPad de Windows, ou bien vi, ed, emacs si vous travaillez sous UNIX. Vous pouvez aussi employer des éditeurs plus évolués comme Microsoft Word ou AppleWorks, à condition d'enregistrer le fichier au format texte seulement, cela pour ne pas introduire de caractères parasites.

Première étape : création du code source

L'idéal est bien sûr d'acquérir un éditeur spécialement conçu pour la programmation, qui va afficher les différents éléments du code en couleurs et offrir une aide précieuse pour la correction des erreurs ou pour indiquer la syntaxe. Le plus connu est le dernier logiciel de la famille Visual Studio, Visual C#, mais il n'est pas incontournable, même si on trouve encore peu d'éditeurs. Voici une sélection des éditeurs disponibles au moment de l'édition de ce livre.

Visual Studio .NET bêta 2 en version française

Lors de la rédaction de cet ouvrage, ce logiciel pouvait être commandé **gratuitement** sur **http://www.microsoft.com/france/msdn/beta/default.asp**. C'est l'outil de développement d'applications sur la plate-forme .NET. Le développement d'applications est simplifié grâce à des fonctionnalités de développement intégrées et à des composants réutilisables. Il s'adresse cependant à des programmeurs chevronnés.

SharpDevelop de Mike Kruger (site en anglais, mais éditeur en français pour la plate-forme .NET de Microsoft)

SharpDevelop est un IDE (*Intelligent Drive Electronic*), un environnement de développement conçu pour la création d'applications basées sur le réseau .NET. Ce logiciel disponible en Open Source est gratuit en français. Il détecte automatiquement l'environnement .NET Framework. Une des commandes de la barre de menus permet de lancer l'exécution du programme. L'outil idéal pour vous exercer avec les exemples de cet ouvrage, même s'il souffre encore de quelques imperfections. Evitez d'utiliser, par exemple, la fonction Rechercher/Remplacer.

SharpDevelop (1 504 Ko) est disponible en téléchargement sur le site **http://www.icsharpcode.net/OpenSource/SD/default.asp**.

Poorman IDE

Duncan Chen a eu la bonne idée de créer cet éditeur pour tous les développeurs qui désirent mettre les mains dans le cambouis C# avant d'installer la plate-forme de Microsoft .NET. Il fournit la coloration syntaxique pour C# et VB7.

Téléchargez-le depuis **http://www.geocities.com/duncanchen/poormanide.htm**.

Jext pour UNIX, Windows ou Mac OS (en anglais)

Un éditeur de texte Java gratuit qui fournit la coloration syntaxique pour les langages ASP, C, C++, Eiffel, Python, Java, JSP, Perl, PHP, HTML, TeX, XML, etc. Il est disponible en téléchargement sur le site **www.jext.org/download.html** (2 458 Ko).

Sauf pour la plate-forme Windows, vous devez aussi installer un JDK (SDK), version 1.2 au minimum. Il s'agit de la plate-forme de développement Java. La version 1.4 est disponible depuis février 2002 (consultez le site **java.sun .com**).

CodeWright (en anglais, pour Windows uniquement, 16 650 Mo !)

CodeWright est aussi un IDE supportant les langages ASP, XML, HTML, C#, Perl, Python et plus encore. Il comprend un ensemble d'outils pour gérer vos projets et vous permet de travailler dans un environnement multi-plate-forme et multilangage.

Vous trouverez toutes les informations le concernant sur le site **www.star-base.com**. Cliquez sur le bouton Download, puis sur Programming tools et enfin sur CodeWright evaluation. Vous avez trente jours pour tester le produit avant de passer à la caisse : il coûte 249 $ (à peu près 287 €) si vous le téléchargez, 299 $ (environ 344 €) si vous voulez recevoir les CD.

Context (en anglais, pour Windows uniquement)

L'éditeur Context est gratuit, et vous avez la possibilité de remplacer tout appel du Notepad Windows par un appel à ce dernier. Il supporte aussi les langages C, C++, Perl, PHP, Assembleur 80x86, Delphi/Pascal, Java, JavaScript, Perl/CGI, Python, Visual Basic, HTML, SQL, TCL/Tk, etc. Vous pouvez le télécharger depuis le site **http://www.fixedsys.com/context** (1 098 Ko).

EditPlus (en anglais, pour Windows uniquement)

Cet éditeur de texte shareware peut remplacer Notepad, mais il offre aussi de nombreuses fonctions intéressantes, par exemple la coloration syntaxique, pour des auteurs de pages Web ou des programmeurs. Il ne coûte que 30 $ (environ 34 €) si vous décidez de le conserver au-delà du mois d'essai. Téléchargez-le depuis le site **www.editplus.com** (970 Ko).

Deuxième étape : compilation des programmes

Pour exécuter un programme, vous devez transformer le fichier texte contenant le code source en fichier exécutable avec un compilateur. Ce dernier traduit vos instructions texte en langage compréhensible par la machine.

Les programmes C# ne fonctionnent pas tout à fait comme les programmes écrits dans d'autres langages. Avec un programme C ou C++ par exemple, il faut créer une "version exécutable" du programme pour chaque plate-forme sur laquelle il doit s'exécuter, et ce fichier est directement produit en langage machine. Vous allez donc créer un fichier pour un Mac, un autre pour un PC, un autre pour votre machine UNIX, etc.

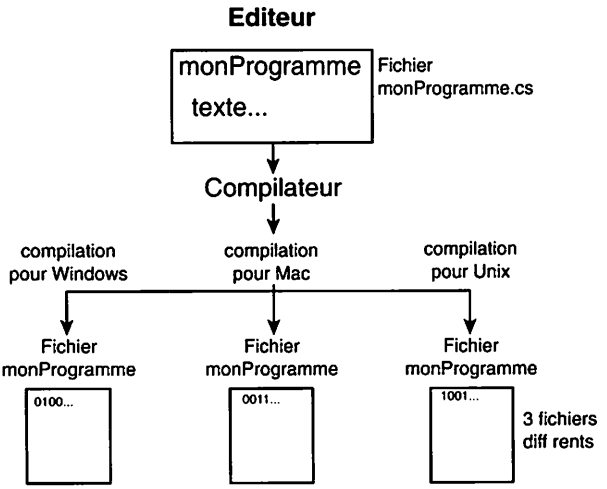


Figure 1.1
Compilation d'un programme en langage traditionnel.

Dans le cas d'un programme C#, vous appelez le compilateur avec la commande `csc` *`nom fichier.cs`*. L'exécutable obtenu n'est pas un fichier ordinaire, c'est un exécutable .NET qui comporte ces éléments importants :

- Le langage intermédiaire (IL, *Intermediate Language*).
- Les *métadonnées* décrivant les types (classes) définis par le programmeur dans l'application et les types du .NET Framework auxquels l'application fait référence, mais qui se situent dans d'autres *assemblages* (assembly en anglais, comme MsCorLib.dll ou System.dll).
- Un *manifeste* décrivant les fichiers qui composent l'assemblage de votre application. En réalité, le manifeste fait partie des métadonnées.

NOUVEAU

En langage .NET, un *assemblage* est une collection d'un ou de plusieurs fichiers qui vont être déployés ensemble. L'assemblage de la plupart des programmes fournis en exemple dans ce livre sera uniquement constitué du fichier `.exe`, et cette caractéristique sera notée dans le *manifeste* intégré à l'exécutable.

Vous pouvez ensuite exécuter ce fichier `.exe` sur de multiples plates-formes à condition que ces machines disposent du CLR de Microsoft ou de tout autre environnement d'exécution compatible avec C#. Cet environnement réalise dynamiquement la dernière étape de "traduction" du code en langage machine selon la plate-forme sur laquelle il s'exécute.

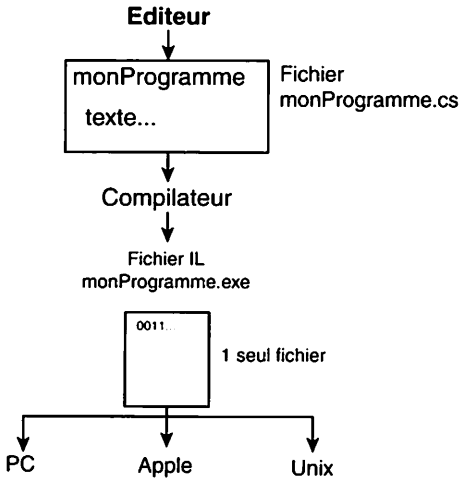


***Figure* I.2**
Compilation d'un programme C#.

Troisième étape : correction des erreurs

Tant que le programme contient des erreurs, le compilateur ne produit pas le fichier IL. Corrigez-les en fonction des indications fournies, puis exécutez de nouveau la compilation jusqu'à ce que l'opération s'achève avec succès.

Quatrième étape : exécution du programme

Au moment où le programme est installé sur la machine, il est en version IL. Quand vous l'exécutez pour la première fois, l'environnement d'exécution ou le CLR le convertit une fois pour toutes en langage machine. Cette première exécution est donc plus longue que les suivantes, mais une fois cette dernière opération de compilation réalisée, le programme est aussi performant que le programme équivalent écrit en C ou en C++.

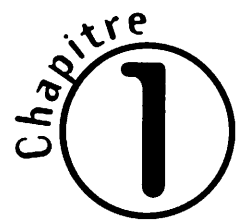
Sites Web

Voici une sélection de pages intéressantes (c'est le mot clé `csharp` qui donne les meilleurs résultats pour un moteur de recherche) :

- C# Corner est un site américain dédié aux programmeurs de tout langage .NET. C'est une véritable mine de codes sources et d'informations si vous maîtrisez l'anglais : **http://www.c-sharpcorner.com**.
- **www.ce1.fr**, un site très intéressant qui propose des articles, forums, tutoriels, exemples de code pour les programmeurs Visual Basic, C#, ASP et Web.
- **http://www.icsharpcode.net**, site en anglais qui propose un compilateur gratuit.
- **C-Sharp:VIX**, site en anglais dédié à C# et à la technologie .NET (tutoriels, cours, exemples, etc.), mais aussi au C, au C++, Java, ASP, PHP, JavaScript, OpenGL, DirectX, etc. : **http://csharp.free.fr/**.
- **www.Programmationworld.com**, site développé par une joyeuse troupe de programmeurs réunis par la passion de la programmation et l'envie de transmettre un savoir de qualité au plus grand nombre. Ce site propose

de nombreux cours, news, forums et liens dédiés à Java, DirectX, C#, COM, .NET Framework, OpenGL, Windows, etc.

- Le site Scriptol, **http://www.scriptol.org**, propose un guide des langages de programmation, des compilateurs gratuits, des IDE et de nombreuses autres ressources concernant Python, PHP, NetRexx, Jython, CSharp, etc. C'est aussi le site de quelques outils : un gestionnaire de fichiers et un générateur de pages Web, le seul et unique convertisseur du C en C++ actuel (écrit en Python), et quelques autres.
- Le site en anglais CodeGuru propose des exemples de code source pour les programmeurs C++, C#, .NET et Visual Basic : **http://www.codeguru.com**.
- Enfin, le site officiel de Microsoft Visual Studio .NET **http://msdn.microsoft.com/vstudio**, et le forum aux questions .NET : **http://www.microsoft.com/france/internet/dotnet/framework/faq.asp**.



Qu'est-ce que l'orienté objet ?

Nous ne pouvons pas plonger dans le code C# avant d'avoir présenté les grandes lignes de la programmation orientée objet. Si vous ne comprenez pas tout, pas d'affolement. Les choses vont s'éclaircir par la suite à mesure que vous découvrirez comment appliquer ces règles dans le code de vos programmes.

Objets et classes d'objets

L'objectif de la programmation est de fournir une solution à un problème, et il existe de multiples façons de parvenir à ce résultat. Dans le cas de la programmation orientée objet, vous n'allez pas raisonner en termes de données et de fonctions pour les manipuler, mais uniquement en termes d'objets. Les objets se comportent comme des boîtes noires dans lesquelles vous rangez des propriétés et des caractéristiques bien spécifiques. Nous y reviendrons un peu plus loin. Si votre programme doit gérer des personnes, vous devrez créer des objets représentant chacune d'elles. Le choix des objets se fait au cours de l'étape d'analyse qui doit précéder toute programmation. Cette analyse présente peu d'intérêt pour développer des programmes courts comme ceux que nous allons étudier, mais elle est primordiale dans le cadre de projets complexes et importants.

Héritage

Les objets sont regroupés en catégories appelées *classes*. Si nous voulions créer un programme pour gérer un zoo, par exemple, nous devrions modéliser ce zoo en créant un objet pour chacun de ses éléments. Le regroupement par catégorie n'est pas difficile à imaginer, vous allez facilement obtenir les classes `Singe`, `Lion`, `Oiseau`, `Reptile`, etc. Si le programme concerne plus particulièrement l'alimentation des animaux, il pourrait être utile de les regrouper dans les classes `Carnivore`, `Végétarien`, `Insectivore`, etc. Cette analyse se traduit par un schéma hiérarchique représentant les classes allant de la plus générale (classe mère) à la plus "spécialisée" (classe fille), jusqu'à ce que nous obtenions une représentation acceptable du monde réel. On dit alors que les classes plus spécialisées *dérivent* de la classe générale située au niveau supérieur. En effet, elles *héritent* (encore un terme important en orienté objet qui justifie l'appellation "classe mère" et "classe fille") de certaines particularités de la classe dont elles sont dérivées. Ce lien qui lie la fille à sa mère se nomme *héritage*.

Quand toutes les classes ont été définies, vous créez les objets de type `Gorille` ou `Lion` spécifiques au dernier niveau de l'arborescence des classes pour représenter chaque élément de l'environnement à gérer (dans notre cas, tous les pensionnaires du zoo). Ces objets sont aussi nommés *instances de classe*. Toute classe peut ainsi avoir un ou plusieurs "représentants" qui possèdent toutes les caractéristiques de la classe dont ils sont issus.

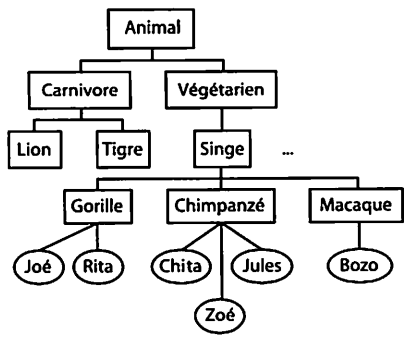


Figure 1.1
Représentation orientée objet de l'environnement du problème à résoudre dans le programme.

L'héritage présente deux avantages importants. Il permet de spécialiser les classes existantes en écrivant uniquement le code modifié. Un `Carnivore`, par exemple, est un `Animal`, et il possède toutes les caractéristiques de cette classe. Ces dernières vont être héritées, il n'est donc pas nécessaire de les redéfinir. Un carnivore est un animal qui se nourrit de viande, nous pouvons donc définir le membre `proies` bien spécifique à cette catégorie d'animaux. Il n'est plus utile de "copier" et "coller" le code, opération qui pose toujours des problèmes en cas de modification puisqu'il faudra retrouver toutes les occurrences d'un bloc de code donné. Grâce à l'héritage, une modification effectuée dans une classe mère est automatiquement répercutée au niveau de ses classes filles.

Encapsulation

Nous venons de voir qu'un programme orienté objet gérait des objets regroupés en catégories nommées classes et organisées de façon hiérarchique. Les classes de l'orienté objet jouent le même rôle que les types de données gérés en programmation non orientée objet. Dans ces programmes, vous manipuliez des entiers, des nombres en virgule flottante ou des caractères, et pour effectuer le traitement nécessaire, vous exécutiez des fonctions récupérées dans une bibliothèque de fonctions ou des fonctions que vous aviez créées.

En programmation orientée objet, vous regroupez les données et les fonctions pour les manipuler dans des classes. Cette technique se nomme *encapsulation*. C'est une notion fondamentale à la base de la puissance de ce type de programmation. En effet, pour utiliser les objets, il suffit de connaître leur "interface utilisateur", inutile de connaître les détails de leur fonctionnement.

Cette notion existe depuis longtemps dans notre vie quotidienne puisque tout le monde sait utiliser un téléviseur, par exemple, ou tout autre appareil ménager courant quel que soit son fabricant. Il suffit de savoir qu'il possède une fonction "Marche/Arrêt", une fonction de sélection de la chaîne, une fonction de réglage du son, etc. Vous exécutez ces fonctions par l'intermédiaire des boutons prévus à cet effet sur l'appareil ou sur sa télécommande (l'interface utilisateur). Ainsi, du moment que vous connaissez les différentes fonctions disponibles, vous êtes capable de les exécuter même si vous ne savez pas comment est câblé l'appareil, ni quels sont ses composants électroniques.

Pour revenir à notre programme de gestion du zoo, vous allez utiliser par exemple un objet de type `chimpanzé`. La classe `Chimpanze` va définir les caractéristiques de l'objet (`couleurPoil`, `mammifère`, `longueurQueue`, etc.) et son comportement avec des fonctions (`manger()`, `crier()`, etc.). Après avoir créé l'objet `chita` dans le programme, vous aurez accès à toutes ses caractéristiques en codant `chita.longueurQueue` ou `chita.couleurPoil`, et vous pourrez exécuter les fonctions de cet objet particulier en codant `chita.manger()` ou `chita.crier()`. Simple, n'est-ce pas ?

Polymorphisme

Poly signifie plusieurs et *morphe* signifie forme. Le *polymorphisme*, autre concept très important de l'orienté objet, est donc la capacité à prendre plusieurs formes. Ce mécanisme complexe sera détaillé par la suite. Retenez simplement qu'il prend deux formes.

Vous pouvez tout d'abord redéfinir dans des classes filles une fonction déjà définie dans une classe de base. Dans notre exemple, nous pourrions redéfinir la fonction générale `Manger()` dans les classes `Vegetarien` et `Carnivore` pour l'adapter à ces deux catégories d'animaux, et même la redéfinir encore au niveau d'une classe plus spécifique pour se rapprocher du comportement réel de l'animal. Cela permet d'écrire du code générique comme :

```
unAnimal.Manger();
```

`unAnimal` étant une sorte de pointeur, cette instruction se traduit par "*appeler la fonction `Manger()` associée au type de l'objet représenté par `unAnimal`*". Même si vous ne savez pas quel type d'animal sera traité dans cette instruction, vous êtes assuré que la bonne version de la fonction `Manger()` sera exécutée et que vous ne donnerez pas de viande à vos perroquets !

L'autre forme de polymorphisme se traduit par la possibilité d'appeler un objet ou une fonction de différentes façons et d'obtenir le même résultat. Dans un programme de géométrie, par exemple, vous pourriez créer deux fonctions différentes nommées `Aire()` pour calculer l'aire d'un cercle. La première recevrait la valeur du rayon et appliquerait la formule simple $3,14 \times \text{rayon}^2$.

La seconde recevrait les coordonnées de 3 points du cercle et effectuerait des calculs plus complexes. C'est justement le nombre et le type des éléments transmis en entrée à la fonction qui permettent au programme de déterminer quelle "version" de la fonction il va exécuter.

Votre premier programme

Cette section est uniquement destinée à vous familiariser avec le processus et les outils de développement, ne vous inquiétez pas si le contenu du code reste en partie obscur.

Ce chapitre et les suivants partent de l'hypothèse que vous avez installé la plate-forme de développement .NET, ou le compilateur avec l'environnement d'exécution de votre choix, et que vous avez configuré votre système comme cela est expliqué dans la documentation du logiciel.

Création, compilation, correction et exécution de votre premier programme

Ouvrez votre éditeur de texte et saisissez le programme suivant en respectant la mise en forme :

Code 1.1 : Premier programme *hello.cs*

```
class hello
{

  public static void Main()
  {
    System.Console.WriteLine("Hello C#");
  }

}
```

Les retours à la ligne et le décalage des lignes de code n'ont aucune signification particulière en C#, mais ils simplifient considérablement la relecture du programme. N'oubliez jamais qu'après seulement quelques semaines, votre cerveau aura libéré la mémoire occupée inutilement par tous les détails de

programmation du projet achevé ou interrompu et que vous devrez analyser de nouveau le code pour poursuivre ou corriger le projet. A ce moment-là, votre degré de frustration sera inversement proportionnel au soin que vous aurez apporté à la rédaction du programme.

NOUVEAU

En C#, les majuscules et les minuscules sont considérées comme des caractères différents. Ainsi, la fonction `Main` est entièrement en minuscules dans les langages C et C++, et vous obtiendrez une erreur de compilation si vous oubliez la majuscule en C#.

Créez un répertoire pour tous les tests de cet ouvrage, nommé CodesEnStock par exemple, dans le dossier réservé pour vos sources C#. Sauvegardez ensuite ce code source dans le fichier **`hello.cs`** de ce répertoire.

Compilation

Si vous utilisez un environnement de développement intégré comme Visual C#, SharpDevelop ou Poorman, il suffit de cliquer sur l'option de menu ou sur l'icône appropriée pour que le fichier IL soit généré.

Si vous vous servez d'un simple éditeur de texte, ouvrez une fenêtre MS-DOS et tapez-y :

```
csc hello.cs
```

Validez ensuite par la touche Entrée. Si vous obtenez une erreur, c'est que le nom du fichier source que vous avez spécifié compte des fautes ou ne se trouve pas dans le répertoire où vous avez appelé **`csc`**. Cette commande a pour effet d'appeler le compilateur (**`csc.exe`**) et de lui faire compiler le fichier source. Si ce dernier contient des fautes de frappe, le compilateur affiche aussi un message d'erreur. Corrigez le fichier, puis exécutez de nouveau la compilation.

Un fichier **`hello.exe`** avec le code IL va alors être créé dans le même répertoire.

Correction des erreurs

Un programme peut contenir deux types d'erreurs : des erreurs de frappe ou de syntaxe, et des erreurs de logique. Les premières sont détectées par le compilateur ou directement dans l'éditeur s'il s'agit d'un véritable éditeur de code (s'il ne reconnaît pas une variable ou une fonction, elle ne sera pas affichée dans la bonne couleur). Si vous oubliez le point-virgule dans notre programme `hello.cs`, par exemple, vous allez recevoir ce message :

```
hello.CS(6,41): error CS1002: ; expected
```

Le compilateur vous signale gentiment qu'à la ligne 6, au 41e caractère, il s'attendait à trouver un point-virgule, mais que ce dernier est absent. Hélas, il n'est pas toujours aussi précis et l'erreur ne se trouve pas toujours à l'emplacement spécifié. Dans ce cas, examinez la ligne située immédiatement avant.

La correction et la mise au point des programmes sont une phase sensible en programmation, et plus vous aurez acquis d'expérience dans ce domaine, plus vous interpréterez rapidement les indications plus ou moins nébuleuses du compilateur. Si vous oubliez la majuscule de la fonction `Main`, par exemple, vous allez recevoir ce message :

```
error CS5001: Program 'hello.exe' does not have an entry
point defined
```

Ce message est beaucoup moins explicite, surtout si vous avez besoin d'un dictionnaire pour le traduire.

ASTUCE

Si le compilateur signale 4 853 erreurs, pas de panique ! Corrigez celles que vous reconnaissez facilement, puis recompilez. Vous pourriez découvrir que ces seules corrections annulent la quasi-totalité des erreurs rapportées.

Exécution

Ouvrez une fenêtre MS-DOS, placez-vous sur le répertoire contenant le programme **hello.exe**, puis appelez-le ainsi :

```
hello
```

L'environnement d'exécution ou le CLR va alors compiler de nouveau l'IL en code natif avant de l'exécuter. Au bout de une ou deux secondes, vous devriez alors avoir la sortie :

```
Hello C#
```

Si vous appelez encore une fois le programme, vous obtiendrez de nouveau cette sortie, mais cette fois instantanément, car le code sera déjà en natif et ne devra pas subir une phase de précompilation. Si le code IL est portable, le code natif lui ne peut être utilisé que sur les systèmes où il aura été exécuté au moins une fois.

INFO

Si vous exécutez le programme en double-cliquant sur l'icône du fichier `hello.exe` dans l'Explorateur Windows, vous ne verrez rien. En effet, le programme va ouvrir une fenêtre MS-DOS, exécuter `hello.exe`, puis fermer la fenêtre, le tout à la vitesse de la lumière, et vous aurez à peine le temps d'apercevoir quelque chose.

Types de programmes

Comme nous l'avons déjà signalé dans l'introduction, vous pouvez programmer toutes sortes de choses avec C#, et en particulier :

- des applications de ligne de commande comme celles que vous allez créer tout au long de cet ouvrage ;
- des applications Windows avec lesquelles vous allez exploiter l'interface graphique fournie par Microsoft Windows ;
- des services Web utilisables sur Internet ;
- des applications ASP.NET qui génèrent dynamiquement des pages Web sur un serveur.

Analyse du programme

C# étant un langage à 99 % orienté objet, notre premier programme comprend évidemment une classe. Le mot clé **class** en première ligne annonce au compilateur que nous allons créer une nouvelle classe. Il est suivi du nom de la classe qui est ici **hello**. La création d'une classe se fait en deux étapes : la déclaration, comme nous venons de le voir, suivie de la définition. Ces deux étapes sont toujours simultanées.

Code 1.2 : Premier programme *hello.cs*

```
class hello
{

  public static void Main()
  {
    System.Console.WriteLine("Hello C#");
  }

}
```

En fait, **Console** est aussi une classe, mais vous n'avez pas besoin de la déclarer ni de la définir car elle fait partie des multiples classes fournies par le .NET Framework.

Définition d'une classe

La définition d'une classe se fait à l'intérieur de ce qu'on appelle un *bloc*. Un bloc en C# est une zone de code délimitée par une accolade ouvrante et une accolade fermante. A l'intérieur de ce bloc, nous définirons les *membres* de cette classe. En effet, une fois qu'ils ont été identifiés, les objets ne seraient pas d'une grande utilité s'ils ne possédaient pas des caractéristiques (bruyant, paresseux, agressif, etc.) et un comportement (mord, aboie, chante, vole, etc.).

Les caractéristiques sont enregistrées sous la forme de *membres de classe* (données membres ou variables membres). Elles permettent d'affecter des valeurs à nos objets pour mieux les définir. Pour chacun des animaux de notre exemple de gestion du zoo, nous pourrions créer les données membres : **nom**, **age**,

`couleur`, `taille`. Pour certains d'entre eux, nous pourrions également définir les caractéristiques `longueurQueue`, `tailleBec`, `nombreMamelles`, etc. La classe `hello` n'a pas de donnée membre, uniquement une *méthode* qui se nomme `Main` :

Code 1.3 : Premier programme *hello.cs*

```
class hello
{

  public static void Main()
  {
    System.Console.WriteLine("Hello C#");
  }

}
```

INFO

Codeurs Java : contrairement au langage Java, un fichier source n'a pas besoin de porter le même nom que la classe principale qui y est définie.

Tout comme les classes, les méthodes sont créées en deux étapes. Vous saisissez l'en-tête de la fonction, puis son contenu (corps).

Déclaration d'une méthode ou fonction

Les *méthodes*, ou *fonctions membres*, définissent un comportement (les actions que l'objet peut réaliser). Elles vont renvoyer des valeurs ou modifier l'environnement. La première catégorie de méthode est celle des accesseurs. Elles renvoient la valeur d'une des variables d'un objet. A chaque objet (animal) de notre gestion de zoo, nous pourrions associer un accesseur `LireAge()` qui renverrait l'âge d'un objet `Animal` permettant à un objet extérieur d'accéder à cette valeur. La seconde catégorie est celle des mutateurs. Ce sont des méthodes qui permettent de changer la valeur de la donnée membre d'un objet. De nouveau, nous pourrions créer la méthode `DefinirAge()` qui permettrait à un objet extérieur de modifier l'âge d'un animal.

Les paramètres que prend la méthode sont placés entre des parenthèses ouvrantes et fermantes (nous aurons l'occasion de revenir sur les arguments de fonction). Ici **Main** ne prend donc aucune valeur en entrée. Le mot clé **void** qui précède le nom de la fonction indique que celle-ci ne renverra aucune valeur. Les mots **public** et **static** décrivent les caractéristiques de notre méthode. **public** indique que notre méthode peut être utilisée par n'importe quelle autre classe. Le mot clé **static** signale que tous les objets instances de notre classe utiliseront la même méthode et non une version qui leur est propre. Cette notion sera développée au chapitre consacré au polymorphisme.

Main a un rôle particulier. Un programme peut contenir un nombre illimité de classes qui peuvent elles-mêmes contenir un nombre illimité de membres, mais il devra obligatoirement y avoir une méthode dont le nom sera **Main**. C'est par les instructions de cette méthode que débutera en effet notre programme. Ce dernier a toujours besoin d'un point de départ pour débuter son traitement. Vous comprenez mieux maintenant le message d'erreur du compilateur "Program 'hello.exe' does not have an entry point defined" lorsqu'il ne retrouve pas (ou qu'il ne reconnaît pas) cette fonction.

La fonction **Main** contient généralement la déclaration des différents objets qui seront les principaux acteurs de notre programme, mais aussi l'appel de leurs méthodes qui vont permettre de les définir ou de les modifier.

NOUVEAU

Main ne fait pas vraiment partie de la classe `hello` qui la contient, mais C# étant pratiquement à 100 % orienté objet, il n'est pas possible de créer la fonction en dehors d'une classe, comme cela est le cas en C++. Le seul objectif de la classe `hello` dans cet exemple est donc de contenir la méthode qui lancera le programme. Dans tous les autres exemples de code de cet ouvrage, cette classe s'appellera `monApplication`.

Examinons maintenant l'unique instruction de **Main** :

```
System.Console.WriteLine("Hello C#");
```

Le corps de cette méthode est un appel à une autre méthode, `WriteLine`, à laquelle on transmet un argument : la chaîne de caractères `"Hello C#"`. `WriteLine` affiche la chaîne sur la sortie standard en la faisant suivre d'un saut de ligne (console MS-DOS pour Windows, terminal xterm pour UNIX et compatible, etc.). Cette chaîne de caractères est reconnue par le compilateur grâce aux guillemets qui l'annoncent. Les mots clés `System` et `Console` précédant `WriteLine` permettent au compilateur de retrouver la bonne définition de cette fonction. Ils indiquent qu'elle fait partie de la classe `Console` de l'espace de noms `System`. Le compilateur devra donc connaître la classe `Console` pour pouvoir compiler notre programme sans générer d'erreurs. `System` peut être considéré comme un chemin d'accès à cette classe.

Les espaces de noms

Un espace de noms (namespace) représente une famille de fonctions. Le .NET Framework fournit une quantité substantielle de classes que vous allez utiliser dans vos programmes, et toutes ces classes sont regroupées par espaces de noms. Vous pouvez aussi employer des classes de sociétés tierces.

L'espace de noms `System` contient la plupart des classes dont les objets auront des actions sur le système d'exploitation sur lequel est installé l'environnement d'exécution.

En C# comme en C++, vous avez la possibilité de spécifier une fois pour toutes où se trouvent les classes dont vous allez vous servir dans le programme, en utilisant la directive `using`.

Code 1.4 : Premier programme *hello.cs*

```
// Vous spécifiez l'espace de noms utilisé :
using System;

class hello
{

  public static void Main()
  {
    Console.WriteLine("Hello C#");
  }

}
```

Grâce à la directive `using`, le compilateur saura où puiser les informations dont il aura besoin lorsqu'il rencontrera le nom d'une classe qu'il ne connaît pas, et vous n'avez plus à qualifier complètement le nom de cette fonction ou classe : `Console.WriteLine` remplace `System.Console.WriteLine`.

INFO

Certains espaces de noms contiennent d'autres espaces de noms. System, par exemple, comprend les espaces Drawing et Data. Si vous ne voulez pas faire précéder les fonctions de l'espace de noms auquel elles appartiennent, vous pouvez indiquer plusieurs instructions using :

```
using System;
using System.Drawing;
using System.Data;
```

Les commentaires

Il ne faut pas négliger l'intérêt des commentaires. C'est un élément primordial du programme, surtout pendant la phase de correction. Ils permettent en outre d'élaborer des programmes clairs et de donner une chance à d'autres programmeurs de comprendre votre travail. Enfin, ils ne pénalisent pas les performances d'exécution du programme puisqu'ils sont supprimés par le compilateur.

Les commentaires peuvent être insérés en n'importe quel endroit d'un fichier source où les caractères d'espacement sont autorisés, sauf dans les chaînes de caractères. Comme en C ou en C++, vous choisissez la syntaxe `//` pour que le compilateur ignore tous les caractères suivants jusqu'à la fin de la ligne, ou vous délimitez vos commentaires par les symboles `/*` et `*/`. Lors de la compilation, les commentaires seront évidemment enlevés du fichier IL généré.

```
//voilà un commentaire sur une ligne..
   bloc de code C#
/* Ce commentaire peut occuper plusieurs lignes
   le compilateur va ignorer tout ce qui se situe
   entre les symboles de début et de fin */
//La ligne qui suit déclenchera une erreur de compilation:
/** Vous n'avez pas le droit d'imbriquer */ ce type de
commentaires*/
/*par contre celui-là est // autorisé*/
```

NOUVEAU

En C#, il existe un troisième type de commentaire pour créer automatiquement une documentation externe sous forme de document XML. Vous reconnaîtrez ces commentaires par les trois slashs qui les précèdent.

Transformez le programme `hello.cs` de la façon suivante :

Code 1.5 : Premier programme *hello.cs* avec documentation XML

```
///<summary>
///Brève description de la classe hello.</summary>
///<remarks>
///Vous pouvez décrire plus longuement la classe hello
///avec ce type de commentaire.</remarks>
class hello
{

///<summary>
///Point d'entrée de l'application.</summary>
///<param name="args">une liste d'arguments de ligne de
    ➥commande</param>
  public static void Main(string[] args)
  {
    System.Console.WriteLine("Hello C# avec documentation
    ➥XML");
  }

}
```

Compilez ce programme en saisissant la commande suivante :

```
csc /doc:fichierXML hello.cs
```

Cette commande demande au compilateur de générer le fichier IL `hello.exe`, mais aussi d'enregistrer le fichier XML de commentaires dans le fichier nommé `fichierXML`. Ce fichier comportera le code suivant :

Code 1.6 : Fichier XML de commentaires

```
<?xml version="1.0" ?>
<doc>
   <assembly>
     <name>Hello</name>
   </assembly>
```

```
    <members>
      <member name="T:hello">
        <summary>Brève description de la classe hello.
        ➥</summary>
        <remarks>Vous pouvez décrire plus longuement la
        ➥classe hello avec ce type de commentaire.</remarks>
      </member>
      <member name="M:hello.Main(System.String[])">
        <summary>Point d'entrée de l'application.</summary>
       <param name="args">une liste d'arguments de ligne de
        ➥commande</param>
      </member>
    </members>
</doc>
```

A titre d'information, voici les balises XML spécifiquement utilisées avec C# :

```
<c>         <code>    <example>    <exception>    <list>
<para>      <param>   <paramref>   <permission>   <remarks>
<returns>   <see>     <seealso>    <summary>      <value>
```

Le compilateur C# contrôle la syntaxe des commentaires et génère des erreurs s'il détecte une balise XML ou des références erronées.

Voici les différentes catégories de commentaires disponibles avec ces balises XML. A ce stade de votre étude, il est normal que vous ne compreniez pas encore le contenu des exemples de code suivants si vous débutez en programmation. Vous pourrez consulter de nouveau cette section par la suite :

1. Description d'un élément :

   ```
   ///<summary>Description d'un élément</summary>
   ```

   Vous pouvez ajouter des paragraphes à cette description avec la balise `<para>` et faire référence à d'autres éléments *via* la balise `<see>` :

   ```
   <para> utilise une variable privée
   <see cref="nFactorial"/>
   </para>
   ```

2. Ajout de remarques ou de listes :

   La balise `<remarks>` permet d'insérer des commentaires plus détaillés qu'avec la balise `<summary>` décrite précédemment.

Vous pouvez introduire dans ces commentaires des listes à puce et des listes numérotées :

```
///<list type="bullet">
///   <item>Constructeur:
///      <see cref="Factorial"/>
///   </item>
///</list>
```

La balise <paramref> permet de décrire le paramètre transmis :

```
///<remarks>
/// <paramref name="nFactorialToComp"/>dans la variable
privée
///<see cref="nFactorialToComp"/>
///</remarks>
```

3. Description des paramètres avec la balise <param> :

```
///<param name="nFactorialToComp">
///.........
////</param>
```

Vous pouvez insérer une balise <para> dans ce type de commentaire.

4. Propriétés :

Avec la balise <value> qui se comporte comme une balise <summary>, vous décrivez une propriété.

5. Exemple :

La balise <example> permet d'afficher dans le document XML des extraits de code de votre application. Ces extraits sont encadrés de balises <code>...</code>.

```
///<example> Voici la fonction principale du programme
///<code>
///public static void Main(string[] srgs)
////.......
////</code>
///</example>
```

Le fichier XML généré présente les identifications suivantes :

```
N---Namespace (espace de noms)
T---Type, c'est-à-dire Class,Interface,struct,enum ou
delegates
F---Champs de classe
P---Indexeur de propriété ou propriété indexée
M---Méthodes spéciales constructeur ou opérateur
(surchargé)
E---Evénements
```

Types de données et objets

Même si C# est un langage orienté objet, vos programmes ne manipuleront pas uniquement des objets. Ce chapitre présente d'autres éléments nommés *primitives*, qui permettent de représenter des valeurs simples dont le programme a besoin (l'âge du capitaine, des numéros, des quantités, des chaînes de caractères, etc.). Elles ne sont pas créées dans la même zone de mémoire que les objets pour obtenir de meilleures performances. Avec ces primitives, vous créez des données en mémoire et vous effectuez des opérations sur elles très rapidement.

Ce chapitre présente aussi les variables qui vont permettre au programme de manipuler les primitives et les objets. Elles font partie des éléments de base d'un programme, et pour les utiliser correctement, vous devez savoir comment elles sont stockées en mémoire et choisir leur type de façon appropriée. Les variables ont aussi un rôle important au sein des objets puisqu'elles permettent de définir leurs caractéristiques.

Stockage des informations

Vous savez que tout ce que contient votre ordinateur est enregistré sous la forme d'une suite de zéros et de un nommés bits. L'ordinateur ne manipule pas ces bits individuellement mais par tranches de format fixe qu'on appelle *mots*.

La taille d'un mot varie d'un ordinateur à l'autre. Sur les plus anciens, elle était de 8 ou 16 bits (soit 1 ou 2 octets), mais on arrive aujourd'hui à 32 bits (soit 4 octets). Pour retrouver une information en mémoire, il suffit de connaître son adresse et sa longueur. En effet, chaque *cellule* de mémoire dans laquelle un bit est stocké possède un numéro, c'est-à-dire une *adresse*. La longueur (le nombre de bits) de l'information est déterminée par son *type*. Nous reviendrons plus loin sur les types.

Prenons l'exemple des gradins d'un stade de football. Chaque siège (cellule) va recevoir un spectateur (bit), et tous les sièges sont numérotés. Lorsque la famille Tartempion (la donnée) prend ses billets (déclaration), le caissier (ordinateur) recherche un groupe de sièges libres (emplacement libre en mémoire) et les leur attribue. Si les billets étaient gérés comme la mémoire de l'ordinateur, la famille recevrait un seul billet avec le numéro (adresse) du premier siège et le nombre de sièges réservés (taille de la donnée).

Quand vous avez besoin de manipuler une valeur dans un programme, vous créez une variable ou une constante. Comme son nom l'indique, la variable sera initialisée puis recevra différentes valeurs selon les besoins du programme. On peut définir au contraire une constante comme une variable de valeur fixe. Le nom de la variable sera l'adresse de la donnée stockée ; une variable est donc un emplacement de stockage nommé pour une donnée. Lorsque vous indiquez le nom de la variable, vous faites référence à son contenu. Puisque vous pouvez stocker toutes sortes d'informations dans une variable, vous devez commencer par la déclarer.

Déclaration des variables

La déclaration signale au compilateur le nom de la variable et le *type* d'information qu'elle va contenir. L'ordinateur peut ainsi réserver exactement la place nécessaire en mémoire RAM. En effet, c'est dans cette partie de la mémoire que le programme va ranger toutes les données de variable.

Pour simplifier les explications, examinons dès maintenant un exemple de code :

Code 2.1 : Déclaration et définition des variables

```
using System;

class monApplication
{
   public static void Main()
   {
      //Déclaration de maVariable:
      int  maVarInt ;
      //Définition de cette variable:
      maVarInt = 5 ;
      //Affichage de la valeur de maVarInt :
       Console.WriteLine("maVariable a pour valeur: {0}",
       ➥maVarInt);
   }

}
```

`int` signale au compilateur que `maVariable` est de type entier. Le type est important : il indique au programme comment interpréter la suite de zéros et de un composant la variable, et surtout il donne la taille de cette variable. Il pourra ainsi correctement restituer un nombre avec une partie décimale (virgule flottante) ou une chaîne de caractères. En C#, les types sont soit prédéfinis, soit définis par le programmeur lui-même. Les types prédéfinis sont les types fondamentaux présentés plus loin. Les types créés par le programmeur peuvent être des pointeurs, des énumérations, des tableaux (arrays), ou bien évidemment des classes. Vous aurez l'occasion d'étudier ces divers éléments dans les prochains chapitres.

ASTUCE

Vous pouvez initialiser la variable dès sa déclaration :

```
int maVariable = 5 ;
```

Et déclarer plusieurs variables du même type sur la même ligne :

```
int maVar1=5, maVar2=7 ;
```

NOUVEAU

Si vous tentez d'utiliser une variable qui n'a pas été initialisée, le compilateur C# génère un message d'erreur et ne produit pas le fichier exécutable. Ce n'est pas le cas en C et en C++. Avec ces langages, le fichier exécutable est créé et la variable contient n'importe quoi.

Vous avez certainement remarqué la syntaxe de la fonction `WriteLine()` dans l'exemple précédent. La paire d'accolades à l'intérieur de la chaîne de caractères annonce l'inclusion d'un paramètre fourni à la suite de cette chaîne. Le chiffre 0 indique le rang du paramètre ; il faut donc inclure à cet endroit le premier paramètre fourni. Examinez l'exemple suivant dans lequel plusieurs paramètres sont transmis à la fonction `WriteLine()`.

Les constantes

Si votre programme manipule des valeurs qui ne seront pas modifiées, vous allez les déclarer comme constantes. Le compilateur pourra ainsi signaler en erreur toute tentative de modification de ces valeurs. Cela permet aussi de simplifier l'écriture du programme. En effet, il sera plus facile de saisir le nom de variable PI ou Euro à chaque fois que vous aurez besoin de ces valeurs, plutôt que 3.1416 ou 6.5595. Pour déclarer une variable comme constante, il suffit d'ajouter le mot clé `const` :

Code 2.2 : Déclaration et définition des constantes

```
using System ;

// Classe qui contient notre fonction Main

class monApplication
{

  public static void Main()
  {
    //Déclarations et initialisations des constantes :
    const byte rayon = 1;
    const float PI = 3.1416F;
```

```
    //Déclaration et calcul de la variable diametre :
       float diametre = 2*PI*rayon ;
       Console.Write("Le diamètre du cercle vaut: {0}\n",
       ⇨diametre);
       Console.WriteLine("Fin du programme");

     }

}
```

Vous obtenez l'affichage :

```
Le diamètre du cercle vaut: 6,2832
Fin du programme
```

Où sont stockées les données ?

Des explications détaillées concernant l'utilisation de la mémoire de l'ordinateur dépasseraient de loin le cadre de cet ouvrage. Il est cependant intéressant d'en connaître les grandes lignes pour mieux comprendre certaines techniques de programmation. Si vous savez comment votre programme s'exécute en mémoire, vous serez en mesure de mieux gérer l'espace disponible et d'améliorer ainsi les performances à l'exécution.

Pour travailler, le programme dispose d'une zone de mémoire qui lui est spécialement affectée, appelée *pile*. Il s'agit d'une file d'attente LIFO (*Last In First Out*, dernier entré premier sorti) dans laquelle il stocke les fonctions. La taille de cette pile évolue en fonction du nombre d'éléments qui y sont stockés, et l'élément du "dessus" est identifié par un pointeur de pile enregistré dans un des registres de l'ordinateur (les registres font partie d'une autre zone de mémoire intégrée à l'unité centrale).

Il dispose aussi du *tas* (heap en anglais), quelquefois nommé segment de mémoire nettoyé, dont la taille peut varier dynamiquement dans les limites de l'espace mémoire disponible et dans lequel il stocke les objets. C'est C# qui se charge de gérer cette mémoire, notamment par l'intermédiaire du programme de récupération de mémoire (décrit un peu plus loin), ce qui réduit considérablement le risque d'erreur.

Enfin, évidemment, les données peuvent être stockées de manière permanente sur le *disque dur* ou sur des zones de *mémoire morte*. Contrairement aux informations contenues dans la pile ou le tas, ces données persistent après l'arrêt de l'ordinateur.

Taille des variables

Nous avons vu que pour utiliser efficacement la mémoire, le compilateur va réserver pour chaque variable ou objet exactement la quantité de mémoire nécessaire en fonction de son type. Il est donc important que vous choisissiez soigneusement votre variable, non seulement en fonction de la nature même de l'information, mais aussi en fonction de son utilisation dans le programme. Si vous avez prévu d'effectuer des calculs, par exemple, vous allez opter pour des variables de type `decimal` plutôt que `float` ou `double` si vous voulez obtenir des résultats plus précis. Le Tableau 2.1 énumère les différents types simples disponibles en C#.

Les types de données de C#

Les types du Tableau 2.1 sont des types *intégrés* au langage. Ces types ont souvent des instructions spéciales dans le langage intermédiaire, de sorte qu'ils peuvent être traités efficacement par le moteur d'exécution.

Tableau 2.1 : Les types intégrés de C#

C#	Taille en octets	Description
`bool`	1	Booléen (true ou false)
`short`	2	Entier signé sur 16 bits (de –32 768 à –32 767)
`ushort`	2	Entier non signé sur 16 bits (de 0 à 65 535)
`int`	4	Entier signé sur 32 bits (–2 147 483 648 à +2 147 483 648)

Tableau 2.1 : Les types intégrés de C# (*suite*)

C#	Taille en octets	Description
`uint`	4	Entier non signé sur 32 bits (0 à 4 294 967 295)
`long`	8	Entier signé sur 64 bits (de –9 223 372 036 854 775 808 à 9 223 372 036 854 775 807)
`ulong`	8	Entier non signé sur 64 bits (de 0 à 18 446 744 073 709 551 615)
`byte`	1	Entier non signé (de 0 à 255)
`sbyte`	1	Entier signé (de –128 à +127)
`float`	4	Nombre en virgule flottante (de $1,5\times10^{-45}$ à $3,4\ 10^{38}$) simple précision
`double`	8	Nombre en virgule flottante (de 5×10^{-324} à $1,7\ 10^{308}$) double précision
`char`	2	Nombre stocké sur 2 octets représentant un caractère Unicode (voir ci-après)
`decimal`	16	Un nombre de 28 chiffres au maximum (de 1×10^{-28} à $7,9\times10^{28}$)
`string`		Chaîne de caractères
`Object`		Type de tous les objets (voir Chapitre 6)

Le type **`string`** est un peu particulier. Lorsque vous stockez une chaîne de caractères dans une variable de ce type, vous stockez en réalité chaque caractère de la chaîne dans un élément de tableau (dont le nom est celui de la variable). Les tableaux sont détaillés au Chapitre 5, consultez-les pour comprendre la syntaxe qui suit :

```
//Définition de la variable maChaine de type string
string maChaine = "Il fait beau";
//maChaine étant un tableau de caractères,
//vous avez accès à chacun d'eux:
```

```
Console.WriteLine(maChaine[1]);  //affiche la lettre "l" du
➥mot "Il"
Console.WriteLine(maChaine);  //affiche la chaîne complète
maChaine[3]='F';  //erreur! vous ne pouvez pas modifier une
➥variable string
```

INFO

Le type `string` est en réalité un autre nom pour la classe `String` définie dans l'espace de noms `System`. Si vous avez besoin de manipuler des chaînes, cette classe met à votre disposition un certain nombre de fonctions dont vous trouverez la liste dans la documentation du .NET Framework.

INFO

Si vous avez besoin de modifier une chaîne de caractères dans votre programme, vous devrez plutôt utiliser la classe `StringBuilder`.

La deuxième version de notre programme présente la syntaxe des diverses affectations de variables. N'oubliez pas le point-virgule à la fin de chaque instruction.

Code 2.3 : Syntaxe des diverses affectations de variables

```
using System;

class monApplication
{
   public static void Main()
   {
      //Déclarations :
      int  maVarInt=147000 ;
      bool maVarBool=true ;
      byte maVarByte=241 ;
      short maVarShort=-30000 ;
      long maVarLong=635987456 ;
      float maVarFloat=26.4F ;
      double maVarDouble=12548 ;
      string maVarString="une chaîne de caractères" ;
      char maVarChar='m' ;
      decimal maVarDecimal=235.5m ;
```

```
    //Affichage de la valeur des variables :
          Console.WriteLine("Variable entière: {0}\n
                             Variable booléenne: {1}\n
                             Variable byte: {2}\n
                             Variable short: {3}\n
                             Variable long: {4}\n
                             Variable virgule flottante: {5}\n
                             Variable double: {6}\n
                             Chaîne de caractères: {7}\n
                             Variable char: {8}\n
                             Variable décimale: {9}\n",
          maVarInt, maVarBool, maVarByte, maVarShort,
          ➥maVarLong, maVarFloat,
          maVarDouble, maVarString, maVarChar, maVarDecimal);
      }

}
```

Vous devriez obtenir ce résultat :

```
Variable entière: 147000
Variable booléenne: True
Variable byte: 241
Variable short: -30000
Variable long: 635987456
Variable virgule flottante: 26,4
Variable double: 12548
Chaîne de caractères: une chaîne de caractères
Variable char: m
Variable décimale: 235,5
```

ATTENTION

Pour initialiser chaque variable dans ce programme, nous avons utilisé des valeurs littérales numériques ou caractères. Quel que soit le type de la variable à laquelle elle est affectée, la valeur littérale possède un type par défaut. Les littérales numériques entières seront de type `int`, `uint`, `long` ou `ulong` selon leur valeur, et les littérales numériques avec partie décimale seront de type `double`. Par exemple, `10` sera de type `int` et `10.0` sera de type `double`. Pour faire correspondre le type de la littérale avec celui de la variable "conteneur", ajoutez une lettre à la fin de la valeur : `u` pour `uint`, `l` pour `long`, `ul` pour `ulong`, `f` pour `float`, et `m` pour `decimal`.

```
float maVarFloat = 26.4f ;
decimal maVarDecimal=235.5m ;
```

Codes de mise en forme

Les variables de type **char** permettent de stocker un caractère ordinaire ou un des caractères de contrôle du Tableau 2.2. Le code de retour à la ligne **\n** apparaît à maintes reprises dans l'exemple de code précédent.

Tableau 2.2 : Les codes de mise en forme

Code	Description
\n	Retour à la ligne
\t	Tabulation horizontale
\b	Retour arrière
\"	Guillemet double
\'	Guillemet simple
\\	Barre oblique inverse

Les trois premiers permettent de mettre en forme le texte affiché à l'écran en sortie du programme, alors que les trois derniers permettent d'inclure des guillemets simples ou doubles ou une barre oblique inverse dans une chaîne de caractères. Pour illustrer ces codes, nous avons modifié notre premier programme **hello.cs**.

Code 2.4 : Programme *hello.cs* du Chapitre 1 revu et corrigé

```
using System ;
class monApplication
{

  public static void Main()
  {
    Console.WriteLine("Ce programme vous dit:\n\n\t\'Hello
    ➥C#\'");
    Console.Write("\nEt affiche 2 autres caractères
    ➥normalement interprétés par C#: ");
    Console.Write("\" et \\");
  }

}
```

Son exécution devrait produire l'affichage suivant :

```
Ce programme vous dit:

            'Hello C#\'
Et affiche 2 autres caractères normalement interprétés par
⇨C#: " et \
```

Notez l'emploi de la fonction `Write` en tous points identique à `WriteLine`, à l'exception du fait qu'elle n'introduit pas de retour à la ligne à la fin de l'instruction.

Les types de données .NET

L'interopérabilité des langages étant l'un des objectifs de la plate-forme .NET, la représentation des types de données est un problème crucial à résoudre. La Figure 2.1 présente les types du .NET Framework tels qu'ils sont définis par le système de types du CLR (runtime). Vous constatez que ce système est divisé en deux sous-systèmes : les types de valeurs et les types de références.

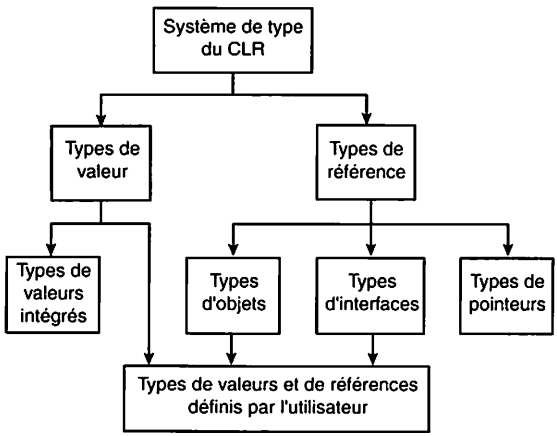


Figure 2.1
Système de types du CLR (*runtime*).

Types de valeurs

Les primitives décrites dans la première partie de ce chapitre sont des *types de valeurs*. Elles n'ont aucun concept d'identité, il s'agit juste d'une séquence de bits en mémoire. Vous pouvez facilement les comparer : par exemple, deux entiers sont égaux s'ils contiennent la même valeur (séquence de bits). Les structures, présentées au Chapitre 5, font aussi partie des types de valeurs. Il s'agit même d'un type de valeur défini par l'utilisateur.

Types de références

Les types de références sont la combinaison d'un emplacement (généralement appelé *identité*) et d'une séquence de bits. Un emplacement désigne un endroit de la mémoire dans lequel il est possible de stocker des valeurs d'un certain type. Les classes font partie des types de références. Si vous comparez deux objets de la même classe, même s'ils possèdent les mêmes valeurs de membres, ils ne sont pas considérés comme égaux parce qu'ils se réfèrent à deux objets différents. Contrairement aux types de valeurs, les types de références ne sont pas stockés dans la pile mais sur le segment de mémoire nettoyé, ou tas.

INFO

`String` et `Object` sont les deux seuls types de références intégrés (voir Tableau 2.1).

Valeur *versus* référence

Une variable d'un type de valeur stocke la valeur réelle de la donnée à l'emplacement désigné par son nom, et cette variable est allouée directement dans la pile. Les manipulations de ce type de variable sont donc particulièrement efficaces, mais leur taille doit rester raisonnable pour ne pas dépasser les capacités de la pile.

Une variable d'un type de référence ne contient pas la valeur réelle de la donnée mais l'adresse à laquelle cette valeur est enregistrée en mémoire. Elle est allouée dans le segment de mémoire nettoyé avec un accès moins performant que dans la pile, mais la contrainte de taille ne s'impose plus.

En fait, une variable d'un type de référence impose une surcharge de traitement puisque l'accès à la valeur n'est pas direct. Cette surcharge ne se justifie que si la taille de la variable est conséquente.

Complément sur le CLR

Chaque type de données C#, comme tous les types de données des langages de programmation supportés par la plate-forme .NET, possède un équivalent .NET dans l'environnement d'exécution CLR. Vous pouvez utiliser ces types dans vos programmes, mais il est beaucoup plus facile de saisir `int`, par exemple, que `System.Int32`. Le Tableau 2.3 présente ces types.

Tableau 2.3 : Equivalence des types C# et .NET

C#	.NET
`bool`	System.Boolean
`short`	System.Int16
`ushort`	System.UInt16
`int`	System.Int32
`uint`	System.UInt32
`long`	System.Int64
`ulong`	System.UInt64
`byte`	System.Byte
`sbyte`	System.SByte
`float`	System.Single
`double`	System.Double
`string`	System.String
`char`	System.Char
`decimal`	System.Decimal

Classes et objets

Le concept des classes en C# fournit aux programmeurs la possibilité de créer de nouveaux types de données, aussi faciles d'utilisation que les types intégrés `int`, `float`, `char`, etc. Si vous créez un programme de gestion des petits amis de votre sœur, par exemple, vous allez gérer les objets de type `Ex`, `petitAmiCourant`, `candidatsPotentiels`, et ainsi de suite, plutôt que des objets de type traditionnels tels que `int` (entier), `string` (chaîne de caractères), etc.

Une classe est constituée d'un ensemble de membres qui vont définir les caractéristiques de chaque objet de ce type et d'un ensemble de fonctions pour consulter ou agir sur la valeur de ces caractéristiques, ou pour définir un comportement. Un objet de type `petitAmiCourant`, par exemple, aura une fonction membre `Embrasser()`. On dit que la classe permet d'"encapsuler" les données et les fonctions, c'est-à-dire de regrouper les caractéristiques d'un objet avec son comportement. Un programme qui manipule des types très proches des éléments de l'environnement du problème à résoudre sera non seulement plus concis et plus facile à comprendre et à maintenir, mais aussi plus fiable dans la mesure où toute utilisation illégale d'un objet sera détectée dès la compilation. Si vous programmez `Ex.Embrasser()` (vous verrez plus loin que l'on fait ainsi référence à la fonction `Embrasser()` de l'objet `Ex`), le compilateur va aussitôt signaler l'erreur, puisqu'un objet de ce type ne possède pas une telle fonction !

Nous allons maintenant approfondir les notions présentées rapidement au Chapitre 1 et examiner en particulier comment développer une classe en définissant des membres de classe, comment créer une instance de classe (c'est-à-dire un objet), et comment manipuler des objets. Nous allons travailler sur un programme composé au départ uniquement d'éléments déjà étudiés, puis nous le développerons à mesure que de nouveaux concepts seront introduits.

Création d'un objet

Une classe peut être assimilée à un moule à partir duquel vous pouvez générer des *instances de classe*, c'est-à-dire des *objets*. Pour créer une instance de classe (un objet), le programme a besoin d'une fonction C++ spécifique, nommée *constructeur*, qui ne renvoie aucune valeur mais peut se charger d'initialiser les variables membres. Si vous ne la créez pas explicitement, le compilateur s'en charge.

INFO

En programmation orientée objet, la création d'une instance de classe, c'est-à-dire d'un objet, se nomme *instanciation*.

Saisissez, puis compilez l'exemple de code suivant :

Code 3.1 : Création et manipulation d'objets

```
// Utiliser l'espace de noms System:
using System;

/*
 * Classe qui contient notre fonction Main
 */
   class monApplication
   {

   //méthode principale de notre programme
   public static void Main()
      {
      Console.WriteLine("Nous n'avons encore rien créé");
      }
   }
```

```
/**
 * classe qui permet la création d'un objet de type
ExPetitAmi
 */
   class ExPetitAmi
   {
      //ExPetitAmi ne dispose pour l'instant d'aucun membre
   }
```

Rien de nouveau dans la syntaxe de ce programme. Par contre, une seconde classe est créée. La première, analogue à celle des exemples précédents, contient la méthode `Main` qui va lancer le programme. La seconde classe va nous permettre de créer des objets de type `ExPetitAmi`. A l'exécution, le programme ne va rien faire puisque notre classe `monApplication` se contente juste d'afficher la chaîne de caractères "Nous n'avons encore rien créé". Pour créer un objet, il faut introduire un constructeur dans la seconde classe :

Code 3.2 : Classe permettant la création d'un objet de type *ExPetitAmi*

```
class ExPetitAmi
{

   //constructeur
   ExPetitAmi()
   {
       //la méthode ne fait rien pour l'instant
   }
}
```

Le constructeur `ExPetitAmi()` respecte toutes les conventions d'une fonction de ce type : la méthode porte le nom de la classe et se termine par des parenthèses. Rajoutons maintenant à la méthode `Main` de la classe `monApplication` l'instruction qui va déclencher la création d'un objet de type `ExPetitAmi`.

Code 3.3 : Classe contenant notre fonction *Main*

```
class monApplication
{

   //méthode principale de notre programme
   public static void Main()
```

```
    {
        Console.WriteLine("Nous n'avons encore rien créé");
        //création d'un objet ExPetitAmi
        new ExPetitAmi();
    }
}
```

Pour que ce programme puisse être compilé sans erreur, il reste un dernier détail à régler : le contrôle de l'accès à la méthode `ExPetitAmi()`.

Portée des variables et objets

Les variables et constantes déclarées dans des méthodes ne sont visibles que dans certaines parties du programme, et cette zone dans laquelle la variable est reconnue détermine sa portée (scope en anglais). La portée d'une variable va de l'endroit où elle est créée jusqu'à la fin du bloc dans lequel elle se trouve. Si vous faites référence à la variable dans un autre bloc de code, le compilateur va générer un message d'erreur. Cette notion de portée est illustrée avec la variable `visibilite` créée dans la section "Fonctions membres" de ce chapitre.

Pour ce qui concerne les objets, la notion de portée est différente. La vie d'un objet dure de l'instant de sa création à l'aide d'un constructeur jusqu'au moment de sa destruction par le programme de récupération de mémoire. Les membres ont la même durée de vie que leur objet, mais vous pouvez contrôler l'accès à ces membres *via* les mots clés `public`, `private` et `protected`.

public, private ou *protected*

Les mots clés `public`, `private` ou `protected` déterminent la portée des membres d'une classe.

- `public`. Les membres déclarés avec le mot clé `public` peuvent être utilisés dans n'importe quelle classe (et par la classe `monApplication` en particulier). Le terme `public` permet donc de rendre visibles certaines fonctionnalités de l'objet et de les utiliser.

- `private`. Par défaut, tous les membres et méthodes d'une classe sont privés. Leur accès est autorisé uniquement pour les fonctions de la même classe. Vous devez donc créer des fonctions publiques dans la classe, appelées *propriétés*, pour consulter ou modifier ces membres. Les propriétés sont présentées dans la dernière section de ce chapitre.
- `protected`. L'accès est autorisé uniquement pour les fonctions de la même classe et de ses descendantes. Le mot clé `protected` n'est utilisé que dans le cadre de l'héritage des classes (l'héritage est traité au Chapitre 6).

Nous allons donc ajouter `public` devant la méthode `ExPetitAmi()`, et pour vérifier qu'un objet a bien été créé, nous allons faire afficher un message par le constructeur :

```
//constructeur
public ExPetitAmi()
     {
      Console.WriteLine("Un objet de type ExPetitAmi vient
     ➥d'être créé !");
     }
```

Si vous compilez puis si vous exécutez le programme à ce stade, vous allez obtenir l'affichage :

```
Nous n'avons encore rien créé
Un objet de type ExPetitAmi vient d'être créé !
```

Un objet est créé quelque part en mémoire, mais il est parfaitement inutile puisque nous n'avons aucun moyen d'y faire référence pour le manipuler. Il faut tout simplement lui attribuer un nom (*handle* en C#).

INFO

Le handle de C# est équivalent au pointeur du C++. Il est ainsi courant de dire qu'un handle pointe sur un objet en mémoire.

En C#, il peut y avoir plusieurs handles par objet, mais un handle ne peut pas référencer plusieurs objets. Enfin, un handle qui correspond à un objet à un certain moment du programme peut correspondre à un autre objet à un autre moment.

Nom de l'objet

Pour être en mesure de manipuler l'objet, nous allons transformer la ligne

```
new ExPetitAmi();
```

dans la méthode **Main** par les lignes

```
ExPetitAmi Maurice;
Maurice = new ExPetitAmi();
```

que nous pouvons résumer en C# en :

```
ExPetitAmi Maurice = new ExPetitAmi();
```

Nous pouvons maintenant préciser quel "*objet de type* ExPetitAmi *vient d'être créé*" :

Code 3.4 : Classe contenant notre fonction *Main*

```
class monApplication
{

   //méthode principale de notre programme
   public static void Main()
   {
      Console.WriteLine("Nous n'avons encore rien créé");
      Console.WriteLine("Création de l'objet Maurice :");
      ExPetitAmi Maurice = new ExPetitAmi();
   }
}
```

ATTENTION

Si vous codez :

```
ExPetitAmi Maurice;
new ExPetitAmi();
```

il n'y a aucun lien entre le handle de la première ligne et l'objet créé sur la seconde. Vous perdez donc l'objet dans la mémoire et vous restez avec un handle sans objet associé.

Comme l'indique la sortie du programme :

```
Nous n'avons encore rien créé
Création de l'objet Maurice :
Un objet de type ExPetitAmi vient d'être créé!
```

nous pouvons maintenant manipuler l'objet `Maurice`, mais cet objet ne sera pas d'une grande utilité s'il ne possède pas des caractéristiques et un comportement.

A ce stade, l'objet ne peut rien faire puisque la classe `ExPetitAmi` ne contient aucun membre (mis à part le constructeur). Pour lui affecter un comportement (les actions que l'objet peut réaliser), il faut définir des *méthodes* ou des *fonctions membres*. Ces dernières vont renvoyer des valeurs ou modifier l'environnement.

Fonctions membres

Nous allons donc rajouter deux méthodes à la classe, que nous nommerons `Raconter()` et `Soupirer()`, qui vont permettre aux instances de `ExPetitAmi` de pouvoir s'exprimer après leur création.

Code 3.5 : Classe permettant la création d'un objet de type *ExPetitAmi*

```
 */
class ExPetitAmi
{
   //constructeur
   public ExPetitAmi()
   {
      Console.WriteLine("Un objet de type ExPetitAmi vient
      ➥d'être créé!");
   }

   // Fonction membre Raconter()
   public void Raconter()
  {
    //Première variable visibilite:
    string visibilite = "sa vie";
    Console.WriteLine("Il raconte...{0}", visibilite);
  }
   // Fonction membre Soupirer()
   public void Soupirer()
```

```
    {
        //Deuxième variable visibilite:
        string visibilite = "encore";
        Console.WriteLine("Il soupire...{0}", visibilite);
    }
}
```

Notre type de donnée `ExPetitAmi` commence maintenant à s'étoffer. Les variables `visibilite` ne présentent pas d'autre intérêt que d'illustrer la notion de portée. Il s'agit bien de deux variables différentes puisqu'elles se situent dans des blocs de code distincts. En examinant le résultat de l'exécution du programme un peu plus loin, vous constaterez qu'elles ont bien été différenciées.

Données membres

Pour mieux définir encore cet objet, nous allons lui attribuer des caractéristiques par l'intermédiaire de variables.

Code 3.6 : Classe permettant la création d'un objet de type *ExPetitAmi*

```
class ExPetitAmi
{
    //constructeur
    public ExPetitAmi()
    {
        Console.WriteLine("Un objet de type ExPetitAmi vient
        ➥d'être créé!");
    }

    // Fonction membre Raconter()
    public void Raconter()
    {
      string visibilite = "sa vie";
      Console.WriteLine("Il raconte...{0}", visibilite);
    }
     // Fonction membre Soupirer()
     public void Soupirer()
    {
      string visibilite = "encore";
      Console.WriteLine("Il soupire...{0}", visibilite);
  }
```

```
    //données membres
      public bool ex = true ;
      public string nom ;
      public byte age ;
}
```

La seule variable initialisée est la variable booléenne **ex**, puisqu'il s'agit d'une caractéristique commune à tous les objets **ExPetitAmi**. Les variables **nom** et **age** seront initialisées au moment de la création de l'objet.

Appel d'un membre

Dans la version finale du programme présentée un peu plus loin, nous allons mettre en scène deux objets dans la fonction **Main**, mais pour les faire "vivre", il faut pouvoir appeler les membres associés. En C#, pour faire référence à un membre d'un objet, il suffit d'indiquer le handle de cet objet suivi du nom du membre à utiliser, en les séparant avec l'opérateur "point". Ainsi, pour faire parler l'objet **Maurice** de type **ExPetitAmi** (ou l'instance **Maurice** de la classe **ExPetitAmi**), il suffit de coder :

```
Maurice.Raconter();
```

Nous avons employé cette notation dès le premier programme dans l'instruction :

```
System.Console.WriteLine("Hello C#");
```

Cette ligne appelle la fonction membre **WriteLine** de la classe **Console** dans l'espace de noms **System**.

De la même façon, pour faire référence à la donnée membre **public age** de l'objet **Maurice** de type **ExPetitAmi**, il suffit de coder : **Maurice.age** (voir le programme complet à la fin du chapitre).

Les membres statiques

Vous avez certainement remarqué le mot clé **static** qui accompagne chaque appel de la fonction principale **Main()**. Qu'il s'agisse d'une fonction ou d'une variable, ce mot clé change la portée de l'élément sur lequel il s'applique.

Une *fonction membre statique* peut être appelée n'importe où dans la classe dans laquelle elle est définie, et elle peut être exécutée dans la classe de la fonction `Main()` sans qu'il soit nécessaire de déclarer un objet puis d'appeler la fonction avec le nom de l'objet. La fonction `WriteLine()` est un excellent exemple de fonction statique. Si vous ne l'aviez pas déjà remarqué, à aucun moment nous n'avons créé d'objet de type `Console`, la classe à laquelle cette fonction appartient.

Les fonctions mathématiques sont des fonctions typiquement statiques puisqu'elles fournissent un service simple qui ne nécessite pas la création d'un objet. Vous pouvez les regrouper dans une classe et réaliser tous vos calculs dans la fonction `Main` sans créer le moindre objet.

Une *donnée membre statique* est une donnée qui sera partagée par toutes les instances de la classe, son existence n'est donc pas liée à un objet particulier. Tous les objets de type `ExPetitAmi`, par exemple, pourraient avoir en commun la donnée membre `sexe` (nous avons volontairement simplifié le programme) :

Code 3.7 : Version simplifiée avec une variable statique

```
using System;

// classe qui permet la création d'un objet de type
➥ExPetitAmi
class ExPetitAmi
{
   //Donnée membre statique :
   public static string sexe = "Masculin";
   //constructeur
   public ExPetitAmi() { }
}

// classe de la fonction Main()
class monApplication //aucune création d'objet!
{
   public static void Main()
   {
      Console.WriteLine("Un objet de type ExPetitAmi est de
      ➥sexe: {0}", ExPetitAmi.sexe);
   }
}
```

Vous obtenez l'affichage suivant :

```
Un objet de type ExPetitAmi est de sexe: Masculin
```

Une variable statique est accessible dans la fonction `Main()`, même si aucun objet n'a été créé, en faisant précéder son nom de celui de sa classe, puis d'un point.

En conclusion, une donnée membre statique est associée à une classe plutôt qu'à un objet spécifique.

INFO

Lorsqu'une classe ne contient que des membres statiques (fonctions et données), la création d'un objet ne présente aucun intérêt.

Le garbage collector

Pour terminer, il ne faut pas oublier un aspect important de la programmation : comment "nettoyer" la mémoire à la fin du programme pour libérer les ressources utilisées pendant son exécution. Dans certains langages, les programmeurs doivent continuellement se préoccuper de libérer la mémoire occupée par les différents objets qu'ils ont créés. Cette libération de mémoire est importante pour le bon fonctionnement des programmes et, à un degré plus élevé, du système d'exploitation. Si un programme génère un grand nombre d'objets et ne libère pas la mémoire réservée pour ces derniers à la fin de son exécution, la mémoire risque de se retrouver encombrée d'objets parasites et inutiles. Certes, un objet ne prend qu'une place infime dans la mémoire, mais si le PC reste allumé longtemps et si le programme est exécuté souvent, alors le système sur lequel le programme est lancé finira irrémédiablement par planter. Ceci sans compter les autres programmes ne libérant pas la mémoire correctement. C# résout ce problème avec le *garbage collector*, un programme de récupération de la mémoire. Sa tâche est de libérer les ressources réservées automatiquement par C# lorsque la mémoire libre commence à manquer. Ce programme s'attaque uniquement aux objets qui ne sont plus utilisés. Il s'active également au moment où le programme se termine et va libérer toutes les ressources qu'il avait réservées.

NOUVEAU

En C#, vous n'avez pas besoin de détruire les objets après les avoir créés dans le code source, puisque la libération des ressources est automatiquement réalisée par le "garbage collector" que nous nommerons *programme de récupération de la mémoire.*

Le programme complet est le suivant :

Code 3.8 : Création et manipulation d'objets

```
using System;

   class monApplication
   {
      public static void Main()
      {
       Console.WriteLine("Création de l'objet Maurice:");
          ExPetitAmi Maurice = new ExPetitAmi();
       //Initialisation des 2 membres nom et age de
         ➥Maurice :
          Maurice.nom = "Dupont" ;
          Maurice.age = 19 ;
       Console.WriteLine("Création de l'objet Marcel:");
          ExPetitAmi Marcel = new ExPetitAmi();
          //Initialisation des 2 membres nom et age de
            ➥Marcel :
          Marcel.nom = "Serdan" ;
          Marcel.age = 20 ;
/*
 *Deux objets ExPetitAmi sont maintenant créés en mémoire
 */
       Console.WriteLine("Que fait Maurice?");
          Maurice.Raconter();
       Console.WriteLine("Il s'appelle {0}", Maurice.nom);
       Console.WriteLine("Que fait Marcel?");
          Marcel.Soupirer();
       Console.WriteLine("Il a {0} ans.", Marcel.age);
       Console.WriteLine("Fin du programme");
      }

   }

/*
```

```
 * classe qui permet la création d'un objet de type
   ⇨ExPetitAmi
 */
class ExPetitAmi
{
   //constructeur
   public ExPetitAmi()
   {
      Console.WriteLine("Un objet de type ExPetitAmi vient
      ⇨d'être créé!");
   }

   // Fonction membre Raconter()
   public void Raconter()
  {
    string visibilite = "sa vie";
    Console.WriteLine("Il raconte...{0}", visibilite);
  }
   // Fonction membre Soupirer()
   public void Soupirer()
  {
    string visibilite = "encore";
    Console.WriteLine("Il soupire...{0}", visibilite);
  }
   //données membres
   public bool ex = true ;
   public string nom ;
   public byte age ;
}
```

L'exécution du programme produit l'affichage suivant :

```
Création de l'objet Maurice:
Un objet de type ExPetitAmi vient d'être créé!
Création de l'objet Marcel:
Un objet de type ExPetitAmi vient d'être créé!
Que fait Maurice?
Il raconte...sa vie
Il s'appelle Dupont
Que fait Marcel?
Il soupire...encore
Il a 20 ans.
Fin du programme
```

Propriétés

Nous avons déjà expliqué tout l'intérêt d'encapsuler les données et les fonctions dans un objet afin d'être capable d'utiliser ce dernier quelle que soit la complexité de son "fonctionnement interne" (souvenez-vous du magnétoscope que n'importe qui est capable de faire fonctionner sans être un as de l'électronique). Pour rendre l'encapsulation encore plus efficace, vous pouvez augmenter le contrôle de l'accès aux données *via* les *propriétés* de C#.

Dans l'exemple de code précédent, les données membres pourraient toutes être définies en `private` et être accessibles en lecture seulement ou en lecture/écriture *via* les propriétés :

Code 3.9 : Classe permettant la création d'un objet de type *ExPetitAmi*

```
class ExPetitAmi
{
   //constructeur
   public ExPetitAmi()
   {
      Console.WriteLine("Un objet de type ExPetitAmi vient
      ➥d'être créé !");
   }

   //données membres privées
   string nomp ;
   bool exp = true ;
   byte agep ;

    //Accès à la variable nomp
   public string nom
   {
      //Lecture :
      get { return nomp ; }
      //Ecriture :
      set { nomp = value ;}
   }

    //Accès à la variable exp en lecture seulement
   public bool ex
   {
      get { return exp ; }
   }

  //Accès à la variable agep
```

```
public byte age
  {
     //Lecture :
     get { return agep ; }
     //Ecriture avec contrôle de la valeur :
     set
     { if (value > 5 && value <80) //si la valeur est
     ➥comprise entre 5 et 80
          agep = value;            //OK
       else                        //sinon
       throw new Exception("L'âge doit être compris entre
       ➥5 et 80");
     }
  }

}
```

Les trois données membres privées ont été rebaptisées **nomp**, **exp** et **agep**. Le reste du programme ne change pas, puisque l'accès à ces membres se fait *via* les propriétés **nom**, **ex** et **age**. Elles illustrent trois possibilités de contrôle d'accès :

- En lecture et en écriture pour le nom : l'instruction **get** qui renvoie la valeur de **nomp** est appelée à chaque fois qu'un programme tente d'obtenir la valeur de **nom**, et l'instruction **set** qui définit la valeur de **nomp** est appelée à chaque fois qu'un programme affecte une valeur à **nom**. Le mot clé **value** est une variable intermédiaire, il représente la valeur affectée à **nom**, et le type de cette valeur doit correspondre à celui de la déclaration de **nom**.
- En lecture seule pour la variable **exp** : la propriété ne contient qu'une instruction **get**.
- La propriété **age** montre que vous pouvez introduire un certain nombre d'opérations de contrôle dans les instructions **get** ou **set**. Dans ce cas, nous vérifions que la valeur est bien comprise entre 5 et 80, sinon nous déclenchons une exception (le programme s'interrompt en affichant un message d'erreur). Les exceptions sont traitées au Chapitre 7.

INFO

La syntaxe de l'instruction conditionnelle if... else est expliquée dans la section "Les opérateurs" de l'annexe.

Fonctions

Les fonctions définissent le comportement des objets et en particulier celui de l'objet application. En effet, le programme lui-même est un objet puisqu'il est défini dans une classe : celle qui contient la fonction principale `Main`.

Dans un programme bien écrit, une fonction réalise une tâche spécifique, clairement identifiée par son nom. Elle peut renvoyer une valeur, et vous pouvez lui transmettre des informations. Jusqu'à présent, vous avez surtout utilisé les fonctions `WriteLine` et `Main`, et vous avez créé quelques fonctions très simples au chapitre précédent.

Ce chapitre détaille le rôle des fonctions en programmation orientée objet. Il explique comment leur transmettre des arguments, comment récupérer le résultat de leur exécution et, point très important, comment implémenter une première forme de polymorphisme.

Le programme que nous allons développer dans ce chapitre calcule le factoriel d'un nombre. Tout ce qui apparaît en caractères gras est détaillé à la suite du programme. Pour simplifier la lecture du code, nous avons inclus de nombreux commentaires qui vous permettront de reconnaître les différentes parties de l'application.

Code 4.1 : Complément sur les fonctions

```
1: using System;
2:
3: //Classe qui contient la fonction principale Main
4: class monApplication
5: {
```

```
 6:   //En-tête de la fonction principale Main
 7:    public static void Main()
 8:    {
 9:     //Corps de la fonction principale Main
10:      Console.WriteLine("Le factoriel de 20 est {0} ",
         ➥Calcul(20));
11:    }
12:
13:    //Fonction membre de calcul du factoriel
14:    public static long Calcul(int nFactorielACalculer)
15:    {
16:      long compteur=1;
17:      long nFactoriel=1;
18:      for(compteur=1;compteur<=nFactorielACalculer;
         ➥compteur++)
19:      {
20:         nFactoriel*=compteur;
21:      }
22:    return nFactoriel;
23:    }
24:}
```

Saisissez ce programme sans les numéros de ligne et compilez-le. A ce stade, il se contente de calculer et d'afficher le factoriel du nombre 20.

Transmission des arguments et valeur renvoyée

Examinons de plus près la fonction principale `Main` puisqu'il s'agit de la première fonction exécutée par le programme. Son *en-tête* apparaît à la ligne 7 et son corps, délimité par une paire d'accolades, occupe les lignes 8 à 11.

En-tête de fonction

L'en-tête apporte un certain nombre d'informations sur la fonction :

- Les restrictions d'accès : `public`, `private` ou `protected` (voir Chapitre 3).
- Le mot clé `static`, obligatoire pour la fonction `Main()`. La classe `monApplication` a en effet ceci de particulier que nous ne créons pas d'instance de cette classe, c'est-à-dire d'objet de type `monApplication`, dans le code du programme. Cet objet est créé automatiquement en

mémoire au moment de l'exécution du programme. Par conséquent, nous ne pouvons pas utiliser la notation *nomObjetApplication*.*Main*() pour appeler cette fonction ou toute autre fonction déclarée dans la même classe. Si vous ne déclarez pas `Main()` avec le mot clé `static`, le compilateur ne trouve pas le "point d'entrée" du programme parce que la fonction n'est pas "visible" en dehors de la classe `monApplication`.

Comme nous l'avons expliqué au Chapitre 3, en déclarant `Calcul()` avec `static`, vous pouvez l'utiliser n'importe où dans la classe `monApplication` sans autre qualification, et en particulier dans la fonction `Main`. Si cette fonction avait été définie toujours en statique dans une autre classe nommée `Factoriel` par exemple, l'instruction principale de la fonction `Main` deviendrait :

```
Console.WriteLine("Le factoriel de 20 est {0} ", Factoriel
➥.Calcul(20));
```

- Le type de la valeur renvoyée : le mot clé `void` signifie que `Main()` ne renvoie pas de valeur.
- Le nom de la fonction suivi d'une paire de parenthèses.
- Pour terminer, vous pouvez transmettre des arguments à la fonction en les indiquant dans la paire de parenthèses comme dans le cas de la fonction `Calcul()` (détaillée ci-après).

Corps de la fonction

C'est dans le corps de la fonction que l'action se déroule. Celui de la fonction `Main()` est très simple puisqu'il ne contient qu'une seule instruction. Mais deux fonctions statiques y sont exécutées :

`WriteLine()`, qui est appelée avec le nom de sa classe (`Console`) parce qu'elle est déclarée et définie dans une autre classe que `monApplication`, et `Calcul()`, qui n'a pas besoin d'autre qualification puisqu'elle est définie dans la même classe que `Main()`.

La fonction `Calcul()` est un peu plus complexe :

Code 4.2 : La fonction *Calcul()*

```
public static long Calcul(int nFactorielACalculer)
{
  //Déclaration d'une variable pour le compteur de la
    ➥boucle :
  long compteur=1;
  //Déclaration d'une variable pour stocker le résultat :
  long nFactoriel=1;
  //Calcul du factoriel
  for(compteur=1;compteur<=nFactorielACalculer;compteur++)
  {
     nFactoriel*=compteur;
  }
return nFactoriel;
```

L'en-tête indique qu'il s'agit d'une fonction en accès public qui ne nécessite pas la création d'un objet (`static`), qui renvoie une valeur de type `long` et qui reçoit en argument la variable `nFactorielACalculer` de type `int`. Si vous examinez la ligne 10 du programme, vous constaterez qu'en effet cette fonction est appelée avec l'entier 20, et si vous observez le résultat de l'exécution du programme, vous verrez que la valeur renvoyée est aussi un nombre (`long`).

Le corps de la fonction contient deux déclarations de variables, puis une boucle `for` pour le calcul du factoriel que nous pouvons traduire de cette façon :

```
compteur et nFactorielACalculer étant initialisés à 1
Multiplier compteur par nFactorielACalculer
Stocker le résultat dans nFactorielACalculer
Incrémenter compteur d'une unité
Si compteur est inférieur ou égal à nFactorielACalculer, on
➥recommence en ligne 2 (la multiplication)
Sinon on continue
```

Vous trouverez la syntaxe des instructions conditionnelles (comme les boucles `for`) et des opérateurs (comme `*=`) à l'annexe.

Le mot clé `return` en dernière ligne permet de renvoyer le résultat du calcul. Le type de `nFactoriel` doit bien sûr correspondre au type (`long`) annoncé dans l'en-tête de la fonction.

Nous allons maintenant transformer le programme pour ne travailler qu'avec des objets. La première étape consiste à déplacer la fonction `Calcul()` dans une classe `Factoriel`. Le calcul du factoriel d'un nombre dans la fonction `Main()` ne se fera plus en appelant simplement une fonction statique, mais en créant un objet de type `Factoriel`, puis en appelant la fonction `Calcul()` associée. L'argument pour lequel vous voulez calculer le factoriel n'est plus transmis directement à la fonction `Calcul()`, mais au constructeur de l'objet `Factoriel` :

Code 4.3 : Classe permettant la création d'un objet de type *Factoriel*

```
public class Factoriel
{
   //Données membres :
   private long nFactoriel=1;
   private int nFactorielACalculer;

   //Constructeur d'un objet de type Factoriel
   public Factoriel(int FactorielACalculer)
   {
      this.nFactorielACalculer=FactorielACalculer;
   }

   //Propriété pour l'accès à nFactorielACalculer
   public int FactorielACalculer
   {
      get { return nFactorielACalculer; }
      set { nFactorielACalculer=value;  }
   }

   //Fonction membre de calcul du factoriel:
   public long Calcul()
   {
      long compteur=1;
      for(compteur=1;compteur<=nFactorielACalculer;
      ➥compteur++)
        {
           nFactoriel*=compteur;
        }
      //On renvoie le résultat:
      return nFactoriel;
   }
}
/* Fin de la déclaration de la classe Factoriel */
```

Nous déclarons les deux variables `nFactorielACalculer` et `nFactoriel` nécessaires au calcul du factoriel comme données membres privées de la classe `Factoriel`, pour suivre la règle de l'orienté objet qui consiste à contrôler l'accès aux données en implémentant des propriétés. `nFactoriel` étant simplement utilisée par la fonction pour stocker le résultat du calcul, il n'est pas nécessaire de lui attribuer une propriété, il suffit de l'initialiser.

La variable `nFactorielACalculer` étant définie au contraire au moment de la création de l'objet, nous déclarons la propriété `FactorielACalculer` avec les deux fonctions habituelles `get()` et `set()`, pour être en mesure d'obtenir ou de définir sa valeur.

Mot clé this

Examinons de plus près le constructeur de la classe `Factoriel` :

```
//Constructeur d'un objet de type Factoriel
public Factoriel(int FactorielACalculer)
{
   this.FactorielACalculer=FactorielACalculer;
}
```

Un constructeur n'est rien de plus qu'une fonction qui joue un rôle particulier : il permet de créer un objet. Comme toute fonction, le constructeur peut recevoir des arguments et effectuer des opérations d'initialisation pour l'objet. Il ne doit pas renvoyer de valeur.

Notre constructeur `Factoriel()` reçoit la valeur `FactorielACalculer` de type `int`, qu'il va affecter à la variable `nFactorielACalculer` *via* la propriété `FactorielACalculer` avec l'instruction :

```
this.FactorielACalculer=FactorielACalculer;
```

Le mot clé `this` fait toujours référence à l'objet en cours, dans ce cas l'objet de type `Factoriel`. Il permet donc de faire explicitement référence à la donnée membre `FactorielACalculer` de la classe `Factoriel` qui est différente de l'argument de même nom transmis au constructeur et qui apparaît à droite de l'opérateur =.

Nous pourrions transformer le constructeur de cette façon :

```
public Factoriel(int argument)
{
   FactorielACalculer=argument;
}
```

`FactorielACalculer` étant la seule variable de ce nom dans la portée, il n'est plus nécessaire de la qualifier avec `this`.

INFO

Le mot clé `this` permet d'accéder aux données membres d'une classe dans les méthodes ou fonctions de cette classe.

Nous allons maintenant transformer la classe `monApplication` pour obtenir le factoriel d'un nombre *via* un objet `Factoriel` :

```
class monApplication
{
   public static void Main()
     {
        //Création d'un objet de type Factoriel
        Factoriel monFactoriel=new Factoriel(20);
        Console.WriteLine("Le factoriel de 20 est {0} ",
                          monFactoriel.Calcul());
     }
}
```

Au moment où l'objet `monFactoriel` est créé, son constructeur est exécuté et il initialise la variable `nFactorielACalculer` avec la valeur transmise : 20. Il ne reste plus qu'à appeler la fonction `Calcul()` associée à cet objet pour obtenir le résultat.

Nous disposons maintenant d'un programme qui calcule le factoriel du nombre 20. Son intérêt est assez limité ; il serait beaucoup plus intéressant de lui faire calculer le factoriel d'un nombre transmis par l'utilisateur du programme sur la ligne de commande. La transmission d'arguments à la fonction `Main` se fait par l'intermédiaire d'un tableau de type `string`. Les tableaux sont détaillés au Chapitre 5. Retenez simplement ici que lorsque vous allez exécuter le programme en saisissant sur la ligne de commande :

```
Factoriel 5
```

Tout ce qui suit le nom du programme est stocké dans une structure de type "tableau" dont vous pouvez choisir le nom, mais que tous les programmeurs nomment **args**. Pour accéder ensuite à ces différents arguments de ligne de commande, vous utilisez la syntaxe de l'accès aux éléments d'un tableau (voir Chapitre 5) : **args[0]** pour le premier élément, **args[1]** pour le second, etc.

Voici le programme de calcul du factoriel complet :

Code 4.4 : Programme *factoriel.cs*

```
/*****************************************/
/*Ce programme reçoit en entrée l'entier */
/*dont il renvoie le factoriel           */
/*****************************************/
 1: using System;
 2:
 3:/*
 4: * classe qui permet la création d'un objet de type
      ⇨Factoriel
 5: */
 6:public class Factoriel
 7:{
 8:    //Données membres :
 9:    private long nFactoriel=1;
10:    private int nFactorielACalculer;
11:
12:    //Constructeur d'un objet de type Factoriel
13:    public Factoriel(int FactorielACalculer)
14:    {
15:       this.FactorielACalculer=FactorielACalculer;
16:    }
17:
18:    //Propriété pour l'accès à nFactorielACalculer
19:    public int FactorielACalculer
20:    {
21:       get { return nFactorielACalculer; }
22:       set { nFactorielACalculer=value;  }
23:    }
24:
25:    //Fonction membre de calcul du factoriel:
26:    public long Calcul()
27:    {
28:       long compteur=1;
```

```
29:       for(compteur=1;compteur<=nFactorielACalculer;
          ➥compteur++)
30:         {
31:            nFactoriel*=compteur;
32:         }
33:       return nFactoriel;
34:    }
35:}
36:/* Fin de la déclaration de la classe Factoriel */
37:
38:class monApplication
39:{
40:    public static void Main(string[] args)
41:      {
42:         //Réception et analyse de l'argument:
43:         int argument = int.Parse(args[0]);
44:         //Création de l'objet de type Factoriel
45:         Factoriel monFactoriel=new Factoriel(argument);
46:         Console.WriteLine("Le factoriel de {0} est {1} ",
47:                           argument, monFactoriel.Calcul());
48:      }
49:}
```

Enregistrez ce programme dans le fichier `factoriel.cs`, puis compilez-le. Saisissez ensuite sur la ligne de commande :

```
factoriel 5
```

Vous obtenez le résultat :

```
Le factoriel de 5 est 120
```

La ligne 43 mérite quelques explications. L'argument est récupéré dans le premier élément d'un tableau de type `string`, il est donc considéré par le compilateur comme une chaîne de caractères, même si vous avez saisi un nombre. Il faut donc le convertir en entier puisque la fonction `Calcul()` doit recevoir une valeur de ce type. C'est exactement ce que fait la fonction `int.Parse()` : elle convertit la représentation `string` d'un nombre en son équivalent entier. Cette fonction est fournie par les bibliothèques de classes du .NET Framework, tout comme la fonction `Console.WriteLine()`.

Transmission par valeur, par référence ou out

Dans tous les exemples de transmission d'arguments de ce chapitre, nous avons transmis des *valeurs* aux fonctions. Cette valeur était utilisée localement par la fonction, puis supprimée de la mémoire à la fin de l'exécution de cette dernière. Lorsque vous créez l'objet `Factoriel` par exemple :

```
Factoriel monFactoriel=new Factoriel(argument);
```

vous ne transmettez au constructeur de l'objet qu'une copie de la valeur de `argument`. Dans certains cas cependant, vous aurez besoin de garder une trace de l'exécution d'une fonction ou de recevoir plusieurs valeurs en retour. Vous allez alors ajouter l'attribut `ref` ou l'attribut `out` devant la déclaration d'argument dans l'en-tête de la fonction :

Code 4.5 : Transmission des arguments

```
1:using System;
2:class MaClasse
3:{
4:  private int valeur;
5:
6:  //in, out, ref parameters
7:  public int Lecture(int n, ref int reference, out
    ➥string chaineOut)
8:  {
9:     valeur = n;
10:    Console.WriteLine("La fonction Lecture change la
       ➥valeur de ses 3 arguments puis renvoie l'argument
11:    ➥transmis par valeur modifié");
12:     valeur += reference;
13:    reference += valeur;
14:     chaineOut = "chaîne de caractères transmise en out";
15:     return valeur;
16:  }
17:}
18:
19:class monApplication
20:{
21:  public static void Main()
22:  {
23:    int maValeur = 10, maRef = 20, valeurRenvoi=0;
24:    string maChaineOut="vide";
25:    Console.WriteLine("Mes 3 arguments initiaux :\nLa
       ➥valeur={0},
```

```
26:         la référence={1}, la chaîne définie en out={2}",
27:         maValeur, maRef, maChaineOut);
28:
29:     MaClasse monObjet = new MaClasse();
30:     valeurRenvoi = monObjet.Lecture(maValeur,ref
        ➥maRef,out maChaineOut);
31:     Console.WriteLine("\nAprès l'exécution de Lecture :
        ➥\n");
32:     Console.WriteLine("La valeur est toujours égale à
        ➥{0}", maValeur);
33:     Console.WriteLine("La référence devient {0}", maRef);
34:     Console.WriteLine("la chaîne transmise en out est
        ➥{0}", maChaineOut);
35:    Console.WriteLine("Enfin la fonction renvoie : {0}
       ➥via son instruction
36:                         return", valeurRenvoi);
37:  }
38:}
```

A la ligne 7, l'argument `reference` est *transmis par référence* à la fonction. Cela signifie qu'elle ne reçoit pas une copie de sa valeur, mais l'adresse à laquelle est stockée cette valeur. Par conséquent, lorsque la fonction est appelée et que l'instruction de la ligne 13 est exécutée, c'est bien la valeur de la variable `reference` qui est modifiée, comme le démontre le résultat de l'exécution du programme suivant.

Toujours à la ligne 7, l'argument `chaineOut` étant précédé du mot clé `out`, cette variable restera accessible après l'appel de la fonction. Cet attribut permet à la fonction de renvoyer des valeurs autrement que par l'unique instruction `return`. Comme la variable `reference`, la variable `chaineOut` existe toujours après l'appel de la fonction, mais contrairement à `reference`, elle n'existait pas avant.

L'exécution du programme donne le résultat suivant :

```
Mes 3 arguments initiaux :
La valeur=10, la référence=20, la chaîne définie en out=vide
La fonction Lecture change la valeur de ses 3 arguments
➥puis renvoie l'argument transmis par valeur modifié

Après l'exécution de Lecture :

La valeur est toujours égale à 10
La référence devient 50
```

```
la chaîne transmise en out est chaîne de caractères
➥transmise en out
Enfin la fonction renvoie : 30 via son instruction return
```

Polymorphisme

Le polymorphisme fait partie des concepts clés de l'orienté objet. Comme nous l'avions brièvement expliqué au Chapitre 1, une forme de polymorphisme se traduit par la possibilité d'"appeler" un objet ou une fonction de différentes façons, et d'obtenir le même résultat.

La fonction `Max()` de la classe `Math` fournie dans la bibliothèque de classes du .NET Framework est une candidate idéale pour illustrer ce concept. Cette fonction compare deux valeurs et renvoie la plus grande des deux.

Surcharge des fonctions

Si vous consultez la documentation MSDN pour savoir comment la fonction `Max` est définie, vous allez trouver la liste suivante :

Overload List (liste des surcharges)

Renvoie le plus grand des deux entiers 8 bits non signés :

```
public static byte Max(byte, byte);
```

Renvoie le plus grand des deux nombres décimaux :

```
public static decimal Max(Decimal, Decimal);
```

Renvoie le plus grand des deux nombres en virgule flottante double précision :

```
public static double Max(double, double);
```

Renvoie le plus grand des deux entiers signés 16 bits :

```
public static short Max(short, short);
```

Renvoie le plus grand des deux entiers signés 32 bits :

```
public static int Max(int, int);
```

Renvoie le plus grand des deux entiers signés 64 bits :

```
public static long Max(long, long);
```

Renvoie le plus grand des deux entiers signés 8 bits. Cette méthode n'est pas conforme avec le CLS :

```
public static sbyte Max(sbyte, sbyte);
```

Renvoie le plus grand des deux nombres en virgule flottante simple précision :

```
public static float Max(float, float);
```

Renvoie le plus grand des deux entiers non signés 16 bits. Cette méthode n'est pas conforme avec le CLS :

```
public static ushort Max(ushort, ushort);
```

Renvoie le plus grand des deux entiers non signés 32 bits. Cette méthode n'est pas conforme avec le CLS :

```
public static uint Max(uint, uint);
```

Renvoie le plus grand des deux entiers non signés 64 bits. Cette méthode n'est pas conforme avec le CLS :

```
public static ulong Max(ulong, ulong);
```

Il s'agit de la liste des *signatures* des différentes versions de la fonction `Max`.

INFO

Chaque méthode est identifiée de façon unique par sa signature, c'est-à-dire par le nombre et le type de ses paramètres.

Lorsque vous appellerez cette fonction dans votre programme, le compilateur reconnaîtra la fonction à exécuter à partir des arguments fournis. Si vous codez dans votre programme :

```
//des instructions...
var1 = Math.Max(5, 5);
//des instructions...
var2 = Math.Max(2.3, 6.1);
//des instructions...
var3 = Math.Max(-1, 5);
```

vous allez en réalité exécuter trois fonctions différentes.

Surcharge du constructeur

Comme toute fonction, le constructeur d'un objet peut être surchargé. Cela permet d'obtenir des objets de même type, mais avec un comportement différent selon les arguments transmis au moment de la création de l'objet.

Pour illustrer ce concept, nous allons transformer une dernière fois le programme `factoriel` de sorte qu'il renvoie la valeur d'un factoriel si un seul argument est transmis au constructeur, ou le produit des deux nombres si deux arguments sont transmis sur la ligne de commande.

Code 4.6 : Surcharge du constructeur

```
/*****************************************/
/*Ce programme reçoit en entrée :        */
/*1 entier : il renvoie le factoriel     */
/*2 entiers: il renvoie leur produit     */
/*****************************************/
using System;

//Classe qui permet la création d'un objet de type
  ➥Factoriel

public class Factoriel
{
   //Données membres :
   private long nFactoriel=1;
   private long FactorielACalculer;
   private bool produit=false;
   private long arg1;
   private long arg2;

   //1er constructeur pour le calcul du factoriel
   public Factoriel(long FactorielACalculer)
   {
      this.FactorielACalculer=FactorielACalculer;
   }

   //2e constructeur pour le calcul du produit
   public Factoriel(long arg1, long arg2)
   {
      produit=true;
      this.arg1 = arg1;
      this.arg2 = arg2;
 }
```

```
    //Fonction membre de calcul du factoriel:
    public long Calcul()
    {
      if (produit == true)
      {
         nFactoriel = arg1*arg2;
      }
      else
      {
        long compteur=1;
        for(compteur=1;compteur<=FactorielACalculer;
        ➥compteur++)
          {
            nFactoriel*=compteur;
          }
      }
       return nFactoriel;
    }
}
/* Fin de la déclaration de la classe Factoriel */

class monApplication
{
   public static void Main(params string[] args)
     {
        //Réception et analyse du 1er argument:
        long argument1 = int.Parse(args[0]);
        //y en a-t-il un second ?
        if (args.Length == 2)
        {
          long argument2 = int.Parse(args[1]);
          Factoriel monFactoriel=new Factoriel(argument1,
          ➥argument2);
          Console.WriteLine("Le produit de {0} et {1} est
          ➥égal à {2} ",
             argument1,argument2, monFactoriel.Calcul());
        }
        else   //un seul argument
        {
        Factoriel monFactoriel=new Factoriel(argument1);
        Console.WriteLine("Le factoriel de {0} est {1} ",
                       argument1, monFactoriel.Calcul());
        }
     }
}
```

Si vous saisissez sur la ligne de commande

```
factoriel 5
```

vous obtenez toujours :

```
Le factoriel de 5 est 120
```

Mais si vous saisissez

```
factoriel 5 6
```

vous obtenez :

```
Le produit de 5 et 6 est égal à 30
```

Les propriétés ont disparu dans ce listing uniquement pour aérer le code et en simplifier la lecture.

Les arguments étant d'abord transmis à la fonction **Main**, il faut lui faire savoir qu'elle va recevoir un nombre variable de paramètres. C'est l'objectif du mot clé **params**. Pour déterminer ensuite le nombre de ces paramètres, nous testons le membre **Length** du tableau **args** (les tableaux sont détaillés au Chapitre 5). Si deux arguments sont détectés, l'objet est créé avec ces deux arguments et le constructeur exécuté (le second) initialise les trois données membres **produit**, **arg1** et **arg2**. Sinon, l'objet est créé comme dans la version précédente du programme et le constructeur appelé (le premier) initialise la variable **factorielACalculer**.

C'est la même fonction **Calcul** qui apporte le résultat dans les deux cas grâce au test de la variable booléenne **produit**. Si cette variable est vraie (**true**), elle calcule le produit, sinon elle renvoie le factoriel.

Pour terminer, examinez la dernière version du programme présentée ci-après. Il donne exactement le même résultat que dans la version précédente, mais c'est la fonction **Calcul()** qui a été surchargée.

Code 4.7 : Surcharge de la fonction *Calcul()*

```
/*****************************************/
/*Ce programme reçoit en entrée :        */
/*1 entier : il renvoie le factoriel     */
/*2 entiers: il renvoie leur produit     */
/*****************************************/
using System;
```

```
//Classe qui permet la création d'un objet de type
  ➥Factoriel

public class Factoriel
{
   //Données membres :
   private long nFactoriel=1;
   private long FactorielACalculer;
   private long arg1;
   private long arg2;

   //constructeur de l'objet factoriel
   public Factoriel()
   {
   }

   //Fonction membre de calcul du factoriel:
   public long Calcul(long FactorielACalculer)
   {
      long compteur=1;
      this.FactorielACalculer=FactorielACalculer;
      for(compteur=1;compteur<=FactorielACalculer;
      ➥compteur++)
         {
           nFactoriel*=compteur;
         }
       return nFactoriel;
   }

   //Fonction surchargée pour le calcul du produit
    public long Calcul(long arg1,long arg2)
   {
      this.arg1=arg1;
      this.arg2=arg2;
      nFactoriel=arg1*arg2;
      return nFactoriel;
   }

}
/* Fin de la déclaration de la classe Factoriel */

class monApplication
{
   public static void Main(params string[] args)
     {
        //Création de l'objet Factoriel
```

```
Factoriel monFactoriel=new Factoriel();
        //Réception et analyse du 1er argument:
        long argument1 = int.Parse(args[0]);
        //y en a-t-il un second ?
        if (args.Length == 2)
        {
          long argument2 = int.Parse(args[1]);
          Console.WriteLine("Le produit de {0} et {1} est
          ➥égal à {2} ", argument1,argument2,
          ➥monFactoriel.Calcul(argument1,argument2));
        }
        else   //un seul argument
        {
        Console.WriteLine("Le factoriel de {0} est {1} ",
        ➥argument1, monFactoriel.Calcul(argument1));
        }
    }
```

Structures de données

Les tableaux

Les tableaux permettent de stocker puis d'accéder simplement à un ensemble de valeurs de même type : les notes d'un étudiant, les jours de la semaine, les arguments d'une fonction, etc. Pour illustrer leur utilisation, nous allons développer un programme de lecture/écriture dans un fichier.

Voici la syntaxe d'une définition de tableau :

```
type[] nom;
nom = new type[taille];
```

ou

```
type[] nom = new type[taille];
```

type représente le type des éléments à stocker dans le tableau, *nom* est le handle, et *taille* correspond au nombre de "cases" à réserver pour recevoir les éléments du tableau.

Comme pour toute variable, le nom du tableau (handle) est l'adresse en mémoire de celui-ci ; il permet donc d'accéder à l'ensemble des éléments, alors que la notation *nom*[*index*] permet d'accéder à chaque élément individuel.

ATTENTION

Les éléments d'un tableau sont numérotés à partir de 0.

Vous obtenez la taille d'un tableau à partir de sa propriété `Length`. Cette propriété est aussi disponible avec les chaînes de caractères, puisque ces dernières sont aussi des tableaux (voir Chapitre 2).

Lecture dans un fichier

Tout ce qui n'a pas déjà été abordé apparaît en caractères gras et est expliqué à la suite du programme.

Code 5.1 : Utilisation des tableaux

```
/*Ce programme reçoit en entrée :        */
/*le nom du fichier à lire               */
/****************************************/
 1:using System;
 2:using System.IO;
 3:
 4:class Fichier
 5:{
 6:   string tampon;  // pour stocker temporairement chaque
      ➥ligne du fichier
 7:   String[] nTableau; //pour stocker toutes les lignes
      ➥du fichier
 8:
 9:   //Propriété pour l'accès à nTableau
10:   public String[] Tableau
11:   {
12:      get { return nTableau; }
13:      set { nTableau=value;  }
14:   }
15:
16:   //Fonction membre Lecture():
17:   public void  Lecture(string nomFichier)
18:   {
19:      StreamReader fluxLecture = File.OpenText(nomFichier);
20:      int index=0;
21:
22:       //On compte le nombre de lignes du fichier
23:        while ((tampon=fluxLecture.ReadLine()) != null)
24:          {
25:           index+=1;
```

```
26:          }
27:
28:      //On peut maintenant réserver la mémoire pour le
           ➥tableau:
29:      Tableau = new String[index];
30:
31:       //On se replace au début du flux
32:       fluxLecture.BaseStream.Seek(0, SeekOrigin.Begin);
33:
34:       //On stocke dans le Tableau toutes les lignes de
            ➥texte
35:       for(int compteur=0; compteur<index; compteur++)
36:       {
37:          Tableau[compteur]= fluxLecture.ReadLine();
38:          Console.WriteLine("Tableau[{0}] = {1}",
             ➥compteur, Tableau[compteur]);
39:       }
40:
41:       fluxLecture.Close();
42:    }   //Fin de la fonction Lecture()
43:}       //Fin de la classe Fichier
44:
45:class monApplication
46:{
47:    public static void Main(string[] args)
48:    {
49:    if (args.Length < 1)   // si aucun argument transmis
50:      {
51:      //On explique comment utiliser le programme
52:      Console.WriteLine("Syntaxe: fichier nom du
         ➥fichier ");
53:      }
54:    else
55:     {
56:     //Création d'un objet de type Fichier
57:     Fichier monFichier = new Fichier();
58:
59:     //On appelle la fonction de lecture
60:     monFichier.Lecture(args[0]);
61:     }
62:
63:    } //Fin de Main()
64:
65:}  //Fin de la classe monApplication
```

Si vous exécutez ce programme sur le fichier texte suivant :

```
Je vous souhaite des rêves à n'en plus finir
Et l'envie furieuse d'en réaliser quelques-uns.
Je vous souhaite d'aimer ce qu'il faut aimer
Et d'oublier ce qu'il faut oublier.
```

vous allez obtenir :

```
Tableau[0] : Je vous souhaite des rêves à n'en plus finir
Tableau[1] : Et l'envie furieuse d'en réaliser quelques-uns.
Tableau[2] : Je vous souhaite d'aimer ce qu'il faut aimer
Tableau[3] : Et d'oublier ce qu'il faut oublier.
```

Analyse du programme

System.IO

```
2:using System.IO;
```

L'espace de noms `System.IO` contient des types de données permettant d'effectuer des opérations de lecture et d'écriture sur des flux et des fichiers de données. Il fournit en particulier les classes `FileInfo` et `StreamReader` utilisées dans ce programme.

Comme son nom l'indique, un flux est un flot d'informations qui va permettre d'envoyer ou de recevoir des données en provenance de la mémoire, du réseau, d'Internet, d'une chaîne de caractères, etc. Vous pouvez le comparer à un tuyau dans lequel vont transiter les données et dont les deux extrémités peuvent être reliées indifféremment à un fichier, au réseau, et ainsi de suite.

Déclaration du tableau

```
7:   String[] nTableau;
```

C'est dans ce tableau `nTableau` de type `String` que nous allons stocker par la suite chaque ligne du fichier texte. Pour le définir complètement, il reste à déterminer sa taille, c'est-à-dire le nombre de lignes que contient le fichier à lire.

Définition de l'objet flux

```
19:     StreamReader fluxLecture = File.OpenText(nomFichier);
```

La classe `File`, comme la classe `FileInfo`, contient diverses méthodes pour ouvrir un fichier en lecture et/ou en écriture. Toutes les méthodes de cette classe étant statiques, il n'est pas nécessaire de créer un objet de type `File` (les fonctions statiques sont examinées au Chapitre 3). L'appel de la fonction `File.OpenText` avec le nom de fichier transmis en argument à la fonction `Lecture()` crée un flux de type `StreamReader` pour une lecture du fichier.

Il existe plusieurs types de flux, et vous choisissez en fonction du contenu du fichier à lire ou à écrire. Puisque notre programme lit un fichier texte, `StreamReader` est la classe appropriée (pour manipuler des octets, vous choisirez plutôt la classe `Stream`). L'instruction de la ligne 19 crée le flux `fluxLecture` et l'alimente avec le contenu du fichier dont le nom est transmis en argument à la fonction `Lecture()`. En effet, un flux est une sorte de conteneur et vous devez le "brancher" sur les données à traiter. Vous pouvez imaginer que les informations sont placées à la queue leu leu dans ce conteneur et que vous vous déplacez dans cette file à mesure que vous traitez les données l'une après l'autre.

ReadLine()

```
23:     while ((tampon=fluxLecture.ReadLine()) != null)
```

L'instruction de la ligne 23 stocke une ligne du flux (`ReadLine()` signifie lire une ligne) dans la variable `tampon`, puis vérifie que cette variable n'est pas vide. La boucle `while` va incrémenter la variable `index` à chaque exécution de cette instruction, jusqu'à ce que la fin de fichier soit atteinte. A ce moment-là, `ReadFile` ne va rien renvoyer, `tampon` devient une chaîne vide, et la boucle `while` se termine (vous trouverez la syntaxe de `while` à l'annexe).

Définition du Tableau

```
29:     Tableau = new String[index];
```

La boucle `while` a permis de déterminer le nombre de lignes du fichier. Nous pouvons donc maintenant réserver la mémoire nécessaire pour le tableau.

Position dans le flux

```
32:         fluxLecture.BaseStream.Seek(0, SeekOrigin.Begin);
```

Après l'exécution de la boucle `while`, nous nous trouvons à la fin du flux. Nous devons donc nous repositionner au début. La fonction statique `Seek()` de la classe `BaseStream` permet de se positionner dans un flux en indiquant un déplacement en octets (1er argument 0) par rapport à un point de référence dans le flux (2nd argument `SeekOrigin.Begin` qui représente le début du flux).

Close()

Vous fermez le flux après son utilisation.

Les structures

Les structures sont en tout point identiques aux classes, à l'exception du fait — et c'est important — qu'il s'agit d'un type de valeur, alors que les classes font partie des types de références et que le mot clé `struct` remplace le mot clé `class`.

NOUVEAU

En C++, une structure et une classe sont plus ou moins la même chose. La seule différence réside dans la portée par défaut (public pour des structures, privé pour des classes). Cependant, en C#, les structures et les classes sont très différentes. Dans C#, les structures sont des types de *valeurs* (enregistrés sur la pile), tandis que les classes sont des types de *références* (enregistrés sur le segment de mémoire ou tas).

Comme nous l'avons expliqué au Chapitre 2, si vous transmettez un type de valeur, vous transmettez directement une copie de la valeur des données, alors qu'avec un type de référence, vous ne transmettez que l'adresse de ces données. Si vous devez manipuler de grosses quantités de données, choisissez de travailler avec une classe plutôt qu'avec une structure, surtout si vous devez les transmettre comme argument à une fonction.

INFO

Travaillez avec une structure si la taille de l'ensemble de ses membres n'excède pas 16 octets, sinon créez une classe.

Code 5.2 : Utilisation des structures

```
 1:using System;
 2:
 3:// Définition de la structure maStructure (type de
     ➥valeur)
 4:
 5:struct maStructure
 6:{
 7:  int x, y;
 8:  public maStructure(int x, int y)
 9:  {
10:    this.x = x;
11:    this.y = y;
12:  }
13:  //Redéfinition de la fonction d'affichage
14:  public override String ToString()
15:  {
16:    return("(" + x + "," + y + ")");
17:  }
18:}       //Fin de la structure maStructure
19:
20://Définition de la classe maClasse (type de références)
21:class maClasse
22:{
23:  int x;
24:  string chaine;
25:  //Constructeur de maClasse
26:  public maClasse(int x, string chaine)
27:  {
28:    this.x = x;
29:    this.chaine = chaine;
30:  }
31:
32:  //Redéfinition de la fonction d'affichage
33:  public override String ToString()
34:  {
35:    return("(" + x + "," + chaine + ")");
36:  }
37:
```

```
38:  //Redéfinition de l'opérateur =
39:  public override bool Equals(Object o)
40:  {
41:  //On modifie la sémantique de ce type de références
       ➥pour considérer
42:  //que 2 objets sont égaux si leurs membres sont égaux.
43:    Console.WriteLine("Avec ma propre fonction
       ➥maClasse.Equals:");
44:    maClasse r = (maClasse) o;
45:    return(r.x == x && r.chaine == chaine);
46:  }
47:
48:}     //Fin de la classe maClasse
49:
50:class monApplication
51:{
52:  public static void Main ()
53:  {
54:    Console.WriteLine("Démo de l'utilisation de types de
       ➥valeurs\n");
55:
56:    maStructure p1 = new maStructure(5, 10);
57:    maStructure p2 = new maStructure(5, 10);
58:    maStructure p3 = new maStructure(3, 4);
59:
60:    Console.WriteLine("Soit les trois structures
       ➥suivantes:");
61:    Console.WriteLine("p1={0}, p2={1}, p3={2}\n", p1,
       ➥p2, p3);
62:
63:    //La comparaison de 2 types de valeurs s'effectue
64:    //en comparant leurs membres
65:    Console.WriteLine("Test de l'égalité de p1 avec p1:
       ➥" + p1.Equals(p1));
66:    Console.WriteLine("Test de l'égalité de p1 avec p2:
       ➥" + p1.Equals(p2));
67:    Console.WriteLine("Test de l'égalité de p1 avec p3:
       ➥{0}\n", p1.Equals(p3));
68:
69:    Console.WriteLine("\nExemple d'utilisation de types
       ➥de références\n");
70:    maClasse r = new maClasse(1, "Il fait beau");
71:
72:    Console.WriteLine("Soit l'objet " + r + " de type
       ➥maClasse");
73:
```

```
74:     //Les types de références sont égaux s'ils se
          ➥rapportent au même objet
75:     Console.WriteLine("Est-il égal à (1,Il fait beau):"
76:           + (r == new maClasse(1, "Il fait beau")));// non
77:     Console.WriteLine("Est-il égal à (1,Il fait beau): "
78:   + (r.Equals(new maClasse(1, "Il fait beau"))));
      ➥// True
79:     Console.WriteLine("Est-il égal à (2,Il ne fait pas
        ➥beau):"
80:       +  (r.Equals(new maClasse(2, "Il ne fait pas
          ➥beau"))));  // False
81:
82:   }  //Fin de la fonction Main()
83:
84:}     //Fin de la classe monApplication
```

L'exécution de ce programme donne le résultat suivant :

```
 1:Démo de l'utilisation de types de valeurs
 2:
 3:Soit les trois structures suivantes:
 4:p1=(5,10), p2=(5,10), p3=(3,4)
 5:
 6:Test de l'égalité de p1 avec p1: True
 7:Test de l'égalité de p1 avec p2: True
 8:Test de l'égalité de p1 avec p3: False
 9:
10: Exemple d'utilisation de types de références.
11:
12:Soit l'objet (1,Il fait beau) de type maClasse
13:Est-il égal à (1,Il fait beau): False
14:Avec ma propre fonction maClasse.Equals:
15:Est-il égal à (1,Il fait beau): True
16:Avec ma propre fonction maClasse.Equals:
17:Est-il égal à (2,Il ne fait pas beau): False
```

Définition de la structure

La structure `maStructure` est définie aux lignes 5 à 17.

A l'exception du mot clé `struct`, ce code devrait vous être familier. Le constructeur, qui reçoit deux arguments, est déclaré aux lignes 8 à 12.

Redéfinition de ToString

La fonction membre `ToString`, déclarée à la ligne 14, est une fonction virtuelle de la classe `Object`. Cette fonction renvoie une chaîne de caractères représentant l'objet courant (s'il s'agit d'un type connu). L'héritage et les classes virtuelles étant traités au chapitre suivant, retenez simplement que nous redéfinissons cette fonction pour l'adapter à notre objet `maClasse`. Cette fonction est appelée *implicitement* dès qu'il s'agit d'afficher la valeur d'un objet.

La ligne 4 du résultat, par exemple, qui affiche la valeur des trois structures `p1`, `p2` et `p3`, est obtenue avec l'instruction de la ligne 66 :

```
Console.WriteLine("p1={0}, p2={1}, p3={2}\n", p1, p2, p3);
```

Nous aurions pu aussi bien coder :

```
Console.WriteLine("p1={0}, p2={1}, p3={2}\n", p1.ToString,
p2.ToString, p3.ToString);
```

Si nous n'avions pas *redéfini* `ToString`, nous aurions obtenu ce que cette fonction affiche dans sa version de base, c'est-à-dire le nom du type. A la ligne 4 du résultat, nous aurions vu :

```
p1=maStructure, p2=maStructure, p3=maStructure
```

Cette fonction est redéfinie une seconde fois aux lignes 32 à 36 pour afficher la valeur d'un objet de type `maClasse`. Tous les objets de cette classe, définie aux lignes 19 à 48, se composent d'un entier et d'une chaîne.

Redéfinition de l'opérateur =

Ce programme démontre que deux types de valeurs sont naturellement égaux si chacun de leurs membres ont la même valeur. Il démontre aussi que deux types de références, comme deux objets différents de la classe `maClasse`, sont toujours considérés comme différents même s'ils possèdent des membres de même valeur.

Si votre programme a besoin de "comparer" des types de références et si vous voulez les considérer comme égaux si leurs membres ont la même valeur, vous allez redéfinir l'opérateur = pour obtenir le comportement recherché, exactement comme vous avez redéfini la fonction `ToString`. L'opérateur = est redéfini aux lignes 38 à 46.

Pour comprendre la syntaxe de cette redéfinition, lisez le Chapitre 6.

Analyse du résultat

Le programme crée les trois structures **p1**, **p2**, **p3**, les affiche (ligne 4 du résultat), puis les compare.

Vous constatez en observant le résultat que **p1** est bien égale à **p2** et que **p1** est différente de **p3**.

Le programme crée ensuite un objet de type `maClasse`, et le résultat montre (ligne 13) que cet objet est bien considéré comme différent d'un autre objet, même si les membres ont la même valeur.

Les énumérations

L'énumération est un autre type de donnée qui permet de créer des variables contenant un nombre limité de valeurs (pour représenter les sept jours de la semaine ou un bouton on/off, par exemple).

Le programme qui suit illustre la définition et l'accès aux différents membres d'une énumération.

La valeur par défaut d'une variable de type énumération, si elle n'est pas initialisée, est 0.

Si les membres d'une énumération ne sont pas initialisés dans sa déclaration, ils prennent les valeurs par défaut 0, 1, 2, etc., selon l'ordre dans lequel ils sont déclarés.

Code 5.3 : Utilisation des énumérations

```
/*********************************************/
 1:using System;
 2:
 3:class monApplication
 4:{
 5:  //Déclaration de l'énumération
 6:  enum Mois  //mot clé enum suivi du nom de l'énumération
 7:  {
 8:    //déclaration et initialisation des membres de
       ⇨l'énumération
 9:    Janvier = 1,
10:    Fevrier,
11:    Mars,
12:    Avril,
13:    Mai,
14:    Juin,
15:    Juillet,
16:    Aout,
17:    Septembre,
18:    Octobre,
19:    Novembre,
20:    Decembre
21:  }
22:
23:  //Déclaration de la structure Anniversaire
24:  struct Anniversaire
25:  {
26:    public int jour;
27:    public Mois mois;
28:    public int annee;
29:  }
30:
31:  public static void Main ()
32:  {
33:    //Déclaration de la structure monAnniversaire
34:    Anniversaire monAnniversaire;
35:    //Définition de ses membres:
36:    monAnniversaire.jour = 22;
37:    monAnniversaire.mois = Mois.Mai;
38:    monAnniversaire.annee = 1961;
39:
40:    Console.WriteLine("Voici la date de mon anniversaire :
41:          le {0} {1} {2}", monAnniversaire.jour,
42:          monAnniversaire.mois, monAnniversaire.annee);
43:  }
44:}
```

On obtient l'affichage suivant :

```
Voici la date de mon anniversaire : le 22 Mai 1961
```

L'énumération `Mois` est déclarée avec les douze mois de l'année aux lignes 6 à 21.

Si les membres d'une énumération ne sont pas initialisés dans sa déclaration, ils prennent les valeurs par défaut 0, 1, 2, etc., selon l'ordre dans lequel ils sont déclarés. En définissant le membre `Janvier` à 1, vous "décalez" la valeur des autres membres, et les mois se trouvent correctement numérotés. Si vous initialisez tous les membres, vous pouvez bien sûr leur attribuer des valeurs distinctes sans respecter de séquence.

A la ligne 27, le membre `mois` de type `Mois` est déclaré dans la structure `Anniversaire`.

La structure `Anniversaire` se compose de deux entiers pour représenter le jour et l'année, et d'une énumération `Mois` pour représenter le mois.

Les membres de `monAnniversaire` sont initialisés aux lignes 36 à 38.

A la ligne 37, le membre `mois` de `monAnniversaire` est initialisé avec la valeur de `Mois.Mai`, c'est-à-dire 5.

Les indexeurs

Pour terminer ce chapitre consacré aux structures de données, nous allons parler des indexeurs.

Un indexeur permet d'accéder aux membres d'un objet par l'intermédiaire d'un index, en traitant l'objet comme un tableau. Comme les propriétés, il s'appuie sur les fonctions `get()` et `set()`.

Le Code 5.4 illustre bien la définition et l'utilisation d'un indexeur.

Code 5.4 : Définition et utilisation d'un indexeur

```
using System;

public class Annuaire
{
  string[] noms = new string[3];
```

```
  string[] numeros = new string[3];

    //constructeur
    public Annuaire()
    {
      noms[0]="Charlotte";
      numeros[0]="0355667788";
      noms[1] = "Arthur";
      numeros[1]="0422667788";
      noms[2]="Romeo";
      numeros[2]="0555227788";
    }

    //définition de l'indexeur
    public string this[int Index]
    {
      get
      {
        return String.Format("{0} : {1}",noms[Index],
        ➥numeros[Index]);
      }
      set
      {
        numeros[Index]=value;
      }
    }

    //propriété Taille
    public int Taille
    {
      get { return (noms.Length);}
    }
  }  //Fin de Annuaire

  public class monApplication
  {
    //La fonction principale
    public static void Main()
    {
      Annuaire monBottin = new Annuaire();
      Afficher(monBottin);
      Console.WriteLine("Quel numéro voulez-vous changer?");
      Console.WriteLine("Aucun, saisissez 0");
      Console.Write("Sinon, saisissez son rang :");
      int reponse=Console.Read();
      if(reponse!=0)
```

```
    {
        Console.Write("Saisissez le nouveau numéro: ");
        string nouvNum = Console.ReadLine();
        monBottin[reponse]=nouvNum;
        Console.WriteLine("Voici l'annuaire modifié:");
       Afficher(monBottin);
       }

    }
       public static void  Afficher(Annuaire a)
       {
       //On affiche le contenu de monBottin:
       int compteur = a.Taille;
       for(int i =0;i < compteur;i++)
          Console.WriteLine(a[i]);
       }

}
```

Vous obtenez le résultat :

```
Charlotte : 0355667788
Arthur : 0422667788
Romeo : 0555227788
Quel numéro voulez-vous changer?
Aucun, saisissez 0
Sinon, saisissez son rang : 2
Saisissez le nouveau numéro: 00000000
Voici l'annuaire modifié:
Charlotte : 0355667788
Arthur : 00000000
Romeo : 0555227788
```

La définition d'un indexeur contient toujours le mot clé **this** pour faire référence à l'objet courant. La fonction **get()** permet non pas d'obtenir un membre spécifique comme pour les propriétés, mais une valeur de l'objet lui-même, et la fonction **set()** de définir une valeur.

Héritage

La notion d'héritage est très facile à comprendre. En orienté objet, on dit qu'une classe définie à partir d'une classe existante et à laquelle on ajoute une fonctionnalité *dérive* de la classe initiale. Cette dernière est la classe de base, et la classe dérivée hérite de ses données et fonctions membres.

La dérivation permet d'exprimer la relation "est un". En effet, si nous reprenons notre exemple de zoo du Chapitre 1, un lion *est un* carnivore, un carnivore *est un* animal, etc. (voir Figure 6.1). Les classes dérivées (également nommées classes filles) héritent des données et fonctions de toutes leurs classes de base (également nommées classes mères). La classe `Lion` va donc hériter de la classe `Animal`, ainsi que des données et fonctions spécifiques à la classe `Carnivore`. C'est pour cette raison que les données et fonctions partagées par certaines catégories d'objets doivent être définies au niveau le plus haut possible dans cette hiérarchie. Dans notre exemple, il faudra définir des données membres telles que `prenom`, `age`, `couleur`, `taille` et les fonctions membres `Manger()` et `Dormir()` au niveau de la classe `Animal`, puisque tous les animaux possèdent ces caractéristiques et ce comportement.

Dérivation des classes

Le premier avantage de l'héritage est que nous pouvons réutiliser des classes existantes comme classes de base et les spécialiser en écrivant uniquement le code correspondant à la *nouvelle fonctionnalité*. Oubliées les opérations de copier-coller qui étaient potentiellement dangereuses en cas de modification du bloc de code reproduit (Où a-t-il été dupliqué ? Combien de fois ?).

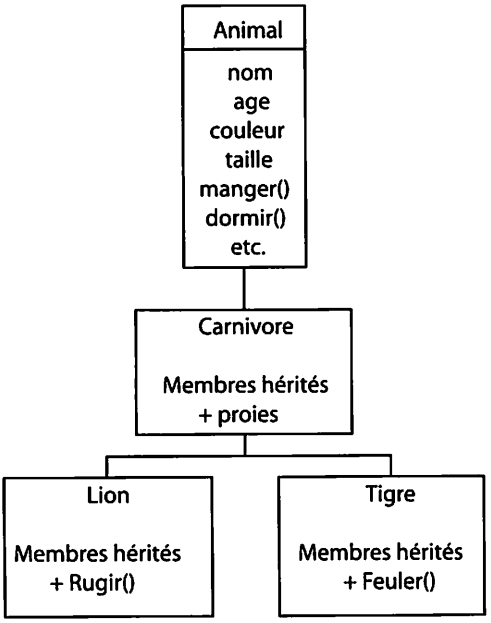


Figure 6.1
Détail de l'héritage entre la classe Animal *et les deux classes* Lion *et* Tigre.

Dans notre hiérarchie de classes, toute modification au niveau d'une classe de base est automatiquement répercutée dans toutes ses classes dérivées.

Cela n'empêche pas les classes dérivées de redéfinir une partie des caractéristiques des classes de base et d'en acquérir de nouvelles.

NOUVEAU

Contrairement au C++, il n'existe pas d'héritage multiple en C#. En effet, une classe ne peut hériter de plusieurs classes de base différentes. A la Figure 6.1, la classe `Gorille` possède une seule classe mère, la classe `Singe`, et cette dernière hérite de la seule classe `Végétarien`, etc.

Les interfaces de C#, traitées un peu plus loin, remplacent en la simplifiant cette notion d'héritage multiple.

Syntaxe

```
class <classe dérivée> : <classe de base>
```

La meilleure façon d'illustrer l'héritage est d'étudier un programme représentatif. Nous allons reprendre le programme du Chapitre 3 dans lequel nous avions créé une classe `ExPetitAmi`. Nous allons ajouter dans ce programme une classe `Ami` et une autre `PetitAmiCourant`. La classe `Ami` est la classe mère (ou classe de base), et `ExPetitAmi` et `PetitAmiCourant` sont ses deux classes filles. La Figure 6.2 présente cette hiérarchie de classes avec les membres associés à chacune d'elle.

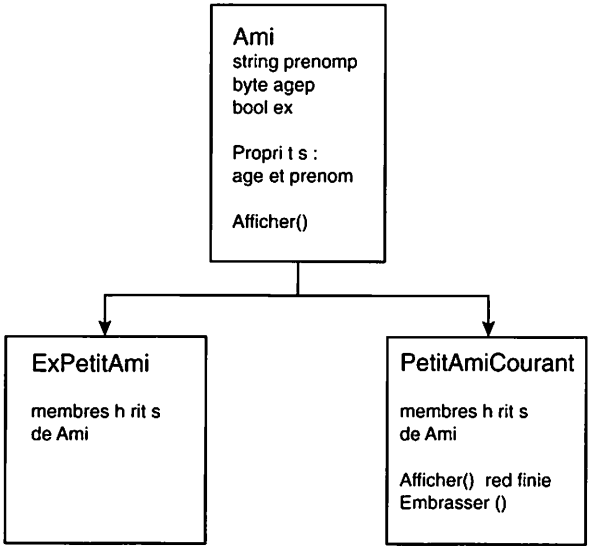


Figure 6.2
Hiérarchie des classes Ami, PetitAmiCourant *et* ExPetitAmi.

Code 6.1 : Déclaration d'une classe de base et de deux classes dérivées

```
1:using System;
2:
3://Déclaration de la classe mère Ami
4:class Ami
5:{
```

```
 6:   //données membres privées
 7:   string prenomp ;
 8:   byte agep ;
 9:   public bool ex = false ;
10:
11:   //constructeurs
12:   public Ami()
13:   {
14:      Console.WriteLine("Exécution du constructeur de
      ➥Ami");
15:   }
16:
17:   public Ami(string pr)
18:   {
19:      Console.WriteLine("Exécution du constructeur de
      ➥Ami");
20:      prenom = pr;
21:   }
22:
23:   public Ami(string pr, byte a)
24:   {
25:      Console.WriteLine("Exécution du constructeur de
      ➥Ami");
26:      prenom = pr;
27:      age = a;
28:   }
29:
30:   //Propriété pour l'accès à la variable prenomp
31:   public string prenom
32:   {
33:      get { return prenomp ; }
34:      set { prenomp = value ;}
35:   }
36:
37:   // Propriété pour l'accès à la variable agep
38:   public byte age
39:   {
40:      get { return agep ; }
41:      set
42:      { if (value > 5 && value <80) //si la valeur est
         ➥comprise entre 5 et 80
43:           agep = value;            //OK
44:        else                        //sinon
45:        throw new Exception("L'âge doit être compris
          ➥entre 5 et 80");
46:      }
```

```
47:    }
48:
49:   //Fonction membre Afficher
50:    public void Afficher()
51:   {
52:     Console.WriteLine("C'est un ami");
53:     if (this.ex == true)
54:       Console.WriteLine("\tC'est aussi un ex");
55:   }
56:}     //Fin de Ami
57:
58://Déclaration de la classe fille ExPetitAmi
59:class ExPetitAmi : Ami
60:{
61:    //constructeurs
62:    public ExPetitAmi() : base()
63:    {
64:       Console.WriteLine("Puis de celui de ExPetitAmi");
65:       ex = true ;
66:    }
67:
68:
69:    public ExPetitAmi(string pr) : base(pr)
70:    {
71:       Console.WriteLine("Puis de celui de ExPetitAmi");
72:       ex = true ;
73:    }
74:
75:
76:    public ExPetitAmi(string pr, byte a) : base (pr, a)
77:    {
78:       Console.WriteLine("Puis de celui de ExPetitAmi");
79:       ex = true ;
80:    }
81:
82:
83:}
84:
85://Déclaration de la classe fille PetitAmiCourant
86:class PetitAmiCourant : Ami
87:{
88:    //constructeurs
89:    public PetitAmiCourant() : base()
90:    {
91:      Console.WriteLine("Puis de celui de
       ➥PetitAmiCourant");
```

```
92:    }
93:
94:    public PetitAmiCourant(string pr) : base(pr)
95:    {
96:        Console.WriteLine("Puis de celui de
           ➥PetitAmiCourant");
97:    }
98:
99:
100:    public PetitAmiCourant(string pr, byte a) :
        ➥base(pr, a)
101:    {
102:        Console.WriteLine("Puis de celui de
           ➥PetitAmiCourant");
103:    }
104:
105:   //Fonction membre Afficher
106:    public new void Afficher()
107:   {
108:     Console.WriteLine("C'est l'élu actuel");
109:   }
110:
111:   //Fonction membre Embrasser
112:    public void Embrasser()
113:   {
114:     Console.WriteLine("Il peut embrasser!");
115:   }
116:
117:}
118:
119:class monApplication
120:{
121:    public static void Main()
122:    {
123:        Console.WriteLine("Création de l'objet Maurice
           ➥(Ex):\n");
124:        ExPetitAmi Maurice = new ExPetitAmi("Maurice");
125:        //Initialisation de l'âge de Maurice :
126:        Maurice.age = 19 ;
127:        Console.WriteLine("\nCréation de l'objet Marcel
           ➥(Ami):\n");
128:        Ami Marcel = new Ami("Marcel",21);
129:        Console.WriteLine("\nCréation de l'objet Brad
           ➥(c'est lui):\n");
130:        PetitAmiCourant Brad = new PetitAmiCourant ();
131:        //Initialisation des 2 membres nom et age de Brad :
```

```
132:          Brad.prenom = "Brad" ;
133:          Brad.age = 20 ;
134:   /*
135:    *Trois objets sont maintenant créés en mémoire
136:    */
137:      Console.Write("\nQui est Maurice?\n\t");
138:      Maurice.Afficher();
139:      Console.Write("Qui est Marcel?\n\t");
140:      Marcel.Afficher();
141:      Console.Write("Qui est Brad?\n\t");
142:      Brad.Afficher();
143:      Console.WriteLine("Fin du programme");
144:    }
145:}
```

L'exécution donne le résultat suivant :

```
 1: Création de l'objet Maurice (Ex):
 2:
 3: Exécution du constructeur de Ami
 4: Puis de celui de ExPetitAmi
 5:
 6: Création de l'objet Marcel (Ami):
 7:
 8: Exécution du constructeur de Ami
 9:
10: Création de l'objet Brad (c'est lui):
11:
12: Exécution du constructeur de Ami
13: Puis de celui de PetitAmiCourant
14:
15: Qui est Maurice?
16:   C'est un ami
17:   C'est aussi un ex
18: Qui est Marcel?
19:   C'est un ami
20: Qui est Brad?
21:   C'est l'élu actuel
22: Fin du programme
```

Analyse du programme

La déclaration de la classe `Ami` est tout à fait classique et ne contient aucun élément nouveau.

Aux lignes 11 à 38, les trois constructeurs de cette classe de base offrent trois possibilités de créer un objet `Ami` : sans argument, en transmettant le prénom, ou en transmettant le prénom et l'âge.

Le lien entre `ExPetitAmi` et `Ami` apparaît sur la première ligne de la déclaration de la classe fille à la ligne 59 : `ExPetitAmi : Ami`. La classe `ExPetitAmi` hérite de tous les membres de `Ami` et contient trois constructeurs à l'image de ceux de la classe mère. Examinez l'en-tête de ces trois constructeurs. La notation `base(arguments)` permet d'exécuter le code du constructeur de la classe de base en lui transmettant les arguments reçus par le constructeur de la classe fille, *puis* d'exécuter les instructions de ce dernier constructeur (dans ce cas, la donnée membre `ex` héritée de la classe de base est définie en `true` dans cette classe). Ce mécanisme est illustré dans la sortie du programme, puisque chaque exécution de constructeur est signalée avec une instruction `Console.WriteLine`.

La seconde classe fille `PetitAmiCourant` est déclarée à la ligne 85. A la ligne 106, l'en-tête de la fonction `Afficher()` comporte le mot clé `new`, parce que cette fonction existe déjà dans la classe de base et qu'elle est *redéfinie* dans cette classe fille.

La fonction principale `Main` commence par créer trois objets appartenant aux trois classes conçues en utilisant les trois méthodes disponibles (sans argument, en transmettant le prénom, ou en transmettant le prénom et l'âge). Vous constatez aux lignes 3-4 et 12-13 que pour chaque objet de type `ExPetitAmi` ou `PetitAmiCourant`, les deux constructeurs des classes mère et fille ont été exécutés.

`Main` appelle ensuite la fonction `Afficher()` associée à chaque objet :

- A la ligne 138, c'est la fonction définie aux lignes 49 à 55 qui est exécutée, puisque la classe `ExPetitAmi` hérite de la fonction définie dans sa classe de base.
- A la ligne 140, c'est toujours la fonction définie aux lignes 49 à 55 qui est exécutée, puisque l'objet est de type `Ami`.
- A la ligne 142, c'est la fonction redéfinie aux lignes 106 à 109 qui est exécutée, puisque la classe `PetitAmiCourant` à laquelle l'objet appartient remplace la fonction de la classe de base par sa propre version.

Polymorphisme via l'héritage

Vous venez de découvrir avec la fonction `Afficher()` et la donnée membre `ex` qu'il est possible de "redéfinir" les fonctions et données d'une classe de base dans une des classes dérivées pour l'adapter au nouveau comportement de la classe. Dans le cas d'une fonction, elle doit conserver le même type de valeur retournée et la même signature (nom + nombre et type des arguments). Son implémentation sera différente dans la classe dérivée. On dit qu'il y a *substitution de fonction*.

Il s'agit de la seconde forme du polymorphisme. En effet, vous allez toujours appeler la fonction `Afficher()` quel que soit le type de l'objet créé, mais la fonction exécutée sera bien celle qui a été définie pour cet objet particulier.

Un même nom pour un comportement différent, c'est bien la définition du polymorphisme.

ATTENTION

Vous ne devez pas confondre substitution de fonction et surcharge de fonction. Le mécanisme est similaire mais, dans le premier cas, vous modifiez le corps de la fonction dans une classe dérivée sans changer la signature. Dans le deuxième cas, la surcharge consiste à déclarer "plusieurs versions" d'une même fonction dans une portée déterminée, chaque version étant associée à une signature différente.

Accès aux méthodes de la classe de base

Plutôt que de redéfinir complètement une fonction dans une classe fille, vous avez la possibilité "d'étendre" sa fonctionnalité. Ainsi, si vous modifiez le programme précédent de la façon suivante :

```
//Fonction membre Afficher de la classe mère Ami
 public void Afficher()
{
  Console.WriteLine("C'est un ami");
  if (this.ex == true)
    Console.WriteLine("\tC'est aussi un ex");
}
```

et que vous introduisiez cette nouvelle définition de fonction dans la classe `ExPetitAmi` :

```
//Fonction membre Afficher de la classe mère Ami
 public void Afficher()
{
  base.Afficher();
  Console.WriteLine("C'est aussi un ex");
```

vous obtenez exactement le même résultat puisque l'appel de la fonction `Afficher()` sur un objet de type `ExPetitAmi` se traduit par l'appel de la fonction `Afficher()` de la classe de base (`base.Afficher()`), puis par l'affichage de la chaîne "`C'est aussi un ex`".

INFO

Le mot clé base permet d'étendre facilement la fonctionnalité d'une classe de base sans avoir besoin de réécrire les instructions communes.

Mot clé protected

La classe dérivée hérite de tous les membres publics et protégés (`protected`) de sa classe de base. Elle n'a pas accès aux membres privés, sauf si des propriétés ont été définies pour ces derniers. Les membres protégés de la classe de base se comportent comme des membres privés vis-à-vis de toute autre classe.

Les fonctions virtuelles

Quand une fonction est redéfinie dans une classe dérivée, le mécanisme d'exécution n'utilise pas les mêmes critères pour sélectionner la bonne version de la fonction, selon que cette dernière est virtuelle ou non.

Dans le cas d'une fonction virtuelle, la version est choisie en fonction du type de l'objet et non de sa référence.

Dans le cas d'une fonction non virtuelle, la version est choisie en fonction de la référence. Vous allez comprendre en étudiant le code qui suit. Cet exemple étant uniquement destiné à illustrer les fonctions virtuelles, nous l'avons réduit à sa plus simple expression en supprimant la surcharge des constructeurs et les propriétés.

Code 6.2 : Fonction virtuelle en action

```
 1:using System;
 2:
 3://Déclaration de la classe mère Ami
 4:class Ami
 5:{
 6:    public bool ex = false ;
 7:
 8:    //constructeurs
 9:    public Ami() {}
10:
11:    //Fonction membre Afficher
12:    public virtual void Afficher()
13:   {
14:     Console.WriteLine("Exécution de la version Ami
        ➥d'Afficher()");
15:   }
16:}
17:
18://Déclaration de la classe fille ExPetitAmi
19:class ExPetitAmi : Ami
20:{
21:    //constructeurs
22:    public ExPetitAmi() : base () {}
23:
24:    //Fonction membre Afficher
25:    public override void Afficher()
26:    {
27:     Console.WriteLine("Exécution de la version
        ➥ExPetitAmi d'Afficher()");
28:    }
29:}
30:
31: //Déclaration de la classe fille PetitAmiCourant
32: class PetitAmiCourant : Ami
33: {
34:     //constructeurs
35:     public PetitAmiCourant() : base(){ }
36:
```

```
37:    //Fonction membre Afficher
38:     public override void Afficher()
39:    {
40:     Console.WriteLine("Exécution de la version
        ➥PetitAmiCourant d'Afficher()");
41:    }
42: }
43:
44: class monApplication
45: {
46:     public static void Main()
47:     {
48:       Console.WriteLine("Création de l'objet Maurice
          ➥(Ex):");
49:        Ami Maurice = new ExPetitAmi();
50:        Maurice.Afficher();
51:       Console.WriteLine("\nCréation de l'objet Marcel
          ➥(Ami):");
52:       Ami Marcel = new Ami();
53:       Marcel.Afficher();
54:       Console.WriteLine("\nCréation de l'objet Brad
          ➥(PetitAmiCourant):");
55:       Ami Brad = new PetitAmiCourant ();
56:       Brad.Afficher();
57:    }
58:}
```

Nous obtenons le résultat :

```
Création de l'objet Maurice (Ex):
Exécution de la version ExPetitAmi d'Afficher()

Création de l'objet Marcel (Ami):
Exécution de la version Ami d'Afficher()

Création de l'objet Brad (PetitAmiCourant):
Exécution de la version PetitAmiCourant d'Afficher()
```

Dans ce programme, la fonction `Afficher()` est déclarée comme virtuelle dans la classe de base (ligne 12, mot clé `virtual`), puis elle est redéfinie dans chaque classe dérivée avec le mot clé `override`. Ce mot clé signale la redéfinition d'une fonction virtuelle. Si une des redéfinitions s'effectue avec `new` plutôt qu'avec `override`, le choix de la fonction pour les objets de ce type se fera sur la référence plutôt que sur l'affectation.

Dans la fonction `Main`, nous créons trois objets de type `Ami` auxquels nous affectons (*via* l'opérateur =) un objet de type `ExPetitAmi`, un objet de type `Ami`, et un objet de type `PetitAmiCourant`.

Le résultat montre bien que le choix de la fonction est fondé sur le type de la valeur affectée à l'objet et non sur le type `Ami` de référence.

Les classes abstraites

Avec les fonctions virtuelles, vous pouvez choisir d'utiliser la version redéfinie ou la version de la classe de base en créant judicieusement les objets et en jouant sur les mots clés **new** et `override`.

Une classe abstraite permet de définir une classe d'objets en faisant abstraction de détails de mise en œuvre. La classe abstraite délègue à ses sous-classes la définition des méthodes ne pouvant être définies à ce niveau d'abstraction. Vous pouvez alors imposer la redéfinition d'une fonction dans chaque classe fille. Voici la syntaxe de la déclaration (notez que cette fonction n'a pas de corps) :

```
abstract void Afficher();
```

INFO

L'équivalent en C++ est une fonction membre virtuelle pure.

Vous déclarez ensuite chaque version "fille" avec le mot clé `override`. Une classe qui comporte une fonction abstraite doit aussi être déclarée abstraite, et toute classe qui en hérite doit redéfinir toutes les méthodes abstraites de sa classe mère, sinon elle est elle-même abstraite.

Les interfaces

Les interfaces font partie des types de références de C#. Une interface ne peut contenir ni code ni données membres — c'est simplement un groupe de noms de méthodes et de signatures. Elle définit ce qu'une classe peut avoir,

mais elle n'implémente aucune fonctionnalité. Elle compte uniquement des fonctions, des propriétés, des événements ou des indexeurs. Elle ne peut comporter de donnée membre, de constructeur ou de destructeur. Tous ses membres sont `public`.

INFO

Une interface C# n'est pas l'équivalent d'une classe abstraite en C++. Une classe abstraite en C++ ne peut pas être instanciée, mais elle peut (et c'est souvent le cas) contenir du code et/ou des données membres. Une interface C# est plus proche d'une interface COM que d'une classe abstraite C++.

L'autre différence majeure est qu'une classe C# ne peut hériter que d'une seule classe (abstraite ou non), mais peut implémenter plusieurs interfaces.

INFO

Les structures ne peuvent pas hériter d'autres structures ou de classes, mais elles peuvent implémenter des interfaces.

Le Code 6.3 présente l'implémentation d'une interface. Il ne comporte aucune difficulté, et vous devriez maintenant être capable de le comprendre uniquement à partir des commentaires ajoutés.

Code 6.3 : Définition d'une interface

```
using System;

//Déclaration de l'interface IForme
public interface IForme
{
  double Aire();
  double Circonference();

  //On peut définir une propriété dans une interface
  int Cotes
  {
    get;
    set;
  }
}
```

```
//Déclaration de la classe Cercle qui hérite de l'interface
public class Cercle : IForme
{
  //données membres
  public double rayon;
  private const float PI=3.1416F;
  private int cotes;

  //Accès à la variable privée cotes via une propriété
  public int Cotes
  {
    get { return cotes; }
    set { cotes = value; }
  }

  //Constructeur
  public Cercle(double r)
  {
    //Initialisation du rayon et du nombre de côtés
    rayon=r;
    Cotes = 1;
  }

  //On définit ensuite la fonction Aire() de l'interface
  public double Aire()
  {
     double aire;
     aire = PI * rayon * rayon;
     //On renvoie l'aire calculée
     return aire;
  }
  //On définit la fonction Circonference() de l'interface
  public double Circonference ()
  {
     return ((double) (2*PI*rayon));
  }
}

//Déclaration de la classe Rectangle qui hérite de
  ➥l'interface
public class Rectangle : IForme
{
  //donnée membre
  public int largeur;
  public int longueur;
  private int cotes;
```

```
//Accès à la variable privée cotes via une propriété
 public int Cotes
 {
   get { return cotes; }
   set { cotes = value; }
 }

 //Constructeur
 public Rectangle(int larg, int lon)
 {
   //Initialisation de la valeur des côtés
   largeur=larg;
   longueur=lon;
   Cotes = 4;
 }

 //On définit ensuite la fonction Aire() de l'interface
 public double Aire()
 {
    return ((double) (largeur*longueur));
 }
 //On définit la fonction Circonference() de l'interface
 public double Circonference ()
 {
    return ((double) (2*largeur*longueur));
 }

}
public class monApplication
{
//Point d'entrée principal du programme
  public static void Main()
  {
    //On crée un objet Cercle de rayon 5
    Cercle monCercle = new Cercle(5);
    //On crée un objet Rectangle
    Rectangle monRectangle = new Rectangle(2,3);

    //On affiche les caractéristiques de chaque objet
    Console.WriteLine("Le cercle :");
    Afficher(monCercle);
    Console.WriteLine("Le rectangle :");
    Afficher(monRectangle);
  }   //Fin de Main()

  //Définition de la fonction Afficher()
  static void Afficher(IForme maForme)
```

```
    {
        Console.WriteLine("Aire: {0}", maForme.Aire());
        Console.WriteLine("Périmètre: {0}",
        ➥maForme.Circonference());
        Console.WriteLine("Côtés: {0}", maForme.Cotes);
    }

}   //Fin de monApplication
```

On obtient :

```
Le cercle :
Aire: 78,5399973392487
Périmètre: 31,4159989356995
Côtés: 1
Le rectangle :
Aire: 6
Périmètre: 12
Côtés: 4
```

La fonction `Afficher()`, définie aux dernières lignes du Code 6.3, mérite quelques commentaires. Cette fonction reçoit en argument une valeur de type `IForme`. Un objet de ce type ne peut exister puisque vous ne pouvez pas instancier (créer un objet de ce type) une classe abstraite, et encore moins une interface. Cependant, cette fonction est polymorphique et elle va accepter tout objet qui implémente l'interface `IForme`. Les fonctions `Aire()`, `Circonference()` et `Cotes()` qu'elle va appeler sont les fonctions définies pour l'objet reçu en argument.

La classe des classes, Object

Nous ne pouvions pas achever ce chapitre consacré à l'héritage sans parler de la classe `Object`, la classe "racine" de toutes les classes de la hiérarchie du .NET Framework.

En C#, tout se traite comme un objet, ce qui signifie que tout type de donnée, qu'il soit intégré ou créé par l'utilisateur, hérite de cette classe. Elle fournit deux fonctions membres particulièrement intéressantes : la fonction `ToString()`, que nous avions redéfinie dans l'exemple du Code 5.2, et la fonction `GetType()`, qui permet d'obtenir le type d'un objet.

L'exemple de code qui suit affiche le calendrier du mois courant. Nous avons ajouté quelques lignes pour illustrer l'utilisation des fonctions `ToString()` et `GetType()`.

Code 6.4 : Affichage du calendrier

```
using System;

public class maDate
{
  //données membres
  public int an;
  public int mois;
  public static int jour=1;

  public maDate()
  {
    //On récupère la date courante
    DateTime aujourdhui=DateTime.Now;
    this.an=aujourdhui.Year;
    this.mois=aujourdhui.Month;
  }


  public String lectureMois(int mois)
  {
    //le tableau tabMois stocke le nom des mois
    String[] tabMois={"Janvier","Février","Mars","Avril",
              "Mai","Juin","Juillet","Août","Septembre",
                       "Octobre","Novembre","Décembre"};

    { return tabMois[mois-1]; }
  }

  public void afficherCalendrier()
  {
    int[] arrjours=new int[35];
    int indexJour;
    DateTime objDate=new DateTime(an,mois,jour);

    //On calcule quel jour tombe le premier du mois
    int indexJourDepart=(int)objDate.DayOfWeek;

    //On affiche le nom du mois et l'année
    Console.WriteLine("{0} - {1}",lectureMois(mois,1),an);
    for(int i=0;i<35;i++)
```

```
    {
        indexJour=(indexJourDepart+i)%35;
        if(i<DateTime.DaysInMonth(an,mois))
        {
          arrjours[indexJour]=(i+1);
        }
        else
        {
          arrjours[indexJour]=0;
        }
      }
      Console.WriteLine(new String('*',64));
      Console.WriteLine("\tDi\tLu\tMa\tMe\tJe\tVe\tSa");
      Console.WriteLine(new String('*',64));

      for(int i=0;i<5;i++)
      {
        Console.Write("\t");
        for(int j=0;j<7;j++)
        {
            if(arrjours[7*i+j]>0)
            {
              Console.Write("{0,-2}\t",arrjours[7*i+j]);
            }
            else
            {
              Console.Write("{0,-2}\t"," ");
            }
        }
        Console.WriteLine();
      }
      Console.WriteLine(new String('*',64));
    }
}

class monApplication
{
  public static void Main()
  {
    maDate calendrier=new maDate();
    Console.Out.WriteLine("Création puis affichage de
    ➥l'objet calendrier de type maDate :");
    calendrier.afficherCalendrier();

    Console.Out.WriteLine("\nRésultat de ToString sur notre
    ➥objet calendrier :{0}",calendrier.ToString());
```

```
        Console.Out.WriteLine("Résultat de GetType sur notre
           ⇨objet calendrier :{0}",calendrier.GetType());
        Console.Out.WriteLine("\nPour terminer, {0} est de type
        ⇨{1}",2002.ToString(), 2002.GetType());
    }
}
```

On obtient le résultat :

```
Création puis affichage de l'objet calendrier de type
maDate :
Janvier - 2002
************************************************************
   Di   Lu   Ma   Me   Je   Ve   Sa
************************************
                1    2    3    4    5
    6    7    8    9   10   11   12
   13   14   15   16   17   18   19
   20   21   22   23   24   25   26
   27   28   29   30   31
************************************************************

Résultat de ToString sur notre objet calendrier :maDate
Résultat de GetType sur notre objet calendrier :maDate

Pour terminer, 2002 est de type System.Int32
```

En lisant ce code, vous devriez maintenant être en mesure de comprendre que **DateTime** fait partie des classes du .NET Framework et que sa propriété **now** permet d'obtenir la date système courante. Vous trouverez aussi une référence à la propriété **DayOfWeek** (jour de la semaine, c'est-à-dire 0 pour dimanche, 1 pour lundi, etc.) et à la fonction membre **DaysInMonth()** qui renvoie, comme son nom l'indique, le nombre de jours du mois transmis en argument.

Examinez les trois dernières lignes de la fonction **Main()**. La fonction **ToString()** héritée de la classe **Object** renvoie une chaîne représentant l'objet courant. Si cette fonction n'est pas redéfinie, elle retourne le nom complet de la classe (ici **maDate** pour le premier appel et **System.int32** pour le second). Vous pouvez la redéfinir pour qu'elle renvoie quelque chose de plus significatif, comme nous l'avions fait au Chapitre 5.

La dernière ligne mérite quelques explications. En C#, tout se traite comme un objet, même les valeurs littérales. C'est pourquoi vous avez le droit d'appeler les membres de la classe `Object` directement sur une valeur littérale (ici 2002). Dans ce cas, la fonction `ToString()` a renvoyé la valeur elle-même et non le nom du type, parce qu'elle est redéfinie dans chaque classe intégrée, comme `int`, pour afficher la valeur. Ce n'est donc pas la version de base qui a été exécutée.

Evénements, délégués et gestion des erreurs

La gestion des erreurs s'appuie sur les exceptions. Les exceptions sont un concept proche des événements, et vous utilisez principalement les délégués avec les événements. Voilà pourquoi nous avons regroupé ces trois éléments dans un même chapitre.

Gestion des événements

Windows est un système d'exploitation fondé sur les événements. A chaque fois qu'un utilisateur clique sur un bouton ou sélectionne une option dans un menu, un événement se produit et le ou les gestionnaires d'événement associés entrent en action. Dans un premier temps, l'événement est émis par une classe, puis il déclenche l'exécution d'un gestionnaire qui lui a été spécialement affecté. Ce principe d'action-réaction permet d'exécuter du code en réponse à une situation.

La création-gestion d'un événement se fait en plusieurs étapes :

- la définition du délégué de l'événement ;
- la création d'une classe pour transmettre les arguments au gestionnaire de l'événement ;
- la déclaration du code de l'événement ;
- la création du gestionnaire ;
- le déclenchement de l'événement.

Le délégué

Un délégué est un type de référence qui définit la signature d'un appel de fonction. Il va filtrer en quelque sorte les appels pour n'exécuter que les fonctions dont la signature présente le bon format.

Voici la syntaxe standard d'un délégué :

```
public delegate typerenvoyé nomDelegue(paramètres);
```

Si vous remplacez le nom du délégué par celui d'une fonction, vous obtenez la signature d'une fonction qui sera acceptée. En exécutant un délégué, vous allez donc exécuter avec cette seule instruction toutes les fonctions dont la signature correspond (n'oubliez pas que la signature comprend le nombre et le type des arguments en entrée, mais aussi le type de la valeur renvoyée).

Le délégué d'un événement a un format particulier :

```
delegate void nomDuGestionnaireDevenement (object source,
xxxEventArgs e);
```

Ce délégué reçoit toujours deux arguments :

- L'objet `source` qui va déclencher l'événement.
- Un objet de type `xxxEventArgs` (classe dérivée de `EventArgs`) contenant les données destinées au gestionnaire de l'événement. En fait, vous pourriez choisir n'importe quel nom, mais `xxxEventArgs` est une convention adoptée par de nombreux programmeurs. Ce nom fait clairement apparaître la filiation avec `EventArgs` et simplifiera la relecture de votre code par vous ou un tiers.

Pour illustrer la manipulation des événements, nous avons repris le programme d'affichage du calendrier présenté au Code 6.4 et nous avons ajouté l'affichage d'un menu pour proposer à l'utilisateur d'afficher un autre mois que le mois courant. Un événement se déclenche si l'utilisateur saisit "q" pour quitter le programme ou un caractère invalide. Ce programme étant assez long, nous avons numéroté les lignes et nous l'avons "découpé" pour simplifier les explications.

Code 7.1 : Création et gestion d'un événement

```
1:using System;
2:
3:public class maDate
4:{
5:  //données membres
6:  public int an;
7:  public int mois;
8:  public static int jour=1;
9:  public String strChoix;
10:
11:  public maDate()
12:  {
13:    //On récupère la date courante
14:    DateTime aujourdhui=DateTime.Now;
15:    this.an=aujourdhui.Year;
16:    this.mois=aujourdhui.Month;
17:  }
18:
19:
20:  public String lectureMois(int mois)
21:  {
22:    //le tableau tabMois stocke le nom des mois
23:    String[] tabMois={"Janvier","Février","Mars","Avril",
24:                "Mai","Juin","Juillet","Août","Septembre",
                    ➥"Octobre","Novembre","Décembre"};
26:
27:    { return tabMois[mois-1]; }
28:  }
29:
30:  public void afficherCalendrier()
31:  {
32:    int[] arrjours=new int[35];
33:    int indexJour;
34:    DateTime objDate=new DateTime(an,mois,jour);
35:
36:    //On calcule quel jour tombe le premier du mois
37:    int indexJourDepart=(int)objDate.DayOfWeek;
38:
39:    //On affiche le nom du mois et l'année
40:    Console.WriteLine("{0} - {1}",lectureMois(mois), an);
41:    for(int i=0;i<35;i++)
42:    {
43:      indexJour=(indexJourDepart+i)%35;
44:      if(i<DateTime.DaysInMonth(an,mois))
45:      {
46:        arrjours[indexJour]=(i+1);
```

```
47:       }
48:       else
49:       {
50:         arrjours[indexJour]=0;
51:       }
52:     }
53:     Console.WriteLine(new String('*',64));
54:     Console.WriteLine("\tDi\tLu\tMa\tMe\tJe\tVe\tSa");
55:     Console.WriteLine(new String('*',64));
56:
57:     for(int i=0;i<5;i++)
58:     {
59:       Console.Write("\t");
60:       for(int j=0;j<7;j++)
61:       {
62:           if(arrjours[7*i+j]>0)
63:           {
64:             Console.Write("{0,-2}\t",arrjours[7*i+j]);
65:           }
66:           else
67:           {
68:             Console.Write("{0,-2}\t"," ");
69:           }
70:       }
71:       Console.WriteLine();
72:     }
73:     Console.WriteLine(new String('*',64));
74:   } //Fin de afficherCalendrier()
75:
```

Vous reconnaissez dans ces premières lignes la classe `maDate` définie au Chapitre 6. Nous introduisons dans cette classe la nouvelle fonction `AfficherMenu()` :

```
76: public string AfficherMenu()
77: {
78:     Console.WriteLine("Menu calendrier");
79:     Console.WriteLine("==================");
80:     Console.WriteLine("1. [> ] Mois suivant");
81:     Console.WriteLine("2. [< ] Mois précédent");
82:     Console.WriteLine("3. [>>] Année suivante");
83:     Console.WriteLine("4. [<<] Année précédente");
84:     Console.WriteLine("5. [q ] Quitter");
85:     Console.Write("Entrez votre choix: ");
86:     strChoix=Console.ReadLine();
```

```
87:     return strChoix; //.ToLower();
88:   }
89:
90:} //Fin de maDate
91:
```

Nous créons maintenant le délégué **choixEventHandler** pour le gestionnaire d'événement. Le premier argument correspond à l'objet qui va déclencher l'événement, le second correspond à l'objet qui contient les informations destinées au gestionnaire. La classe de cet objet, **choixEventArgs**, est définie aux lignes 96 à 104, et elle enregistre la chaîne saisie par l'utilisateur.

```
92://Déclaration du délégué choixEventHandler
93:delegate void choixEventHandler(object source,
   ➥choixEventArgs e);
94:
95://classe qui permettra de transmettre le choix du menu
     ➥au gestionnaire d'événement
96:public class choixEventArgs : EventArgs
97:{
98:  public string choixSaisi;
99:    //Constructeur de la classe
100:  public choixEventArgs(string choixSaisi)
101:  {
102:    this.choixSaisi=choixSaisi;
103:  }
104:}
```

Aux lignes 105 à 126, voici enfin la classe **ControlChoix** qui permettra de déclencher l'événement. Un événement est toujours déclaré en suivant la syntaxe (voir ligne 111) :

```
public event xxxEventHandler nomEvenement;
```

où *xxxEventHandler* est le délégué créé pour l'événement, et *nomEvenement* le nom de l'événement déclaré. La ligne 111 crée l'objet événement **testChoix** avec le délégué **choixEventHandler**. Cet objet permettra d'exécuter le gestionnaire associé à l'événement.

Les lignes 112 à 125 définissent les propriétés pour accéder au **choix_saisi**.

La fonction `set()` commence par vérifier que l'objet `testChoix` qui vient tout juste d'être déclaré comme événement n'est pas nul. Cet objet sera nul si aucun gestionnaire n'a été associé à l'événement (voir plus loin). S'il n'est pas nul, la ligne 120 crée un objet de type `choixEventArgs`, et la valeur saisie par l'utilisateur est ainsi transmise au constructeur de cette classe.

La ligne 121 contient l'appel au délégué avec l'objet événement créé à la ligne 111. Le délégué recherche alors toutes les fonctions qui ont été associées à cet objet. La ligne 122 enregistre la chaîne contenue dans l'objet `choixEventArgs` (c'est-à-dire toujours la saisie de l'utilisateur) dans la donnée membre `choixSaisi`. Si la ligne précédente a appelé un gestionnaire d'événement qui a modifié la donnée, cette ligne enregistre la valeur à jour dans l'objet événement. La fonction `set()` a rempli son rôle.

```
105://classe qui va permettre de déclencher l'événement
106:class ControlChoix
107:{
108:  string choix_saisi;
109:
110:  //création de l'objet événement testChoix avec le
        ➥délégué choixEventHandler
111:  public event choixEventHandler testChoix;
112:  //Propriétés pour l'accès à choix_Saisi
113:  public string choix_Saisi
114:  {
115:    get {return choix_saisi; }
116:    set
117:    {
118:      if(testChoix != null)
119:      {
120:        choixEventArgs args = new choixEventArgs(value);
121:        testChoix(this, args);
122:        choix_saisi = args.choix_Saisi;
123:      }
124:    }
125:  }
126:} //Fin de ControlChoix
127:
```

Voici la classe contenant le point d'entrée du programme. Nous commençons par afficher le calendrier du mois courant, puis un menu proposant divers choix à l'utilisateur.

```
128:class monApplication
129:{
130:
131:  public static bool sortir=false, mauvaisChoix=false;
132:
133:  public static void Main()
134:  {
135:
136:
137:    maDate calendrier=new maDate();
138:    //Création puis affichage de l'objet calendrier de
          ➥type maDate
139:    calendrier.afficherCalendrier();
140:
141:    //On associe le gestionnaire et l'événement:
142:    //en créant un objet ControlChoix:
143:    ControlChoix testeur = new ControlChoix();
144:    //puis en transmettant le nom du gestionnaire
          ➥Controle au délégué choixEventHandler
145:   testeur.testChoix += new choixEventHandler(Controle);
146:
147:    do
148:      {
149:    testeur.choix_Saisi= calendrier.AfficherMenu();
150:     if (sortir==true )
151:        return;
152:      }while(mauvaisChoix==true);
153:
154:     switch(calendrier.strChoix)
155:     {
156:     case ">":
157:          if(calendrier.mois==12)
158:          {
159:            calendrier.an+=1;
160:            calendrier.mois=1;
161:          }
162:          else
163:          {
164:            calendrier.mois+=1;
165:          }
166:          break;
167:        case "<":
168:          if(calendrier.mois==1)
169:          {
170:            calendrier.mois=12;
171:            calendrier.an-=1;
172:          }
```

```
173:          else
174:          {
175:            calendrier.mois-=1;
176:          }
177:          break;
178:        case ">>":
179:          calendrier.an+=1;
180:          break;
181:        case "<<":
182:          calendrier.an-=1;
183:          break;
184:    } //Fin de switch
185:
186:
187:   calendrier.afficherCalendrier();
188: }  //Fin de Main()
189:
```

Voici enfin le gestionnaire de l'événement, `Controle()`, une fonction créée au format du délégué. Cette fonction est appelée à chaque fois que l'événement se produit. Son premier argument correspond à l'objet "déclencheur" de l'événement, et le second à l'objet contenant les données dont il a besoin, c'est-à-dire la chaîne saisie par l'utilisateur. Si cette chaîne est "`q`", il termine le programme en affichant "`Bye Bye!`", et s'il s'agit d'un caractère invalide, il affiche un message d'erreur et invite l'utilisateur à recommencer.

```
190:  static void Controle(object source, choixEventArgs e)
191:  {
192:    if(e.choixSaisi == "q")
193:      {
194:        sortir=true;
195:        Console.WriteLine("Bye Bye!");
196:      }
197:      else
198:      {
199:         if (e.choixSaisi!="<" && e.choixSaisi!="<<" &&
            ➥e.choixSaisi!=">" && e.choixSaisi!=">>")
200:        {
201:         Console.WriteLine("Choix invalide,
            ➥recommencez!");
202:           mauvaisChoix=true;
203:         }
204:      }
205:  } //Fin de Controle()
206:}   //Fin de monApplication
```

L'exécution du programme produit le résultat suivant :

```
Janvier - 2002
*****************************************
   Di   Lu   Ma   Me   Je   Ve   Sa
*****************************************
              1    2    3    4    5
    6    7    8    9   10   11   12
   13   14   15   16   17   18   19
   20   21   22   23   24   25   26
   27   28   29   30   31
*****************************************
Menu calendrier
==================
1. [> ] Mois suivant
2. [< ] Mois précédent
3. [>>] Année suivante
4. [<<] Année précédente
5. [q ] Quitter
Entrez votre choix: 1

 Choix invalide, recommencez!

Menu calendrier
==================
1. [> ] Mois suivant
2. [< ] Mois précédent
3. [>>] Année suivante
4. [<<] Année précédente
5. [q ] Quitter
Entrez votre choix: <

Decembre - 2001
*********************************************
    Di    Lu    Ma    Me    Je    Ve    Sa
*********************************************
    30    31                              1
     2     3     4     5     6     7     8
     9    10    11    12    13    14    15
    16    17    18    19    20    21    22
    23    24    25    26    27    28    29
*********************************************
```

Gestion des exceptions

Quand vous écrivez un programme, vous devez envisager tous les incidents possibles pour les prendre en compte (comme la saisie d'un caractère invalide dans l'exemple précédent), et vous devez même traiter ce qui n'est pas prévisible.

De nombreuses erreurs seront évitées par des contrôles logiques dans le programme, comme le test de la longueur d'une variable, de la présence d'un argument de ligne de commande, ou le contrôle d'une plage de valeurs.

Si un problème surgit malgré toutes les précautions que vous aurez prises, ou à cause d'une erreur dans le programme que le compilateur n'a pu détecter, une exception se produit à l'exécution. Si vous ne voulez pas que vos utilisateurs se fassent injurier en charabia par leur système, vous devez trouver les exceptions et les traiter dans un bloc `try-catch-finally` (il s'agit du même bloc `try-catch-finally` qui existe déjà en C++).

Le runtime (CLR) supporte un modèle de gestion des exceptions fondé sur la création d'objets de type exception et sur des blocs de code protégé. Il crée cet objet exception dès que l'incident se produit. Bien sûr, vous pouvez aussi créer votre propre objet exception en choisissant judicieusement sa classe de base parmi les classes d'exception existantes.

Tous les langages qui utilisent le runtime (et ils sont nombreux) vont gérer les exceptions sur ce même modèle.

Examinons un premier exemple fondé sur les exceptions standard.

Code 7.2 : Gestion des exceptions

```
 1:using System;
 2:
 3:class Erreur
 4:{
 5:  public static void Main()
 6:  {
 7:    int i=0, j=0;
 8:    //La classe Random génère une séquence de
 9:    // nombres pseudo-aléatoires
10:    Random objRan = new Random();
11:
```

```
12:     //Début du bloc try-catch
13:      try {
14:       //NextDouble() renvoie un nombre entre 0.0 et 1.0
15:        double d = objRan.NextDouble();
16:        i = (int)(d * 2.0); // génère 0 ou 1
17:
18:        j = 100/i;
19:      }
20:      catch(DivideByZeroException ex)
21:      {
22:        Console.WriteLine("Voici la représentation
          ➥\"ToString\" de l'erreur:");
23:        Console.WriteLine("Exception: " + ex.ToString());
24:        Console.WriteLine("\nMais vous pourriez afficher
          ➥votre propre message");
25:      }
26:      catch(Exception ex)
27:      {
28:        Console.WriteLine("Exception: " + ex.ToString());
29:      }
30:      finally
31:      {
32:        Console.WriteLine("100 / {0} = {1} ",i, j );
33:      }
34:  }
35:}
```

Pour une fois, nous n'avons pas appelé cette classe `monApplication`, mais `Erreur`.

A la ligne 10, nous créons un objet de type `Random` à partir duquel nous obtiendrons des valeurs.

A la ligne 15, le membre `NextDouble()` de cette classe fournit une valeur de type `double` située entre 0,0 et 1,0 (un autre membre intéressant de cette classe, `Next()`, renvoie un nombre aléatoire).

A la ligne 16, en multipliant le nombre obtenu par 2, vous obtenez 0 ou 1 pour `i` (puisque `=(int)` transforme le type double à droite de l'égalité en type `int`). Ce qui implique que la division à la ligne 18 déclenchera une fois sur deux une exception de type `DivideByZeroException` (division par zéro).

Dans ce cas, le bloc `catch`, à la ligne 20, est exécuté. Ce bloc contient la fonction membre `ToString()` que vous connaissez bien maintenant et qui renvoie le type de l'objet sur lequel elle est appelée. Elle fournit dans le cas d'un type exception la description standard de cette dernière.

A la ligne 26, le second bloc `catch` intercepte toutes les autres exceptions, pour qu'en cas d'incident le programme puisse se terminer de façon élégante.

Pour terminer, le bloc `finally` sera toujours exécuté, même si une exception s'est déclenchée. Il permet d'effectuer des opérations de "nettoyage" avant la fin du programme.

Le tableau qui suit présente un échantillon des exceptions standard de C#.

Tableau 7.1 : Les exceptions standard

Exception	Classe racine de tous les types exception
`SystemException`	Classe de base pour les erreurs d'exécution
`ArgumentException`	Classe de base pour les erreurs liées aux arguments
`InteropException`	Classe de base pour les erreurs extérieures au runtime .NET
`IndexOutOfRangeException`	Déclenchée quand l'index d'un tableau est hors limite
`NullReferenceException`	Déclenchée quand un objet nul est référencé
`InvalidOperationException`	Déclenchée quand une fonction invalide est appelée
`ArgumentNullException`	Déclenchée quand l'argument attendu est nul
`ArgumentOutOfRangeException`	Déclenchée quand un argument ne se trouve pas dans la bonne plage de valeurs
`ComException`	Exception provenant de COM

Voici un deuxième exemple très simple dans lequel nous créons notre propre exception.

Code 7.3 : Créer sa propre exception

```
 1:using System;
 2:
 3:// Déclaration de mon objet exception qui dérive de
 4:// la classe de base Exception
 5:public class monException:Exception
 6: { }
 7:
 8:public class monApplication
 9:{
10:
11:  //Point d'entrée de l'application
12:  public static void Main()
13:  {
14:    try
15:    {
16:       //exécution du membre qui déclenche l'exception
17:       monApplication.TestException();
18:    }
19:     catch(Exception ex)
20:     {
21:       Console.WriteLine(ex);}
22:     }
23:  } //Fin de Main()
24:
25:  //Définition de la fonction statique pour déclencher
       ➥l'exception
26:  public static void TestException()
27:  {
28:    throw new monException("Une exception vient de se
       ➥produire, mon programme a fait ça...");
29:  }
30:
31:}// Fin de monApplication
```

Ma classe `exception` dérive forcément d'une des classes `exception` du .NET Framework.

L'exécution de ce programme donne le résultat suivant :

```
monException: Une exception vient de se produire, mon
➥programme a fait ça...
   at monApplication.TestException() in
   ➥E:\monexception.CS:line 26
   at monApplication.Main() in E:\monexception.CS:line 17
```

Windows Forms

`Windows.Forms` est l'espace du .NET Framework dédié à la programmation d'interfaces utilisateur client Windows.

Avec les Windows Forms, vous allez créer des applications avec interface graphique, c'est-à-dire qui apparaîtront sous la forme de boîtes de dialogue avec des boutons, des zones de texte, etc. Les Windows Forms sont un nouveau style d'application qui s'appuie sur les classes de l'espace de noms `System.Windows.Forms` de la bibliothèque de classes du .NET Framework. Leur utilisation permet d'obtenir des applications plus fiables, plus efficaces et plus cohérentes que celles qui sont fondées sur l'API Win32 ou les MFC (pour ceux d'entre vous qui ont déjà pratiqué), et qui s'exécutent dans l'environnement géré du CLR .NET.

Les Windows Forms sont un excellent point de départ pour étudier la bibliothèque des classes du .NET Framework : ils permettent d'écrire des applications graphiques qui créent des fenêtres ou qui traitent les données saisies par l'utilisateur. Une fois que vous aurez appris à utiliser ces fenêtres, les autres parties du .NET Framework paraîtront assez simples.

Les événements

La programmation des Windows Forms repose en grande partie sur la gestion des événements, thème que nous avons abordé au chapitre précédent. Un événement est une notification émise par une classe lorsque quelque chose se produit. D'autres classes peuvent alors agir en fonction de cette notification. Microsoft Windows est le meilleur exemple de programme orienté événement.

A chaque fois qu'un utilisateur agit dans une boîte de dialogue, par exemple en cliquant sur un bouton, en sélectionnant un menu, ou en saisissant du texte, un événement se déclenche. Le gestionnaire d'événement qui a été associé à cet événement est alors exécuté.

Les Windows Forms présentés dans ce chapitre créent tous des gestionnaires d'événement.

Création d'une fenêtre simple

Le terme de Windows Form est synonyme de fenêtre de "premier niveau". La fenêtre principale d'une application est un Windows Form, et toute autre fenêtre de "premier niveau" de cette application est aussi un Windows Form (ainsi que les boîtes de dialogue). Les classes nécessaires pour ce type de programmation se trouvent dans l'espace de noms `System.Windows.Forms`. Il s'agit en particulier de la classe `Form` qui fournit le comportement des fenêtres, de la classe `Menu` qui représente les menus, et de la classe `Clipboard` qui permet aux applications d'utiliser le Bloc-notes. Cet espace fournit aussi de nombreuses classes pour représenter des contrôles tels que `Button` (bouton), `TextBox` (zone de texte), `ListView` (liste) ou `MonthCalendar` (date). Toute application à base de Windows Forms repose sur une classe dérivée de `Form`, dont l'instanciation produit la fenêtre principale de l'application. Cette classe peut alors faire appel à la fonctionnalité très riche de sa classe mère. Vous voulez connaître les dimensions de la zone client du Windows Form ? Il suffit de consulter la propriété `ClientRectangle` ou `ClientSize`. Vous avez accès à de nombreuses propriétés en lecture ainsi qu'en écriture. Vous pouvez changer, par exemple, le style de bordure d'un Windows Form en redéfinissant simplement sa propriété `BorderStyle`, ou vous pouvez le redimensionner à l'aide de sa propriété `Size` ou `ClientSize`.

Toute application fondée sur les Windows Forms :

- Contient au moins une classe dérivée de la classe `Form`. Cette classe permet alors d'accéder à la fonctionnalité très riche de `Form`.
- Appelle la fonction `Run` de la classe `System.Windows.Forms.Application` dans sa fonction principale `Main()` pour lui transmettre une instance du Windows Form en argument. La fonction `Run` exécute "une boucle d'affichage" qui maintient la fenêtre affichée.

Sans plus tarder, examinons un Windows Form dans sa forme la plus simple : le programme `"Hello, world!"` traditionnel en version graphique !

Code 8.1 : Version graphique de *"Hello, world!"*

```
 1:using System;
 2:using System.Windows.Forms; //espace de noms de la
                                 ➥classe Form
 3:using System.Drawing;       //espace de noms pour la
                                 ➥fonction DrawString
 4:
 5://notre classe MaFenetre dérive de Form:
 6:public class MaFenetre : Form
 7:{
 8:   //Constructeur
 9:   public MaFenetre ()
10:   {
11:     Text = "Mon Windows Form";
12:   }
13:
14:   protected override void OnPaint (PaintEventArgs e)
15:   {
16:     e.Graphics.DrawString ("Hello, world!", Font,
17:         new SolidBrush (Color.Black), ClientRectangle);
18:   }
19:
20:   public static void Main (string[] args)
21:   {
22:     Application.Run (new MaFenetre());
23:   }
24:}
```

***Figure* 8.1**
"Hello, world!" *dans une fenêtre.*

La propriété `Text` définie à la ligne 11 fait partie des nombreuses propriétés disponibles avec `Form`. Nous en utiliserons d'autres dans la suite du chapitre. Elle définit le texte qui apparaît dans la barre de titre de la fenêtre.

La fonction `OnPaint` redéfinie à la ligne 14 est une fonction virtuelle de la classe `Control`. Dans une interface graphique, la plupart des effets sont obtenus en répondant aux messages `WM_PAINT` émis par le système. La méthode `OnPaint` virtuelle est l'équivalent de ce message pour les Windows Forms. En redéfinissant cette fonction, une classe dérivée de `Form` pourra se redessiner, donc se "rafraîchir", en réponse aux messages `WM_PAINT`.

A la ligne 14, cette fonction reçoit en argument un objet de type `PaintEventArgs` qui spécifie le `Graphics` (propriété qui désigne l'unité de sortie, ici l'écran) à utiliser pour dessiner le contrôle et le `ClipRectangle` (propriété qui désigne la zone à redessiner) dans lequel dessiner. Ici, nous avons employé `ClientRectangle`, propriété qui représente la zone client, c'est-à-dire l'intérieur de la fenêtre.

A la ligne 16, la fonction `DrawString` associée au `Graphics` précédent (l'écran) affiche `"Hello, world!"` dans la police désignée par l'objet `Font`, avec la couleur et/ou la texture désignée par un objet de type `Brush` (ici `SolidBrush (Color.Black)`, c'est-à-dire noir), dans la zone indiquée par la propriété `ClientRectangle` (zone client du Windows Form).

La propriété `Color` est disponible avec de nombreux contrôles puisque vous pouvez facilement choisir les couleurs du texte, des boutons, de la zone client d'une fenêtre ou d'un panneau, etc. Pour que vous puissiez laisser libre cours à votre imagination, nous avons inclus la liste des valeurs possibles pour cette propriété dans l'annexe.

A la ligne 22, nous exécutons la fonction `Run`, indispensable pour afficher notre Windows Form et maintenir la boucle de message. Lorsque nous créons l'objet de type `MaFenetre` qu'il reçoit en argument, le constructeur associé est exécuté, et celui-ci redéfinit la barre de titre de la fenêtre (pour y inscrire `"Mon Windows Form"`). Le système envoie alors son message `WM_PAINT` qui déclenche l'exécution de la fonction `OnPaint` redéfinie.

Introduction de menus, contrôles et autres éléments de l'interface graphique

Pour qu'un Windows Form soit réellement opérationnel, il faut le compléter avec des contrôles. Il s'agit des boutons, zones de texte, menus, listes déroulantes, tableaux de données, etc., que vous avez l'habitude de voir dans les fenêtres Windows.

Le mécanisme qui permet aux Windows Forms de répondre aux menus, boutons et autres éléments graphiques est un élément important du modèle de programmation des Windows Forms. Les applications Windows traditionnelles répondent aux messages `WM_COMMAND` et `WM_NOTIFY` par l'intermédiaire des événements de traitement des Windows Forms. Pour C# et les autres langages supportant le CLR, les événements sont des membres de classe aussi importants que les fonctions, les données et les propriétés. Presque toutes les classes de contrôle des Windows Forms déclenchent des événements. Les contrôles de bouton (objets de type `System.Windows.Forms.Button`), par exemple, déclenchent un événement `Click` lorsque l'utilisateur clique dessus. Pour qu'un Windows Form puisse répondre aux clics sur un bouton, celui-ci doit être associé à un gestionnaire de cet événement. Examinons la syntaxe de cette opération en développant notre Windows Form initial. Nous allons le transformer en éditeur d'image.

Menu

Pour commencer, ajoutons un menu d'options proposant une première commande `Quitter` pour fermer l'application. La barre de menus qui apparaît sous la barre de titre d'une fenêtre est un objet de type `System.Windows.Forms.MainMenu`. Vous associez cet objet au Windows Form en lui affectant la propriété `Menu` de ce dernier (héritée de la classe `Form`). Chaque élément de menu est représenté par un objet de type `MenuItem`. Voici le code à ajouter dans le constructeur de `MaFenetre()` :

Code 8.2 : Ajout d'un menu

```
1: // Création d'un menu
2: MainMenu menu = new MainMenu();
3: MenuItem option1 = menu.MenuItems.Add ("&Options");
```

```
 4: option1.MenuItems.Add(new MenuItem ("Q&uitter", new
    ➥EventHandler(OnExit)));
 5:
 6: //On associe le menu au Windows Form
 7: Menu = menu;
 8: }
 9:
10: //Gestionnaire de la commande Quitter
11: private void OnExit(object source, EventArgs e)
12: {
13: Close ();
14: }
```

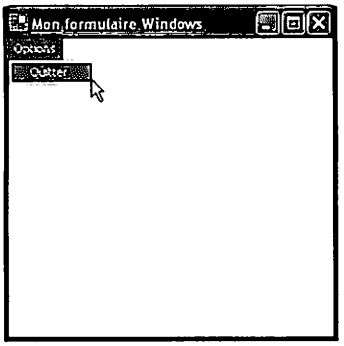


Figure 8.2
Windows Form avec un menu.

A la ligne 3, on ajoute l'élément de menu `&Options` et on stocke une référence à cet élément dans la variable `option1`. `MenuItems` est une collection représentant tous les éléments d'un menu. La fonction `Add()` de cette collection va donc ajouter un élément au menu et renvoyer un autre objet de type `MenuItem`.

A la ligne 4, on ajoute une commande `Quitter` au menu `Options` et on l'associe au gestionnaire d'événement nommé `OnExit`. Ce gestionnaire appelle la fonction `Close()` du Windows Form (toujours héritée de la classe `Form`) qui ferme la fenêtre et termine l'application.

Elément de menu

L'ajout d'un élément de menu est très simple, il suffit d'appeler de nouveau la fonction `Add()` :

```
option1.MenuItems.Add (new MenuItem ("&Ouvrir", new
➥EventHandler (OnOpenImage), Shortcut.Ctrl0));
option1.MenuItems.Add ("-");
```

Cette fois, nous lui avons transmis un troisième paramètre : `Shortcut.Ctrl0`. `Shortcut` est une énumération définie dans l'espace `System.Windows.Forms`. `Ctrl0` est l'élément de cette énumération correspondant à la combinaison de touches `Ctrl+0`. Cela vous permet d'affecter cette combinaison comme raccourci de la commande `Ouvrir` (ce raccourci apparaît dans le menu).

Il ne faut pas oublier de fournir le gestionnaire de la commande `Ouvrir`, c'est-à-dire de l'événement `OnOpenImage` :

Code 8.3 : Gestionnaire de la commande *Ouvrir*

```
1:// Gestionnaire de la commande Ouvrir
2:private void OnOpenImage(object source, EventArgs e)
3:{
4:  //On ouvre une boîte de dialogue standard "Ouvrir"
5:  OpenFileDialog ofd = new OpenFileDialog();
6:
7:   //On définit le contenu du champ Fichiers de type:
8:   ofd.Filter = "Fichiers image(JPEG, GIF, BMP, etc.)|" +
9:      "*.jpg;*.jpeg;*.gif;*.bmp;*.tif;*.tiff;*.png|"    +
10:     "Fichiers JPEG (*.jpg;*.jpeg)|*.jpg;*.jpeg|"      +
11:     "Fichiers GIF (*.gif)|*.gif|"                     +
12:     "Fichiers BMP (*.bmp)|*.bmp|"                     +
13:     "Fichiers TIFF (*.tif;*.tiff)|*.tif;*.tiff|"      +
14:     "Fichiers PNG (*.png)|*.png|"                     +
15:     "Tous les fichiers (*.*)|*.*";
16:
17:  if (monIndex != -1) //si l'utilisateur a sélectionné
     ➥un filtre
18:  ofd.FilterIndex = monIndex; //on enregistre ce choix
19:
20:  if (ofd.ShowDialog() == DialogResult.OK)
21:  {
22:    String nomFichier = ofd.FileName; //on récupère le
       ➥nom du fichier
23:    if (nomFichier.Length != 0)
```

```
24:     {
25:       monIndex = ofd.FilterIndex;
26:       try
27:       {
28:         MonBitmap = new Bitmap(nomFichier);
29:         Text = "Edition de l'image - " + nomFichier;
30:         Invalidate();
31:       }
32:       catch
33:       {
34:         MessageBox.Show(String.Format ("{0} n'est pas " +
35:           "un fichier valide", nomFichier), "Erreur",
36:           MessageBox.OK | MessageBox.IconError);
37:       }
38:     }
39:   }
40:}
```

Il ne faut pas oublier non plus de définir les deux données membres `monIndex` et `MonBitmap` (voir le programme complet) en tête de la classe `MaFenetre` :

```
protected int monIndex = -1;
protected Bitmap monBitmap;
```

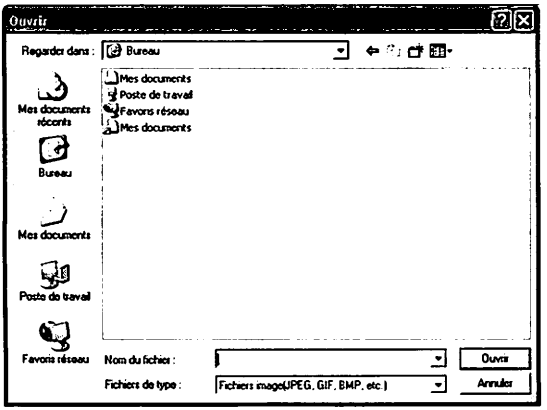


Figure 8.3
Fenêtre Ouvrir.

Ce gestionnaire commence par créer un objet `OpenFileDialog` (une boîte de dialogue standard de Windows) à partir duquel l'utilisateur va sélectionner

l'image qu'il veut afficher. `OpenFileDialog`, qui fait partie de l'espace de noms `System.Windows.Forms`, fournit la fonctionnalité de la fenêtre Windows présentée à la Figure 8.3, ce qui permet d'utiliser cette dernière dans du code managé.

Les lignes 8 à 15 illustrent comment définir le contenu du champ filtre "`Fichiers de type:`", disponible dans le bas de la boîte de dialogue. `filter` est bien sûr une propriété de la classe `OpenFileDialog`, héritée de la classe plus générale `FileDialog`.

A la ligne 18, `FilterIndex` est la propriété de `FileDialog` qui permet d'obtenir ou de définir le "filtre" du type de fichier à afficher.

A la ligne 20, la boîte de dialogue est ouverte avec la fonction `OpenFileDialog.ShowDialog`. Si l'utilisateur saisit un nom de fichier puis clique sur OK, le gestionnaire lit ce nom *via* la propriété `FileName` (ligne 22), puis il ouvre le fichier en créant un objet `Bitmap` (ligne 28).

`Bitmap` est une classe très intéressante de l'espace de noms `System.Drawing`, à partir de laquelle vous pouvez ouvrir une image (presque tous les types sont reconnus) en lui transmettant simplement le nom du fichier.

Le bloc `try-catch` permet de détecter la sélection d'un fichier dont le type n'est pas supporté par notre programme. Cette erreur sera d'abord détectée par le constructeur de `Bitmap` qui va déclencher une exception. Cette exception va entraîner l'exécution du bloc `catch` qui affiche le message d'erreur présenté à la Figure 8.4.

Figure 8.4
Affichage d'un message d'erreur.

A la ligne 34, `MessageBox` (`System.Windows.Forms`) est la classe du .NET Framework qui fournit la fonctionnalité des boîtes de message, et `Show` est la fonction statique (il n'y a pas eu de création d'objet `MessageBox`) d'affichage

de la boîte. Si vous consultez la documentation du .NET Framework, vous trouverez de nombreuses versions de cette fonction. Nous avons utilisé ici la signature `Show(string texte, string légende, MessageBoxButtons bouton)`.

A la ligne 30, `invalidate()` est une fonction de la classe `Form` qui invalide une partie d'un Windows Form (ici la zone client complète), ce qui force la fenêtre à se "réafficher" (avec exécution de la fonction `OnPaint`). La prochaine étape consiste donc à modifier la fonction `OnPaint` pour lui faire afficher le contenu de notre fichier image plutôt que le texte `"Hello, world!"`.

Zone client du Windows Form

L'affichage de l'image fait appel à une autre fonction de la classe `Graphics` : `DrawImage()`. Comme pour la classe `Show()` précédente, il existe de nombreuses versions surchargées de cette fonction pour dessiner une image. Ici, nous avons redéfini la version :

```
DrawImage (Image monImage, float x, float y, float largeur,
float hauteur);
```

Les coordonnées `x` et `y` définissent la position du coin supérieur gauche de l'image dans la zone client du Windows Form, et les propriétés `Width` et `Height` fournissent respectivement la largeur et la hauteur de l'image.

Code 8.4 : Affichage de l'image

```
protected override void OnPaint(PaintEventArgs e)
{
  if (monBitmap != null)
  {
  Graphics g = e.Graphics;
  g.DrawImage (monBitmap, 0, 0, monBitmap.Width,
  ➥monBitmap.Height);
  }
}
```

Quand vous aurez affiché une de vos images dans cet éditeur, vous constaterez qu'une partie seulement de cette image est visible si sa taille est supérieure à celle de la zone client de la fenêtre. Il faut donc ajouter des barres de défilement, ce qui représente l'ultime étape de la construction de ce programme.

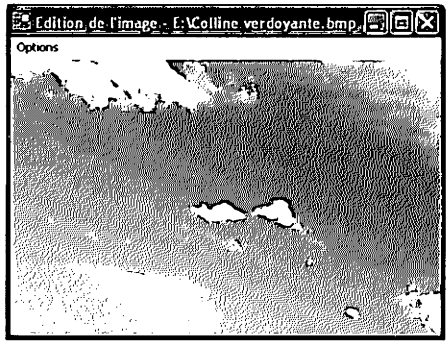


***Figure* 8.5**
L'image est ouverte dans le Windows Form.

Barres de défilement

La classe `Form` hérite de la propriété booléenne `AutoScroll` qui permet d'activer ou de désactiver la fonctionnalité des barres de défilement. Quand cette propriété est définie en `true` (vrai), une barre apparaît automatiquement en vertical si la hauteur de l'image dépasse celle de la fenêtre et/ou en horizontal si la largeur de l'image est supérieure à celle de la fenêtre, à condition d'avoir bien enregistré dans la propriété `AutoScrollMinSize` les dimensions de la zone d'affichage. Nous allons définir ces deux propriétés juste après la création de l'objet `Bitmap` :

```
AutoScroll = true;
AutoScrollMinSize = monBitmap.Size;
```

Il faut aussi modifier l'appel de `DrawImage` pour que la portion d'image visible corresponde à la position courante des curseurs de défilement :

```
g.DrawImage (monBitmap, AutoScrollPosition.X,
➥AutoScrollPosition.Y, monBitmap.Width, monBitmap.Height);
```

Pour terminer, voici le code du programme complet, suivi de la Figure 8.6 présentant la version finale du Windows Form.

Code 8.5 : Programme complet de l'éditeur d'image

```
using System;
using System.Windows.Forms;
using System.Drawing;

//notre classe MaFenetre dérive de Form:
public class MaFenetre : Form
{
  //Données membres
  protected int monIndex = -1;
  protected Bitmap monBitmap;

  //Constructeur
  public MaFenetre ()
  {
    Text = "Mon Windows Form";

    // Création d'un menu
    MainMenu menu = new MainMenu();
    MenuItem option1 = menu.MenuItems.Add ("&Options");
    option1.MenuItems.Add (new MenuItem ("Q&uitter",
                           new EventHandler(OnExit)));
    option1.MenuItems.Add (new MenuItem ("&Ouvrir",
                           new EventHandler(OnOpenImage),
                           Shortcut.CtrlO));
    option1.MenuItems.Add ("-");

    //On associe le menu au Windows Form
    Menu = menu;
  }

  // Gestionnaire de la commande Ouvrir
  private void OnOpenImage(object source, EventArgs e)
  {

    OpenFileDialog ofd = new OpenFileDialog();

    ofd.Filter = "Fichiers image(JPEG, GIF, BMP, etc.)|" +
           "*.jpg;*.jpeg;*.gif;*.bmp;*.tif;*.tiff;*.png|"+
             "Fichiers JPEG (*.jpg;*.jpeg)|*.jpg;*.jpeg|"+
                           "Fichiers GIF (*.gif)|*.gif|"+
                           "Fichiers BMP (*.bmp)|*.bmp|"+
             "Fichiers TIFF (*.tif;*.tiff)|*.tif;*.tiff|"+
                           "Fichiers PNG (*.png)|*.png|"+
                           "Tous les fichiers (*.*)|*.*";

  if (monIndex != -1)
  ofd.FilterIndex = monIndex;
```

```
    if (ofd.ShowDialog () == DialogResult.OK)
    {
      String nomFichier = ofd.FileName;
      if (nomFichier.Length != 0)
      {
        monIndex = ofd.FilterIndex;
        try
        {
          monBitmap = new Bitmap(nomFichier);
           Text = "Edition de l'image - " + nomFichier;
          AutoScroll = true;
          AutoScrollMinSize = monBitmap.Size;
          Invalidate();
        }
        catch
        {
           MessageBox.Show(String.Format("{0} n'est pas " +
                 "un fichier valide", nomFichier), "Erreur",
                  MessageBoxButtons.OK);
        }
      }
    }
  }

    //Gestionnaire de la commande Quitter
    private void OnExit (object source, EventArgs e)
    {
      Close ();
    }

    //Fonction d'affichage du Windows Form
    protected override void OnPaint(PaintEventArgs e)
    {
      if (monBitmap != null)
      {
        Graphics g = e.Graphics;
        g.DrawImage(monBitmap, AutoScrollPosition.X,
                  AutoScrollPosition.Y, monBitmap.Width,
                  monBitmap.Height);
      }
    }

    public static void Main (string[] args)
    {
      Application.Run (new MaFenetre());
    }
}
```

***Figure* 8.6**
Version finale de l'éditeur d'image.

Les contrôles Button, Panel et TextBox

Pour illustrer la conception et la gestion d'autres contrôles dans une fenêtre, nous allons construire la calculatrice présentée à la Figure 8.7.

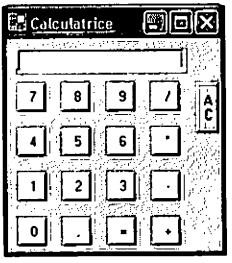


***Figure* 8.7**
Notre calculatrice.

Toutes les déclarations et définitions de boutons apparaissent en caractères gras dans le Code 8.6.

La première étape de la création d'un bouton consiste à créer un objet de type `Button` :

- Déclaration du tableau de boutons `b` (ligne 9). Ce tableau permet ensuite de définir les boutons correspondant aux chiffres de la calculatrice dans une boucle `for` aux lignes 53 à 61.
- Déclaration des boutons `bVirgule`, `bPlus`, `bSoustr`, `bMul`, `bDiv`, `bEgal` et `bEfface` correspondant aux différents boutons d'une calculatrice standard (ligne 10).

Il faut ensuite définir les propriétés de chaque bouton. Le Tableau 8.1 présente celles qui sont les plus utilisées. Vous trouverez la liste complète de ces propriétés dans la documentation du .NET Framework.

Tableau 8.1 : Une sélection de propriétés de bouton

Propriété	Description
`BackColor`	Couleur d'arrière-plan (utilisée à la ligne 58)
`BackgroundImage`	Image qui va s'afficher sur l'arrière-plan du bouton
`Bottom`	Distance entre le bas du bouton et le bord supérieur de la zone client de son conteneur
`Enabled`	Valeur indiquant si le bouton est activé
`Height`	Hauteur du bouton
`Image`	Image qui va s'afficher sur le bouton
`Left`	Position du bord gauche du bouton
`Location`	Position du coin supérieur gauche du bouton par rapport au coin supérieur gauche de son conteneur (utilisée aux lignes 62 à 73)
`Right`	Position du bord droit du bouton
`Size`	Largeur et hauteur du bouton (utilisée à la ligne 57)

Tableau 8.1 : Une sélection de propriétés de bouton *(suite)*

Propriété	Description
`Text`	Intitulé du bouton (utilisée à la ligne 56)
`TextAlign`	Position de l'intitulé sur le bouton
`Top`	Position du bord supérieur du bouton
`Visible`	Indique si le bouton est visible
`Width`	Largeur du bouton

Un bouton déclenche toujours une action, vous devez donc lui associer un ou plusieurs événements. Comme nous l'avons expliqué au Chapitre 7, vous commencez par définir la fonction qui sera appelée lorsque l'événement sera détecté :

- La méthode destinée à gérer l'événement **`ClickBouton`** (clic sur un bouton) est définie à la ligne 133.
- La méthode destinée à gérer l'événement **`Operation`** (clic sur un bouton d'opération) est définie à la ligne 154.
- La méthode destinée à gérer l'événement **`Effacer`** (clic sur le bouton `"AC"`) est définie à la ligne 167.
- La méthode destinée à gérer l'événement **`Egal`** (clic sur le bouton `"="`) est définie à la ligne 172.

Pour activer l'événement, il faut l'associer au délégué approprié. **`System.EventHandler`** est un objet délégué lié à tous les événements de fenêtre. En associant vos gestionnaires à ce délégué, vous obtiendrez le comportement recherché. La syntaxe est :

```
nomControle.Event += new EventHandler(this.nomGestionnaire)
```

nomControle correspond ici au nom du bouton, `Event` correspond à l'événement, et *nomGestionnaire* est le nom de la fonction de gestion. Vous retrouvez cette syntaxe :

- aux lignes 59 et 80, pour associer respectivement le clic sur un des boutons de chiffre ou sur le bouton de la virgule avec le gestionnaire `ClickBouton` ;
- aux lignes 87, 94, 101 et 108, pour associer respectivement le clic sur le bouton de l'addition, de la soustraction, de la multiplication et de la division avec le gestionnaire `Operation` ;
- à la ligne 115, pour associer le clic sur le bouton égale avec le gestionnaire `Egal` ;
- à la ligne 122 pour associer le clic sur le bouton effacer avec le gestionnaire `Effacer`.

Notre calculatrice n'a pas que des boutons, elle contient aussi une zone de texte pour saisir les nombres (`TextBox` déclaré à la ligne 12) et un contrôle de type `Panel` (déclaré à la ligne 11). Il existe quantité d'autres contrôles disponibles, et chacun d'eux possède de nombreuses fonctions membres et propriétés. Nous vous conseillons d'examiner en particulier la hiérarchie des classes dérivées de `Control` dans la documentation du .NET Framework.

Code 8.6 : La calculatrice

```
 1:using System;
 2:using System.Windows.Forms;
 3:using System.Drawing;
 4:
 5:public class MaFenetre:Form
 6:{
 7:
 8:  //Données membres
 9:  Button[] b = new Button[10];
10:  Button bVirgule,bPlus,bSoustr,bMul,bDiv,bEgal,bEfface;
11:  Panel panCalc;
12:  TextBox txtCalc;
13:
14:  Double resultat;
15:  Double tampon;
```

```
16:  bool onEfface,nombre1;
17:  String nomOper;
18:
19:  //Constructeur de mon objet MaFenetre
20:  public MaFenetre()
21:  {
```

On regroupe dans un premier bloc `try` toutes les instructions de "construction" de la fenêtre pour détecter une exception à ce niveau :

```
22:     try
23:     {
24:    this.Text="Calculatrice";
25:    panCalc = new Panel();
26:    txtCalc = new TextBox();
27:
28:    txtCalc.Location = new Point(10,10);
29:    txtCalc.Size=new Size(150,10);
30:    txtCalc.ReadOnly=true;
31:    txtCalc.RightToLeft=RightToLeft.Yes;
32:    panCalc.Size=new Size(200,200);
33:    panCalc.BackColor=Color.Orange;
34:    panCalc.Controls.Add(txtCalc);
35:    AjouterBoutons(panCalc);
36:    this.Size=new Size(200,225);
37:    this.Controls.Add(panCalc);
38:
39:    resultat=0;
40:    tampon=0;
41:    nombre1=true;
42:    onEfface=false;
43:    nomOper="=";
44:      }
```

Ce bloc de construction commence par définir le texte qui apparaît dans la barre de titre de la fenêtre (ligne 24), puis il crée un contrôle `Panel` et un contrôle `TextBox`.

`Panel` permet de regrouper d'autres contrôles. Vous allez typiquement l'utiliser pour diviser une fenêtre par fonctions. Ce regroupement logique est uniquement visuel pour l'utilisateur, mais il permet aussi d'effectuer facilement des opérations sur l'ensemble des contrôles en phase de conception.

En effet, lorsque vous déplacez le contrôle **Panel**, vous déplacez aussi tous les contrôles qu'il contient, ou lorsque vous définissez sa propriété **Enabled** à **false**, vous désactivez aussi l'ensemble des contrôles contenus. **Panel** est analogue au contrôle **GroupBox** mais ils diffèrent par le fait que le premier peut posséder des barres de défilement et que le second peut posséder un intitulé. Pour faire apparaître les barres de défilement, il faut définir la propriété **AutoScroll** en **true**. Comme pour le contrôle bouton, vous pouvez aussi régler son apparence avec les propriétés **BackColor**, **BackgroundImage** et **BorderStyle**. Cette dernière propriété indique si le contrôle ne possède pas de bordure visible (**none**), si cette bordure est une ligne continue (**FixedSingle**) ou une ligne ombrée.

Le contrôle **TextBox** permet d'obtenir des données en entrée de l'utilisateur ou d'afficher du texte. Ce contrôle est généralement défini pour du texte modifiable, mais pour notre calculatrice il est défini en lecture seulement (propriété **ReadOnly** à la ligne 30). A la ligne 31, la propriété **RightToLeft** va faire apparaître les chiffres saisis ou le résultat des calculs à droite de la zone de texte plutôt qu'à gauche.

A la ligne 34, vous introduisez le contrôle **TexBox txtCalc** dans le contrôle **Panel**.

A la ligne 37, vous ajoutez le contrôle **Panel** dans le Windows Form.

Pour terminer, ce constructeur initialise quelques variables : **nomOper** (pour le nom du bouton opération sélectionné), **onEfface** (booléen pour supprimer l'affichage de la zone de texte), **nombre1** (booléen pour différencier le premier et le second opérande de l'opération), enfin **tampon** et **resultat**, des variables intermédiaires.

```
45:       catch (Exception e)
46:       {
47:           Console.WriteLine("erreur:" +  e.StackTrace);
48:       }
49:   } //Fin de maFenêtre
50:
```

La propriété `StackTrace` utilisée à la ligne 47 fournit des détails sur l'élément à l'origine de l'exception, et peut même indiquer le nom du fichier source avec le numéro de la ligne de programme en cause si ces informations sont disponibles (voir Figure 8.8).

Comme son nom l'indique, la fonction membre `AjouterBoutons()` définie aux lignes 51 à 131 :

- Crée les boutons des chiffres de zéro à neuf (dans la boucle `for`), les positionne dans la fenêtre (lignes 62 à 73), puis les ajoute au contrôle `Panel` dont le nom a été transmis en argument à la fonction `AjouterBoutons()`.
- Crée les autres boutons de commande en associant à chaque fois l'événement `Click` sur ce bouton au gestionnaire approprié.

```
51:  private void AjouterBoutons(Panel p)
52:  {
53:    for (int i=0;i<=9;i++)
54:    {
55:      b[i]=new Button();
56:      b[i].Text=Convert.ToString(i);
57:      b[i].Size=new Size(25,25);
58:      b[i].BackColor=Color.Lavender;
59:      b[i].Click+=new EventHandler(ClickBouton);
60:      p.Controls.Add(b[i]);
61:    }
62:    b[0].Location=new Point(10,160);
63:    b[1].Location=new Point(10,120);
64:    b[4].Location=new Point(10,80);
65:    b[7].Location=new Point(10,40);
66:
67:    b[2].Location=new Point(50,120);
68:    b[5].Location=new Point(50,80);
69:    b[8].Location=new Point(50,40);
70:
71:    b[3].Location=new Point(90,120);
72:    b[6].Location=new Point(90,80);
73:    b[9].Location=new Point(90,40);
74:
75:    bVirgule=new Button();
76:    bVirgule.Size=new Size(25,25);
77:    bVirgule.Location=new Point(50,160);
78:    bVirgule.BackColor=Color.Lavender;
```

```
79:     bVirgule.Text=",";
80:     bVirgule.Click+=new EventHandler(ClickBouton);
81:
82:     bPlus=new Button();
83:     bPlus.Size=new Size(25,25);
84:     bPlus.Location=new Point(130,160);
85:     bPlus.BackColor=Color.Lavender;
86:     bPlus.Text="+";
87:     bPlus.Click+=new EventHandler(Operation);
88:
89:     bSoustr=new Button();
90:     bSoustr.Size=new Size(25,25);
91:     bSoustr.Location=new Point(130,120);
92:     bSoustr.BackColor=Color.Lavender;
93:     bSoustr.Text="-";
94:     bSoustr.Click+=new EventHandler(Operation);
95:
96:     bMul=new Button();
97:     bMul.Size=new Size(25,25);
98:     bMul.Location=new Point(130,80);
99:     bMul.BackColor=Color.Lavender;
100:    bMul.Text="*";
101:    bMul.Click+=new EventHandler(Operation);
102:
103:    bDiv=new Button();
104:    bDiv.Size=new Size(25,25);
105:    bDiv.Location=new Point(130,40);
106:    bDiv.BackColor=Color.Lavender;
107:    bDiv.Text="/";
108:    bDiv.Click+=new EventHandler(Operation);
109:
110:    bEgal=new Button();
111:    bEgal.Size=new Size(25,25);
112:    bEgal.Location=new Point(90,160);
113:    bEgal.BackColor=Color.Lavender;
114:    bEgal.Text="=";
115:    bEgal.Click+=new EventHandler(Egal);
116:
117:    bEfface=new Button();
118:    bEfface.Size=new Size(20,45);
119:    bEfface.Location=new Point(170,40);
120:    bEfface.BackColor=Color.Orange;
121:    bEfface.Text="AC";
122:    bEfface.Click+=new EventHandler(Effacer);
123:
```

On ajoute maintenant tous ces boutons au contrôle **Panel** dont le nom a été transmis en argument à la fonction **AjouterBoutons()**.

```
124:   p.Controls.Add(bVirgule);
125:   p.Controls.Add(bPlus);
126:   p.Controls.Add(bSoustr);
127:   p.Controls.Add(bMul);
128:   p.Controls.Add(bDiv);
129:   p.Controls.Add(bEgal);
130:   p.Controls.Add(bEfface);
131: }  //Fin de AjouterBoutons()
132:
```

La méthode **ClickBouton()** va gérer l'événement **Click** sur un bouton :

- Si **onEfface** est vrai, elle efface la zone de texte.
- Elle affiche la valeur (propriété **Text**) du bouton sélectionné (un chiffre ou une virgule) à la suite de ce qui apparaît déjà dans la zone de texte (lignes 138 et 139).
- Si cette zone de texte comporte seulement une virgule (l'utilisateur a commencé à saisir un nombre décimal), elle ajoute un zéro devant (ligne 143).
- Elle enregistre dans la variable **tampon** le contenu de la zone de texte après avoir converti la représentation chaîne du nombre inscrit dans cette dernière en type **double** avec **Convert.ToDouble**. La classe **Convert** propose de nombreuses autres fonctions de conversion d'un type dans un autre.

```
133: private void ClickBouton(object obj,EventArgs ea)
134: {
135:   if(onEfface)
136:     txtCalc.Text="";
137:
138:   Button boutonTemp=(Button)obj;
139:   txtCalc.Text+= boutonTemp.Text;
140:
141:    if (txtCalc.Text==",")
142:     txtCalc.Text="0,";
143:
144:   tampon=Convert.ToDouble(txtCalc.Text);
145:
146:   onEfface=false;
147: }
148:/*************FONCTION PRINCIPALE*******************/
```

```
149: private static void Main()
150: {
151:   Application.Run(new MaFenetre());
152: }
153:
```

La méthode `Operation()` va gérer l'événement `Click` sur un bouton d'opération :

- On affecte à `tmp` l'identité du bouton sélectionné.
- On affecte à `nomOper` la propriété `Text` du bouton, c'est-à-dire un des signes mathématiques.
- Si `nombre1` est vrai (il manque le second opérande pour effectuer l'opération), on enregistre simplement la valeur double de la zone de texte dans la variable `resultat`, puis on définit `nombre1` en `false` et `onEfface` en `true` pour que la zone de texte soit purgée et prête à accueillir le prochain opérande.

```
154: private void Operation(object obj,EventArgs ea)
155: {
156:   Button tmp=(Button)obj;
157:   nomOper=tmp.Text;
158:   if (nombre1)
159:     resultat=tampon;
160:   else
161:     calc();
162:
163:   nombre1=false;
164:   onEfface=true;
165: }
166: //Méthode destinée à gérer l'événement Click sur effacer
167: private void Effacer(object obj,EventArgs ea)
168: {
169:   clear();
170: }
171: //Méthode destinée à gérer l'événement Click sur le
       ➥signe égale
172: private void Egal(object obj,EventArgs ea)
173: {
174:   calc();
175: }
176: //Méthode de calcul
177: private void calc()
178: {
179:   switch(nomOper)
```

NOUVEAU

Le code-behind est une nouvelle fonctionnalité d'ASP.NET qui permet aux développeurs de bien marquer la séparation entre la présentation HTML et le code écrit en C#, Visual Basic.NET, ou d'autres langages .NET. Ce code permet de gérer le dialogue avec l'utilisateur, de valider les champs saisis, etc. En étant isolé dans son propre fichier, il peut facilement être réutilisé dans d'autres projets, et la page Web est plus facile à lire. Cela permet aussi de distribuer la page avec la partie code compilée pour protéger le code source.

Une application ASP.NET très simple

Eh oui, voici notre programme `"Hello, world!"` (encore lui !) à la sauce ASP.

Code 9.1 : *"Hello, world!"* dans une page *.aspx*

```
<%@ Page Language="c#" ClassName="Hello" %>
<HTML>
<HEAD>
  <script runat="server">
    protected void ClickmonBouton(object source,
    ⇨EventArgs e)
    {
       monLabel.Text="Hello, world!";
    }
  </script>
</HEAD>

<BODY>
  <H3>Notre premier Web Form</H3>

  <FORM runat="server">

      <asp:Button id="monBouton"
                  runat="server"
                  Text="Mon bouton"
                  onclick="ClickmonBouton" />
<br><br>

      <asp:Label id="monLabel"
                 runat="server"/>
  </FORM> <!-- Toutes les balises doivent posséder leur
  ⇨balise de fin -->
</BODY>
</HTML>
```

```
149: private static void Main()
150: {
151:   Application.Run(new MaFenetre());
152: }
153:
```

La méthode `Operation()` va gérer l'événement `Click` sur un bouton d'opération :

- On affecte à `tmp` l'identité du bouton sélectionné.
- On affecte à `nomOper` la propriété `Text` du bouton, c'est-à-dire un des signes mathématiques.
- Si `nombre1` est vrai (il manque le second opérande pour effectuer l'opération), on enregistre simplement la valeur double de la zone de texte dans la variable `resultat`, puis on définit `nombre1` en `false` et `onEfface` en `true` pour que la zone de texte soit purgée et prête à accueillir le prochain opérande.

```
154: private void Operation(object obj,EventArgs ea)
155: {
156:   Button tmp=(Button)obj;
157:   nomOper=tmp.Text;
158:   if (nombre1)
159:     resultat=tampon;
160:   else
161:     calc();
162:
163:   nombre1=false;
164:   onEfface=true;
165: }
166: //Méthode destinée à gérer l'événement Click sur effacer
167: private void Effacer(object obj,EventArgs ea)
168: {
169:   clear();
170: }
171: //Méthode destinée à gérer l'événement Click sur le
       ➥signe égale
172: private void Egal(object obj,EventArgs ea)
173: {
174:   calc();
175: }
176: //Méthode de calcul
177: private void calc()
178: {
179:   switch(nomOper)
```

```
180:    {
181:      case "+":
182:        resultat+=tampon;
183:        break;
184:      case "-":
185:        resultat-=tampon;
186:        break;
187:      case "*":
188:        resultat*=tampon;
189:        break;
190:      case "/":
191:        resultat/=tampon;
192:        break;
193:    }
194:
195:    nomOper="=";
196:    nombre1=true;
197:    txtCalc.Text=Convert.ToString(resultat);
198:    tampon=resultat;
199: }
200:
201: private void clear()
202: {
203:    resultat=0;
204:    tampon=0;
205:    nombre1=true;
206:    txtCalc.Text="";
207:    txtCalc.Focus();
208: }
209:}
```

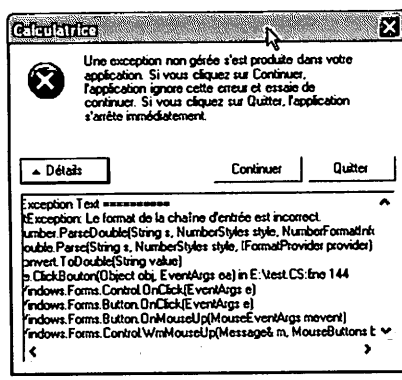


Figure 8.8
Voici la boîte de dialogue obtenue lorsque vous utilisez la propriété StackTrace *et qu'une exception se produit.*

Formulaires Web (Web Forms)

Si vous voulez programmer pour le Web, vous allez naturellement rechercher un support multi-plate-forme qui n'existe pas pour les Windows Forms étudiés précédemment. Vous devrez alors vous tourner vers les standards Web tels que HTML et XML, et votre application s'ouvrira dans la fenêtre d'un navigateur Internet.

Ce chapitre très court n'a d'autre ambition que de vous donner un aperçu des possibilités offertes par les formulaires Web. Pour tester l'exemple du Code 9.1, vous devez être en mesure d'utiliser les services d'un serveur Web supportant ASP.NET, comme IIS (*Internet Information Services*), correctement configuré. Nous supposerons aussi que vous connaissez le langage HTML puisque nous allons principalement nous intéresser à la relation ASP.NET/C#.

Une application fondée sur les formulaires Web se présente sous la forme d'une application ASP.NET. Elle sera donc enregistrée dans un fichier d'extension **.aspx**. Un formulaire Web est composé d'éléments visuels construits à partir de balises HTML et de code. Ce code peut être écrit de deux façons : directement dans le fichier d'extension **.aspx**, comme dans le Code 9.1, ou dans un module de *code-behind*. Bien qu'il soit possible d'écrire l'application complète dans le fichier **.aspx** tout en bénéficiant des avantages du code compilé et des autres améliorations de la plate-forme .NET, il est recommandé d'isoler le code dans les modules de code-behind, c'est-à-dire dans un fichier **.cs** classique distinct. Le fichier **.aspx** joue le rôle de conteneur pour les éléments visuels HTML et les contrôles Web Forms.

NOUVEAU

Le code-behind est une nouvelle fonctionnalité d'ASP.NET qui permet aux développeurs de bien marquer la séparation entre la présentation HTML et le code écrit en C#, Visual Basic.NET, ou d'autres langages .NET. Ce code permet de gérer le dialogue avec l'utilisateur, de valider les champs saisis, etc. En étant isolé dans son propre fichier, il peut facilement être réutilisé dans d'autres projets, et la page Web est plus facile à lire. Cela permet aussi de distribuer la page avec la partie code compilée pour protéger le code source.

Une application ASP.NET très simple

Eh oui, voici notre programme `"Hello, world!"` (encore lui !) à la sauce ASP.

Code 9.1 : *"Hello, world!"* dans une page *.aspx*

```
<%@ Page Language="c#" ClassName="Hello" %>
<HTML>
<HEAD>
  <script runat="server">
    protected void ClickmonBouton(object source,
    ➥EventArgs e)
    {
       monLabel.Text="Hello, world!";
    }
  </script>
</HEAD>

<BODY>
  <H3>Notre premier Web Form</H3>

  <FORM runat="server">

      <asp:Button id="monBouton"
                  runat="server"
                  Text="Mon bouton"
                  onclick="ClickmonBouton" />
<br><br>

      <asp:Label id="monLabel"
                 runat="server"/>
  </FORM> <!-- Toutes les balises doivent posséder leur
    ➥balise de fin -->
</BODY>
</HTML>
```

Ce fichier n'a pas besoin d'être compilé, il suffit de le copier sur le répertoire racine du serveur Web. Si vous travaillez avec IIS et si vous avez conservé les paramètres par défaut, le répertoire racine de ce serveur est C:\inetpub\wwwroot\. Si vous avez copié la page dans ce répertoire, saisissez http://localhost/hello.aspx dans la zone Adresse de votre navigateur pour obtenir l'écran présenté à la Figure 9.1.

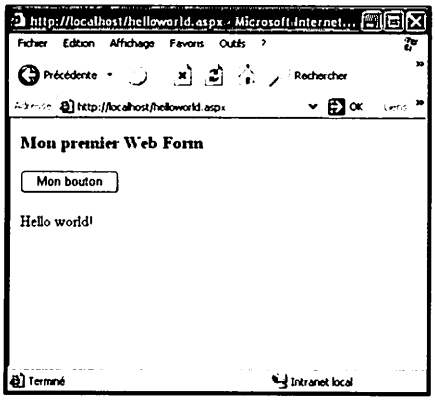


Figure 9.1
Hello, world! *version* ASP.NET *quand vous cliquez sur le bouton.*

Lorsque vous demandez l'affichage du fichier `helloworld.aspx` dans votre navigateur, le serveur sait qu'il s'agit d'une page ASP.NET grâce à l'extension .aspx, et il l'exécute. Il va donc interpréter tout le code compris entre les deux balises <% et %> comme des directives ASP.NET, et ce code lui apprend (à la première ligne) que le langage de script utilisé est C#. C'est parce que cette page est exécutée par le serveur et non par le navigateur du client que vous allez pouvoir y introduire des traitements, même très complexes. Au final, le serveur va générer une page standard au format XHTML que n'importe quel navigateur sera capable d'afficher (vérifiez-le en affichant le source de la page dans votre navigateur).

Ce code présente quelques particularités par rapport à une page HTML standard comportant une partie script :

- Les balises HTML des lignes 4 et 15 contiennent `"runat="server""`. Ce code signale au serveur qu'il doit convertir ce contrôle de formulaire HTML standard vers le contrôle serveur HTML équivalent, ce qui vous permet de le manipuler. Le serveur ignorera tous les autres contrôles HTML et les transmettra tels quels au navigateur. Les contrôles serveur HTML du .NET Framework sont présentés à la section suivante.
- Aux lignes 5 à 8, vous reconnaissez la définition d'un événement. Celui-ci se déclenche quand l'utilisateur clique sur le bouton (ligne 20), et il définit la propriété `Text` du contrôle `Label` en `"Hello, world!"` (ligne 7).
- Enfin, les balises commençant aux lignes 17 et 23 débutent par `asp:` et ne ressemblent pas à des balises HTML standard. Il s'agit de la seconde catégorie de contrôles utilisables avec les pages ASP.NET, les contrôles serveur Web présentés plus loin.

Les contrôles serveur HTML

Les éléments HTML d'un fichier ASP.NET sont considérés par défaut comme du texte littéral et ne sont pas accessibles en programmation pour les développeurs de pages. En ajoutant l'attribut `runat="server"`, vous indiquez au serveur que cet élément HTML doit au contraire être analysé et traité comme un contrôle de serveur.

L'attribut `id` unique permet de faire référence au contrôle dans le code. Les contrôles serveur HTML doivent être inclus dans un contrôle `<FORM>` possédant l'attribut `runat="server"`. ASP.NET n'exige pas que le code HTML soit parfaitement rédigé, mais il faut absolument que chaque balise soit correctement fermée par la balise `</...>` associée, sans chevauchement dans les imbrications.

C'est l'espace de noms `System.Web.UI.HtmlControls` qui fournit les classes à partir desquelles sont créés les contrôles serveur HTML d'une page Web. Chacun de ces contrôles s'exécute sur le serveur et produit dans la page finale la balise HTML standard équivalente, supportée par tous les navigateurs.

Vous avez ainsi la possibilité de contrôler dans votre code les éléments HTML de la page Web.

Tableau 9.1 : Les contrôles serveur HTML

Contrôle .NET	Description ou balise HTML équivalente au contrôle
`HtmlAnchor`	`<a>`
`HtmlButton`	`<button>`
`HtmlForm`	`<form>`
`HtmlGenericControl`	Toute balise non représentée par un des contrôles serveur HTML existants
`HtmlImage`	`<img>`
`HtmlInputButton`	`<input type= button>`, `<input type= submit>` et `<input type= reset>`
`HtmlInputCheckBox`	`<input type= checkbox>`
`HtmlInputControl`	Toutes les balises d'entrée telles que `<input type= text>`, `<input type= submit>` ou `<input type= file>`
`HtmlInputFile`	`<input type= file>`
`HtmlInputHidden`	`<input type= hidden>`
`HtmlInputImage`	`<input type= image>`
`HtmlInputRadioButton`	`<input type= radio>`
`HtmlInputText`	`<input type= text>` et `<input type= password>`
`HtmlSelect`	`<select>`
`HtmlTable`	`<table>`
`HtmlTableCell`	`<td>` et `<th>` incluses dans un contrôle `HtmlTableRow`
`HtmlTableRow`	`<tr>` incluse dans un contrôle `HtmlTable`
`HtmlTextArea`	`<textarea>`

Vous trouverez tous les membres disponibles avec ces classes en consultant la documentation du .NET Framework.

Les contrôles serveur Web

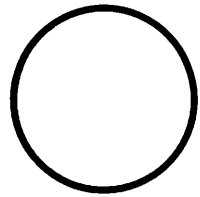
Les contrôles serveur Web sont très proches des contrôles de Windows Form. Le Tableau 9.2 présente la liste de ces contrôles.

Tableau 9.2 : Les principaux contrôles serveur Web

Contrôle	Description
`AdRotator`	Affiche une bannière
`Button`	Crée un bouton
`Calendar`	Affiche le calendrier du mois avec possibilité d'afficher le suivant et le précédent
`CheckBox`	Crée une case à cocher
`CheckBoxList`	Crée un groupe de cases à cocher avec sélection multiple
`DataGrid`	Affiche les éléments d'une source de donnée dans un tableau
`DataList`	Affiche les éléments d'une source de donnée avec un format particulier
`DropDownList`	Affiche une liste déroulante
`HyperLink`	Affiche un lien hypertexte
`Image`	Affiche une image
`ImageButton`	Affiche une image et vous permet de gérer les clics de l'utilisateur
`Label`	Affiche du texte

Tableau 9.2 : Les principaux contrôles serveur Web *(suite)*

Contrôle	Description
`LinkButton`	Crée un bouton lien hypertexte
`ListBox`	Affiche une zone de liste à choix multiple ou unique
`Panel`	Crée un conteneur pour d'autres contrôles
`PlaceHolder`	Réserve un emplacement pour des contrôles créés par le programme
`RadioButton`	Crée un bouton d'option
`RadioButtonList`	Affiche un groupe de boutons d'options
`Table`	Crée un tableau
`TableCell`	Crée une cellule de tableau
`TableRow`	Crée une ligne de tableau
`TextBox`	Affiche une zone de texte sur une ou plusieurs lignes
`Xml`	Affiche un document XML ou le résultat d'une transformation XML



Annexe

Reportez-vous à cette annexe à chaque fois que vous aurez besoin de vous rafraîchir la mémoire à propos de la syntaxe d'un opérateur ou d'une instruction de contrôle.

Tableau A.1 : Les mots clés de C#

abstract	base	bool	break	byte
case	catch	char	checked	class
const	continue	decimal	default	delegate
do	double	else	enum	event
explicit	extern	false	finally	fixed
float	for	foreach	goto	if
implicit	in	int	interface	internal
is	lock	long	namespace	new
null	object	operator	out	override
params	private	protected	public	readonly
ref	return	sbyte	sealed	short
sizeof	static	string	struct	switch

Tableau A.1 : Les mots clés de C# *(suite)*

`this`	`throw`	`true`	`try`	`typeof`
`uint`	`ulong`	`unchecked`	`unsafe`	`ushort`
`using`	`virtual`	`void`	`while`	

Les noms de variables

En C#, le nom d'une variable doit répondre aux critères suivants :

- Il peut contenir des lettres, des chiffres et le caractère _ (trait de soulignement).
- Le premier caractère doit être une lettre du jeu de caractères Unicode. Le caractère _ peut être choisi, mais son emploi est déconseillé car il est présent dans de nombreuses commandes spéciales.
- C# prend en compte la casse des caractères. Ainsi, `MaClasse` et `maClasse` font référence à deux variables distinctes.
- Vous ne pouvez pas choisir un mot clé de C# (voir section précédente).

Voici quelques exemples de noms illégaux ou simplement déconseillés :

- `Ma#classe` : le caractère # n'est pas autorisé.
- `_ma_classe` : autorisé mais déconseillé.
- `9var` : le premier caractère n'est pas une lettre.
- `String` : c'est un mot clé réservé.

Les opérateurs

Vous pouvez utiliser tous les opérateurs arithmétiques habituels : + (addition), – (soustraction), * (multiplication), / (division), % (modulo). Ce dernier opérateur permet d'obtenir le reste de la division du premier opérande par le second (7%3=1).

En associant ces derniers avec l'opérateur =, vous obtenez des opérateurs d'affectation composés :

Tableau A.2 : Les opérateurs d'affectation composés

Opérateur	Exemple	Description
+=	x += 1	x = x + 1
-=	x -= 1	x = x - 1
*=	x *= 1	x = x * 1
/=	x /= 1	x = x / 1
%=	x %= 1	x = x % 1

Préincrémentation ou postincrémentation

++ et -- sont des opérateurs unaires (ils s'appliquent à un seul opérande). Le premier ajoute une unité à l'opérande, alors que le second soustrait une unité.

Ils présentent cependant une particularité selon qu'ils se placent à gauche ou à droite de l'opérande :

```
var1 = 5
var2 = ++var1  //on incrémente var1 (6) puis on affecte
➥cette valeur à var2 (6)
var3 = --var1  //on décrémente var1 (5) puis on affecte
➥cette valeur à var3 (5)
var4 = var1++  //on affecte la valeur de var1 (5) à var4
➥puis on incrémente var1 (6)
var5 = var1--  // on affecte la valeur de var1 (6) à var5
➥puis on décrémente var1 (5)
```

La place de l'opérateur règle l'ordre des affectations.

Ces opérateurs sont surtout utilisés dans les boucles `for` (voir plus loin).

Les opérateurs relationnels

Ces opérateurs permettent de comparer des valeurs, ils renvoient donc `true` ou `false`.

`>`	**Supérieur à**
`<`	**Inférieur à**
`==`	**Egal à**
`!=`	**Différent de**
`>=`	**Supérieur ou égal à**
`<=`	**Inférieur ou égal à**

Les opérateurs logiques

`&&` (et) renvoie `true` si les deux opérandes sont vrais. Il permet de vérifier si plusieurs conditions sont remplies : `age > 15 && sexe == 'F'`.

`||` (ou) renvoie `true` si un des deux opérandes est vrai : `true || false` renvoie `true`.

`!` (non) renvoie la valeur booléenne inverse : `!true` renvoie `false`.

Concaténation de chaînes

L'opérateur `+` permet de concaténer des chaînes avec des opérandes de type `string`. Vous l'avez aperçu dans de nombreuses instructions :

```
Console.WriteLine("Bonjour" + nom + "et bienvenue");
```

L'opérateur d'indexation [ ]

C'est l'opérateur utilisé pour définir les tableaux et obtenir la valeur d'un élément de tableau :

```
int monTab[10]; //définit un tableau de 10 éléments
var = monTab[4]; //affecte à var la valeur du 4e élément du
➥tableau
```

L'opérateur de conversion

Quand vous travaillez sur des variables de types différents, vous ne pouvez pas affecter la valeur de l'une à l'autre, ni réaliser une opération sur deux variables de types différents, sauf si une conversion est réalisée. Cette conversion peut être implicite ou explicite.

La conversion de type implicite se fait selon une série de règles prédéfinies. Le compilateur suit ces règles pour savoir ce qu'il a à faire dans le cas où les opérandes d'une opération ne sont pas du même type. Une conversion implicite n'entraîne jamais de perte de données. On peut dire ainsi que le type de destination a une taille supérieure ou égale au type à convertir. Ainsi un **short** (2 octets) peut être converti implicitement en **int** (4 octets), mais pas en **byte** (1 octet).

Par contre, si vous voulez stocker une valeur de type **long** dans une variable de type **int**, vous devez réaliser une conversion explicite avec l'opérateur de conversion (*type cible*) et assumer ainsi la responsabilité de l'acte :

```
int varInt = 0;
long varLong = 1548;
varInt =(int) varLong;
```

Vous indiquez entre parenthèses le type dans lequel la variable concernée par la conversion (à droite de l'opérateur) doit être convertie.

L'opérateur conditionnel

Cet opérateur renvoie une valeur ou une autre suivant une condition. La syntaxe est la suivante :

```
Condition ? instruction_si_true : instruction_si_false ;
```

Si la condition est vraie, la première instruction est exécutée (**instruction_si_true**), sinon c'est la seconde (**instruction_si_false**).

L'opérateur sizeof

L'opérateur unaire `sizeof` indique la taille du type qu'on lui donne en opérande, il renvoie donc une taille mémoire. Toute action directement liée à la mémoire pose un problème avec la prise en charge du programme par le CLR. En effet, ce type d'opération diffère d'une plate-forme à l'autre. Cependant, les programmes C# sont extrêmement sûrs et ne peuvent théoriquement pas planter votre machine. Si le programmeur veut passer outre, il lui suffit d'indiquer au runtime que des instructions ne doivent pas être protégées par le CLR, en faisant précéder le nom de la méthode du mot clé `unsafe`. Par exemple :

Code A.1 : L'opérateur *sizeof()*

```
using System ;

class monApplication
{

  public static void Main()
  {
    AfficherTailleType();
  }

  public static unsafe void AfficherTailleType()
  {
    Console.Out.WriteLine("Sur cet ordinateur, un byte fait
                          ➥{0} octets", sizeof(byte));
    Console.Out.WriteLine("\t\tun short fait {0} octets",
                                          ➥sizeof(short));
    Console.Out.WriteLine("\t\tun int fait {0} octets",
                                           ➥sizeof(int));
    Console.Out.WriteLine("\t\tun long fait {0} octets",
                                          ➥sizeof(long));
    Console.Out.WriteLine("\t\tun float fait {0} octets",
                                         ➥sizeof(float));
    Console.Out.WriteLine("\t\tun double fait {0} octets",
                                        ➥sizeof(double));
    Console.Out.WriteLine("\t\tun decimal fait {0} octets",
                                       ➥sizeof(decimal));
  }
}
```

Compilez ce programme avec la commande `csc typeof.cs /unsafe`. Vous précisez au compilateur la présence d'instructions non protégées.

Vous obtenez quelque chose comme :

```
Sur cet ordinateur, un byte fait 1 octet
        un short fait 2 octets
        un int fait 4 octets
        un long fait 8 octets
        un float fait 4 octets
        un double fait 8 octets
        un decimal fait 16 octets
```

NOUVEAU

Contrairement aux versions C et C++ de cet opérateur, `sizeof` renvoie avec C# la taille du type qui lui a été transmis comme opérande. Vous ne pouvez pas lui transmettre un identificateur de données.

L'opérateur typeof

L'opérateur unaire `typeof` accepte n'importe quel type en opérande et renvoie un objet de type `Type` qui contient toutes les informations sur le type de l'opérande. La classe `Type` compte, entre autres, la méthode `GetType()` qui renvoie une chaîne de caractères comprenant le nom du type. Exemple :

Code A.2 : L'opérateur *typeof()*

```
using System ;

class monApplication
{
  public static void Main()
  {
     // on déclare deux objets de type Type
     Type t1, t2;

     t1 = typeof(Object);
     t2 = typeof(Console);

     Console.WriteLine(t1);
     Console.WriteLine(t2);
  }
}
```

Vous obtenez :

```
System.Object
System.Console
```

L'opérateur is

L'opérateur `is` permet au programmeur de savoir si un objet appartient à un type donné. En effet, un objet peut avoir plusieurs types, puisque la classe à partir de laquelle il a été instancié possède au moins une classe mère (sauf s'il s'agit de la classe `Object`). Il renvoie `true` lorsque son opérande de droite est du type de son opérande de gauche, et `false` dans le cas contraire. Exemple :

Code A.3 : L'opérateur *is*

```
using System;

class premClasse{ }

class secondeClasse{ }

public class monApplication
{

  // on vérifie à quelle classe appartient l'objet transmis
     ➥en paramètre
  public static void Test(object o)
   {

   Console.WriteLine( o is premClasse ?
            "Cet objet est de type premClasse" :
            "Cet objet n'est pas de type premClasse" );

    Console.Out.WriteLine( o is secondeClasse ?
                "Il est de type secondeClasse" :
                "Il n'est pas de type secondeClasse");
  }

  public static void Main()
  {
    premClasse objet1 = new premClasse();
    secondeClasse objet2 = new secondeClasse();

 Console.Out.WriteLine("Test d'objet1 :");
```

```
        Test (objet1);
            Console.Out.WriteLine("\nTest d'objet2 :");
            Test (objet2);
            Console.Out.WriteLine("\nTest de la chaîne de
            ⇨caractères :");
            Test ("une chaîne de caractères");
        }
    }
```

Vous obtenez :

```
Test d'objet1 :
Cet objet est de type premClasse
Il n'est pas de type secondeClasse

Test d'objet2 :
Cet objet n'est pas de type premClasse
Il est de type secondeClasse

Test de la chaîne de caractères :
Cet objet n'est pas de type premClasse
Il n'est pas de type secondeClasse
```

L'opérateur as

L'opérateur **as** permet une conversion entre types compatibles. Si l'opérande de gauche est un type compatible avec l'opérande de droite, alors l'opérateur renvoie l'opérande de gauche avec le type de l'opérande de droite. Sinon, `null` est renvoyé. La syntaxe est la suivante :

```
expression as type
```

Elle est équivalente à :

```
expression is type ? (type)expression : (type)null
```

Exemple :

Code A.4 : L'opérateur *as*

```
using System;
//Déclaration de la première classe
class premClasse : Object{ }

//Déclaration de la seconde, dérivée de la première
class secondeClasse : premClasse{ }
```

```
public class monApplication
{

   public static void Main()
   {
     premClasse objet1 ;
     secondeClasse objet2 = new secondeClasse();
     String sortie;

     //on tente une conversion de l'objet
     //de secondeClasse en premClasse
     objet1 = objet2 as premClasse;

     //si le handle ne pointe pas sur null,
     //c'est que objet2 est aussi de type premClasse

     sortie = ( objet1 == null ? "objet2 n'est pas de type
     ➥premClasse":"objet2 est aussi de type premClasse");
     Console.WriteLine( sortie );
   }
}
```

On obtient :

```
objet2 est aussi de type premClasse
```

Les instructions de contrôle

Les instructions de contrôle permettent d'exécuter certaines lignes de code de façon répétitive ou de choisir différents jeux d'instructions selon certains tests ou conditions.

If... else

Avec `if... else`, vous exécutez des instructions lorsqu'une certaine condition est respectée, et une autre série d'instructions si elle ne l'est pas :

```
if (variable==0)
{
   //Si la condition est vraie, on exécute les instructions
     ➥de ce bloc
   Console.WriteLine("La variable est nulle");
 //L'exécution se poursuit avec l'instruction qui suit le
   ➥bloc else
```

```
}
else
{
   //Si la condition est fausse, on exécute les
     ➥instructions de ce bloc-ci
   Console.WriteLine("La variable n'est pas nulle");
}
```

L'instruction `if` peut être codée sans le bloc `else`.

for

L'instruction `for` permet d'exécuter du code tant qu'une condition reste vraie :

```
for(instruction initiale ; condition ; instruction de
➥contrôle)
{
   //instructions
}
```

1. L'instruction initiale est exécutée une seule fois au début de l'exécution de l'instruction `for`.
2. Si la condition est évaluée comme étant vraie, les instructions sont exécutées, sinon l'exécution de `for` se termine.
3. L'instruction de contrôle est exécutée (en général, il s'agit d'un compteur).
4. Retour à l'étape 2.

switch

L'instruction `switch` permet d'exécuter du code selon la valeur d'une expression évaluée :

```
switch (expression évaluée)
{
  case valeur1:
     ligne de code;  //exécutée si l'expression a la valeur
                     ➥valeur1
     break;
  case valeur2:
     ligne de code;  //exécutée si l'expression a la valeur
                     ➥valeur2
     break;
```

```
    default:
        ligne de code;  //exécutée dans tous les autres cas
                        ➥de figure
}
```

INFO

En C++, vous pouviez exécuter plusieurs instructions de case d'affilée en omettant l'instruction break qui termine chacune d'elles. En C#, vous devez procéder différemment :

```
switch (nombre)
{
  case 1:   //première possibilité de regroupement
  case 3:
  case 5:
          System.Console.WriteLine("il est impair");
          break;
  case 2:   //deuxième possibilité de regroupement
          goto case 6;
          break;
  case 4:
          goto case 6;
          break;
  case 6:
          System.Console.WriteLine("il est pair");
          break;
  default:
          System.Console.WriteLine("il est supérieur à 6");
          break;
}
```

do

Avec une instruction **do**, le bloc de code est exécuté tant que le test booléen est vrai.

```
do
{
    lignes de code;
} while (test booléen);
```

Vous avez la possibilité de sortir de la boucle avec une instruction **break**.

while

C'est le même principe que **do**, sauf que le test booléen est évalué avant le bloc de code :

```
while (test booléen)
{
   lignes de code;
}
```

foreach

Comme son nom l'indique, **foreach** permet de faire une itération "pour chaque" élément d'un tableau ou d'une collection. Sa syntaxe est la suivante :

```
foreach(type de l'élément de tableau ou de collection)
{
   lignes de code;
}
```

Par exemple :

```
char[] monTableau = new char[ ] {'B','o','n','j','o','u','r'};
foreach(char i in monTableau)
{
    System.Console.Write("{0}", i);
}
```

La première ligne déclare un tableau contenant les lettres de "Bonjour", puis nous effectuons une itération sur les éléments de ce tableau. Devinez ce qui apparaît à l'écran ?

Les couleurs prédéfinies du .NET Framework

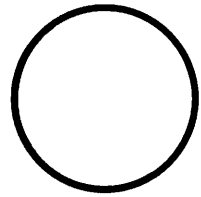
Color est une structure contenant toutes les valeurs de couleurs prédéfinies du .NET Framework. Vous pouvez les appliquer à tous les éléments de l'interface utilisateur de votre programme qui possèdent une propriété de couleur.

Tableau A.3 : Les couleurs prédéfinies du .NET Framework

AliceBlue	AntiqueWhite	Aqua	Aquamarine	Azure
Beige	Bisque	Black	BlanchedAlmond	Blue
BlueViolet	Brown	BurlyWood	CadetBlue	Chartreuse
Chocolate	Coral	CornflowerBlue	Cornsilk	Crimson
Cyan	DarkBlue	DarkCyan	DarkGoldenrod	DarkGray
DarkGreen	DarkKhaki	DarkMagenta	DarkOliveGreen	DarkOrange
DarkOrchid	DarkRed	DarkSalmon	DarkSeaGreen	DarkSlateBlue
DarkSlateGray	DarkTurquoise	DarkViolet	DeepPink	DeepSkyBlue
DimGray	DodgerBlue	Firebrick	FloralWhite	ForestGreen
Fuchsia	Gainsboro	GhostWhite	Gold	Goldenrod
Gray	Green	GreenYellow	Honeydew	HotPink
IndianRed	Indigo	Ivory	Khaki	Lavender
LavenderBlush	LawnGreen	LemonChiffon	LightBlue	LightCoral
LightCyan	LightGoldenrodYellow	LightGray	LightGreen	LightPink
LightSalmon	LightSeaGreen	LightSkyBlue	LightSlateGray	LightSteelBlue
LightYellow	Lime	LimeGreen	Linen	Magenta
Maroon	MediumAquamarine	MediumBlue	MediumOrchid	MediumPurple
MediumSeaGreen	MediumSlateBlue	MediumSpringGreen	MediumTurquoise	MediumVioletRed
MidnightBlue	MintCream	MistyRose	Moccasin	NavajoWhite
Navy	OldLace	Olive	OliveDrab	Orange
OrangeRed	Orchid	PaleGoldenrod	PaleGreen	PaleTurquoise

Tableau A.3 : Les couleurs prédéfinies du .NET Framework *(suite)*

PaleVioletRed	PapayaWhip	PeachPuff	Peru	Pink
Plum	PowderBlue	Purple	Red	RosyBrown
RoyalBlue	SaddleBrown	Salmon	SandyBrown	SeaGreen
SeaShell	Sienna	Silver	SkyBlue	SlateBlue
SlateGray	Snow	SpringGreen	SteelBlue	Tan
Teal	Thistle	Tomato	Transparent	Turquoise
Violet	Wheat	White	WhiteSmoke	Yellow
YellowGreen				



Index

Symboles

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Z

Achevé d'imprimer le 15 mars 2002
sur les presses de l'imprimerie «La Source d'Or»
63200 Marsat
Dépôt légal : 1cr trimestre 2002
Imprimeur n° 9301