Chapitre 3
Programmation orientée objet

1. Principes de la programmation orientée objet

La programmation orientée objet (POO) est un paradigme très répandu en développement logiciel. Il vient compléter un panorama déjà riche du paradigme procédural ainsi que du paradigme fonctionnel.

La POO est une forme de conception de code visant à représenter les données et les actions comme faisant partie de classes, elles-mêmes devenant des objets lors de leur création en mémoire. Cette notion a été rapidement présentée dans le chapitre précédent, il est maintenant temps de comprendre son fonctionnement plus en détail.

1.1 Qu'est-ce qu'une classe ?

Une classe est un élément du système que forme votre application. Une classe contient deux types d'élément de code : des données ainsi que des méthodes, représentant des actions. Il faut voir la classe comme étant une boîte dans laquelle il est possible de ranger ces deux types d'éléments. Pour faire un parallèle avec la vie réelle, nous pouvons facilement comprendre que la définition d'une classe s'applique à un objet comme un ordinateur, par exemple. Ce dernier dispose de méthodes (allumer, éteindre...) ainsi que des propriétés (nombre d'écrans, quantité de RAM...).

Conceptuellement, une classe n'est qu'une définition. Une fois que vous avez statué sur ce qu'elle doit contenir ainsi que ses méthodes, il convient de la créer. Cette action s'appelle l'instanciation. À la suite de cette opération, nous obtenons une instance en mémoire d'un objet.

Pour tenter une comparaison, prenons l'exemple d'une usine de fabrication d'objets en bois. Afin de pouvoir créer un objet, il faut un plan (la classe). Grâce à ce dernier, la machine peut découper et assembler les divers éléments (les données et méthodes) afin de créer une nouvelle instance (instanciation).

En C#, la déclaration d'une classe se fait grâce au mot-clé `class`. Il y a quelques spécificités possibles, notamment la portée, que nous étudierons juste après, dans la section Que peut-on déclarer dans une classe ? - Les méthodes, ainsi que les concepts de `static`, `sealed` et celui de `partial`. La syntaxe complète de la déclaration d'une classe est la suivante :

```
PORTÉE [static] [sealed] [partial] class NOM_CLASSE
```

Le nom de la classe est libre mais répond à deux règles :

– Il ne peut contenir que des caractères alphanumériques et le signe underscore (« _ »).

– Il ne peut pas commencer par un chiffre.

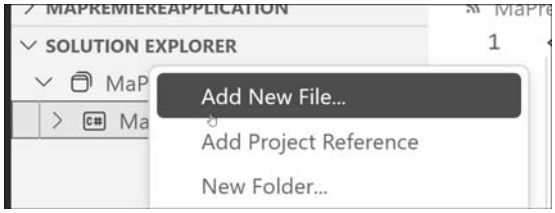
En plus de ces règles, les développeurs C# respectent souvent une convention syntaxique : l'utilisation du PascalCase. Cela indique que le nom commence par une majuscule et chaque mot est symbolisé par une majuscule également, par exemple, `OrdinateurPortable`. Le langage et le compilateur n'interdisent pas d'écrire `ordinateurPortable`, `Ordinateurportable` ou encore `ordinateurportable`, mais ces différentes déclarations ne respectent pas la convention largement admise et appliquée. Finalement, bien que ce soit possible, il est recommandé d'éviter tout caractère accentué dans le nom d'une classe. Par exemple, il est préférable d'appeler sa classe `Pieton` plutôt que `Piéton`, cela afin que le code C# produit soit le plus proche possible de ce que nous aurions en anglais.

Dans le programme de base créé en C# dans le précédent chapitre, une classe `Program` est créée par défaut. Nous pouvons constater qu'il n'y a ni notion de portée ni notion de `partial` ou `static`. Une fois qu'une classe est déclarée, elle définit un bloc, dans lequel nous pouvons implémenter les données et les méthodes dont notre programme a besoin pour fonctionner.

1.1.1 Les classes dans Visual Studio Code

Pour créer une classe dans Visual Studio Code, il est nécessaire de suivre les étapes suivantes :

- Dépliez la vue **Solution Explorer** du projet.
- Placez-vous sur le dossier où vous souhaitez créer la nouvelle classe (ou directement sur le nom du projet si vous souhaitez la créer à la racine).
- Faites un clic droit pour sélectionner l'élément de menu **Add New File**.
- Renseignez le nom de la classe dans la petite pop-up qui s'est ouverte en haut au centre de l'écran (toujours sans espaces ni caractères spéciaux).



Ajout d'une nouvelle classe avec Visual Studio Code

À la suite de ces manipulations, un nouveau fichier, portant le nom de la classe suivi de l'extension .cs, est disponible dans la hiérarchie à gauche. Par défaut, ce fichier sera ouvert et contient la classe qui a été déclarée dans l'espace de noms correspondant au dossier de destination.

1.1.2 L'héritage

Il existe un concept extrêmement important en POO : l'héritage. Globalement, si vous avez la possibilité de dire « X est un Y », l'équivalent pourrait être de dire « X hérite de Y ». X reprend toutes les propriétés et tous les comportements de Y, mais le spécifie. Donnons un exemple concret : « Un Mac est un ordinateur ». Donc, au niveau du développement orienté objet, un Mac reprend toutes les propriétés d'un ordinateur ainsi que ses comportements, mais les spécifie en y apportant ses propres éléments. On dit dans ces cas-là que `Mac` est une classe fille de la classe `Ordinateur`.

En C#, cette notion est centrale car tous les éléments que vous allez manipuler héritent naturellement de la classe `System.Object`, qui définit le comportement de base de n'importe quel objet. De surcroît, contrairement à d'autres langages (comme le C++), il n'est pas possible, en C#, d'hériter de plusieurs classes : une seule classe mère est possible. Si aucune classe mère n'est spécifiée, c'est par définition la classe `System.Object` qui constitue la classe mère (sans qu'une quelconque manipulation soit requise).

Remarque

En C#, il n'est pas possible d'hériter de plusieurs classes. Il faut donc choisir la classe dont on hérite. En l'absence de précision, le compilateur génère automatiquement, de façon transparente, un héritage de la classe `System.Object`, comme décrit ci-dessus. Si nous spécifions un héritage, cela ne veut pas dire que la classe hérite de `System.Object` ET de la classe héritée, mais uniquement de la classe héritée, qui remplace l'héritage généré par le compilateur. La classe héritée, elle-même, hérite soit d'une autre classe, soit directement de `System.Object`. En finalité, toutes les classes en C# héritent d'une façon ou d'une autre de `System.Object`.

Afin d'indiquer qu'une classe hérite d'une autre, il faut utiliser le deux-points, suivi de la classe dont on souhaite hériter :

```
class Ordinateur { }
class Mac : Ordinateur { }
```

Bien entendu, ce n'est pas parce qu'une classe hérite d'une autre qu'elle a forcément accès à tout ce qui a été défini au sein de la classe mère.

1.1.3 L'encapsulation

Tout ce qui se trouve à l'intérieur d'une classe est désigné par un terme bien spécifique : l'encapsulation. Avec celle-ci vient également la notion de portée, qui indique comment les choses sont perçues d'un point de vue extérieur à la classe.

La portée permet de définir la visibilité d'un élément d'une classe ou de la classe elle-même. Il existe en tout sept portées en C# :

- `public` : définit que l'élément est totalement visible dans et en dehors de la classe.
- `private` : définit que l'élément n'est visible qu'au sein de la classe où il est déclaré alors qu'il est totalement invisible de l'extérieur.
- `internal` : définit que l'élément est visible uniquement au sein du projet où il est déclaré. Nous pouvons considérer l'élément comme étant public, mais simplement au sein du projet dans lequel il est déclaré. Un autre projet qui référence notre projet n'a pas connaissance d'un élément déclaré comme `internal`. Par défaut, en l'absence de portée explicite sur une classe, c'est la portée `internal` qui est sélectionnée par le compilateur.
- `protected` : définit que l'élément est visible uniquement au sein de la classe où il est déclaré ainsi que dans sa hiérarchie de classes filles. Cela rejoint le concept de l'héritage, que nous verrons plus loin dans ce chapitre.
- `protected internal` : définit un cumul entre `protected` et `internal`. Un élément déclaré avec cette portée est visible par la classe concernée ainsi que ses classes filles, tout comme par toutes les autres classes au sein du même projet. Cela signifie également que si une classe fille est déclarée en dehors du projet actuel, elle peut accéder à un élément `protected internal`, tout comme n'importe quelle classe du même projet.
- `private protected` : définit une intersection entre `protected` et `internal`. Un élément déclaré avec cette portée n'est visible que par la classe concernée ainsi que les classes filles qui sont définies au sein du même projet. Cela veut dire qu'une classe fille définie en dehors du projet actuel ne pourra pas accéder à cet élément.

– `file` : ajoutée en C# 11, cette portée définit une visibilité uniquement dans le cadre du fichier en cours. Cette portée est très particulière car elle n'a pas vocation à être directement utilisée par les développeurs. Elle existe surtout pour les outils de génération automatique de code. Néanmoins, dans de très rares cas, il peut être utile de déclarer un élément qui n'existe que dans le cadre d'un fichier pour un algorithme précis. Il est à noter également que, contrairement aux autres portées, cette portée n'est valide que pour la déclaration d'un type. Elle ne peut pas s'appliquer sur une méthode, un champ ou une propriété.

Avec toutes ces portées, il est possible de créer la classe qui correspond finement au besoin de votre application, pour éviter que certains éléments ne sortent du périmètre de la classe. En reprenant notre exemple, considérons que la classe `Ordinateur` dispose d'un booléen indiquant si la machine est allumée ou non. Afin d'éviter que quelqu'un ne puisse manipuler directement cette donnée, la manière de procéder est de la définir comme étant publiquement accessible en lecture, mais privée pour ce qui est de l'écriture. En conséquence, seule une méthode publique, définie dans cette classe, comme par exemple `Allumer` ou `Eteindre`, peut changer la valeur de cet indicateur. Nous nous préservons ainsi d'un changement d'état non maîtrisé (car nous pouvons considérer que l'opération d'extinction nécessite d'effectuer quelques opérations en amont avant de basculer le booléen).

1.2 Que peut-on déclarer dans une classe ?

Nous l'avons vu, il existe deux types d'éléments que nous pouvons déclarer dans une classe : des méthodes (actions) et des données. Voyons rapidement comment les déclarer.

1.2.1 Les méthodes

Une méthode traduit une action qu'il est possible d'invoquer sur la classe. Lors de la déclaration d'une méthode, il faut se poser les questions suivantes :

– S'agit-il d'une action qui doit pouvoir être réalisée depuis l'extérieur ou uniquement depuis l'intérieur de la classe ?

– Est-ce qu'une valeur de retour particulière est attendue ?

– Certaines informations sont-elles nécessaires pour que cette méthode fonctionne ?

Vous avez déjà eu un aperçu d'un appel de méthode dans le premier chapitre, sur la classe `Console` : `WriteLine` et `ReadLine`. Ces deux méthodes illustrent bien les points cités précédemment :

– `WriteLine` doit pouvoir être appelée depuis l'extérieur. Nous n'attendons pas de valeur en retour à son appel, mais il est nécessaire de lui transmettre l'information que nous souhaitons écrire.
– `ReadLine` doit également pouvoir être appelée depuis l'extérieur. Nous avons besoin de récupérer l'information saisie par l'utilisateur uniquement, sans besoin de lui transmettre une quelconque information.

La syntaxe de déclaration d'une méthode dans une classe est la suivante :

```
PORTÉE [static] TYPE_RETOUR NOM_METHODE([PARAMÈTRES])
```

Le type de retour doit correspondre à un type C# connu. Par exemple, si nous souhaitons créer une méthode qui réalise l'addition de deux nombres et renvoie le résultat, le tout accessible publiquement, nous la déclarons comme suit :

```
public int Addition(int premier, int second) {}
```

Remarque

Dès lors que nous déclarons une méthode avec une valeur de retour sans écrire le contenu de la méthode, le compilateur émet immédiatement une erreur de compilation. Ceci est dû au fait que chaque méthode retournant un résultat doit obligatoirement comporter une instruction `return`.

Lorsqu'une méthode doit renvoyer une valeur, il faut utiliser le mot-clé `return` afin de définir la valeur que nous souhaitons renvoyer. L'instruction `return` peut être utilisée directement avec une valeur ou alors nous pouvons nous servir d'une variable du type de retour attendu. Dans le cas de l'exemple ci-dessus, ces deux façons d'écrire la méthode sont valides :

```
public int Addition(int premier, int second)
{
    return premier + second;
}
```

```
public int Addition(int premier, int second)
{
    int resultat = premier + second;
    return resultat;
}
```

Un élément important à garder en mémoire : à l'instar de ce que nous avons vu dans le chapitre précédent avec la déclaration de classes du même nom au sein du même espace de noms, il n'est pas possible de déclarer deux fois la même méthode à l'intérieur d'une même classe. Si les noms sont identiques et que les paramètres le sont également, alors le compilateur C# considère qu'il s'agit de la même méthode. La valeur de retour ne constitue pas un élément distinctif. Ainsi, la déclaration des deux méthodes suivantes dans la même classe est impossible et cela provoque une erreur de compilation :

```
public int Addition (int premier, int second)
{
    return premier + second;
}
public void Addition (int premier, int second)
{
}
```

Remarque

Comme nous pouvons le constater dans l'exemple ci-dessus, le mot-clé void précise que la méthode ne renvoie aucun résultat. La notion de type de retour étant obligatoire, il faut utiliser ce mot-clé pour indiquer les cas où il n'y en a pas.

Si la méthode ne prend pas de paramètres, la présence de parenthèses ouvrantes et fermantes accolées au nom de la méthode est malgré tout nécessaire pour signifier qu'il s'agit d'une méthode :

```
public void MaMethode()
{
}
```

Au sein d'une méthode qui déclare son propre bloc, il est possible de déclarer des variables et constantes qui sont considérées uniquement comme locales (c'est-à-dire visibles au sein de la méthode et de tous ses sous-blocs, mais invisibles dans les blocs parents, directs ou indirects).

Chapitre 4
Les types du C#

1. "En C#, tout est typé !"

Le terme générique "**type**" regroupe les classes, les structures, les records, les interfaces, les énumérations et les délégués. Ces types sont décrits dans la CTS (*Common Type System*) pour que des compilateurs de langages différents puissent générer un code exploitable par la CLR (*Common Language Runtime*). Un programme utilise les différents types et un assemblage peut implémenter plusieurs types.

Voici les définitions succinctes des différents types proposés par le C# :

– Le type "Classe" est l'implémentation C# de ce qui a été présenté dans les premiers chapitres. La classe est évidemment le type le plus utilisé dans les applications. Le chapitre Création de classes en définit précisément la syntaxe de déclaration, d'allocation et d'utilisation.

– Le type "Structure" est assez voisin de celui du langage C. Avant la démocratisation de la programmation objet, les structures étaient le moyen le plus commun offert aux développeurs pour composer leurs propres types. Retenons pour l'instant que les structures du C# sont très proches des classes et que, quand elles sont utilisées à bon escient, elles peuvent améliorer les performances d'une application. Nous verrons au chapitre suivant que le .NET encapsule la plupart de ses types "simples" (les entiers, les caractères, etc.) dans des structures. Sachez que les structures n'existent pas en Java.

– Le type "Record" est un objet pouvant être classe ou structure qui s'identifie par son contenu et non par son emplacement en mémoire. Cette distinction subtile sera détaillée plus loin.

– Le type "Interface" est largement utilisé dans le .NET et contribue à la communication entre les classes. Retenons pour l'instant qu'une interface est une classe souvent sans code qui formalise un lot de méthodes obligatoires pour la classe qui l'implémentera. Le chapitre Héritage et polymorphisme traite du sujet.

– Le type "Enumération" permet la définition de listes clés-valeurs et la création des données dont les contenus seront limités à ces clés. Rappelons l'exemple d'un type *Jour* pouvant contenir de *lundi* à *dimanche*. Si, lors de la rédaction du programme, on tente de copier dans un objet de ce type la clé *Mars*, il y aura une erreur de compilation. En C#, ce type apporte un lot de méthodes permettant de gérer cette liste par programmation.

– Le type "Delegate" (Délégué) encapsule la notion de pointeur de fonction du C/C++, origine de bien des soucis, en commençant par lui attribuer un typage fort. En effet, le pointeur de fonction "conventionnel" n'est autre qu'une adresse mémoire sans autre précision sur la signature et l'application s'arrête en erreur quand les paramètres passés ne correspondent pas aux paramètres attendus... C'est pourquoi le type *delegate* du C# va être défini précisément avec la signature de la méthode qui lui sera associée. Ensuite, l'instance de type *delegate*, généralement créée au sein d'une classe amenée à communiquer avec d'autres, gère une liste "d'abonnés" via une syntaxe déconcertante de simplicité. Il suffit en effet d'utiliser l'opérateur += du *delegate* pour s'abonner à la liste de diffusion et -= pour s'en désenregistrer. Les *delegate* sont très largement utilisés dans le C# ; on les retrouvera beaucoup dans les interfaces graphiques pour que les composants puissent notifier l'application de leurs changements d'états.

Durant ce chapitre seront abordées des notions illustrées par des extraits de code. Ces extraits de code utilisent des syntaxes décrites dans les chapitres suivants mais la compréhension des chapitres suivants passe... par celle de ce présent passage ! Vous avez donc à prendre pour argent comptant dans un premier temps les syntaxes des exemples mais nous les approfondirons par la suite.

Remarque

Tous les exemples de ce chapitre sont des projets de la solution Visual Studio TypesDuCSharp.sln figurant dans le répertoire Chap4 du .zip accompagnant cet ouvrage. Ce fichier d'accompagnement est à télécharger sur le site des Éditions ENI www.editions-eni.fr.

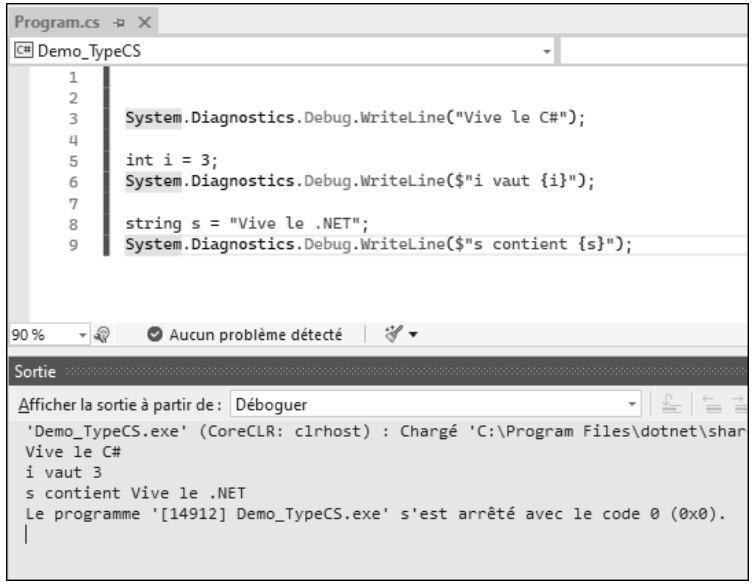
Introduction à System.Diagnostics.Debug

Ces exemples utilisent des classes de tests et également une classe système appelée *System.Diagnostics.Debug*. Cette classe permet, entre autres, d'écrire des messages dans la fenêtre **Sortie** de Visual Studio et également de vérifier des conditions passées en paramètres. Elle sera étudiée avec d'autres classes de même type dans le chapitre Traçage et instrumentation des applications.

Syntaxe d'affichage dans la fenêtre **Sortie** de Visual Studio

```
System.Diagnostics.Debug.WriteLine("le message");
```

Exemple d'utilisations simples et composées



Le signe $ qui précède le contenu de la chaîne permet d'effectuer une "interpolation", à savoir un remplacement de séquence {blabla} par le contenu de la variable blabla. Cette extension très souple et très pratique est arrivée avec le C# 6.

La méthode *System.Diagnostics.Debug.Assert* permet de vérifier qu'une condition est vraie pendant l'exécution de votre code. En utilisant cette méthode vous n'intervenez pas sur le déroulement du programme en tant que tel ; vous vérifiez juste que ce qui est prévu à tel endroit du code est correct. Si la condition est fausse, une boîte de dialogue sera affichée pour vous en informer.

Syntaxe d'utilisation de la méthode System.Diagnostics.Debug.Assert

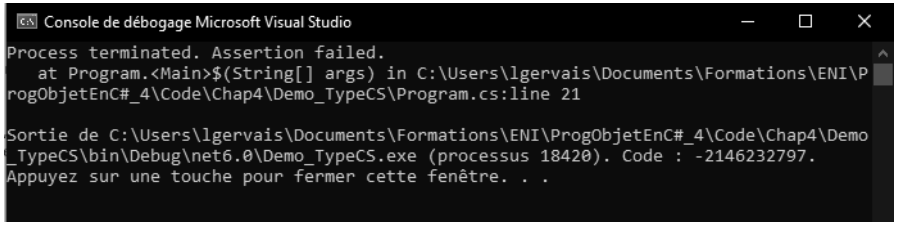
```
System.Diagnostics.Debug.Assert(<condition>);
```

Exemples d'utilisation de la méthode Assert

```
// Vérification de la condition "1 est différent de 2"
System.Diagnostics.Debug.Assert(1 != 2);
// Comme la condition est vraie, le programme
// passe à la ligne suivante


// Pour voir l'effet produit
// lorsqu'une condition n'est pas vérifiée,
// une erreur est "forcée" en ligne suivante
System.Diagnostics.Debug.Assert(1 == 2);
```

À l'exécution de la seconde ligne de l'extrait, le programme affiche une boîte de message et attend que l'utilisateur la referme avant de poursuivre l'exécution du code.

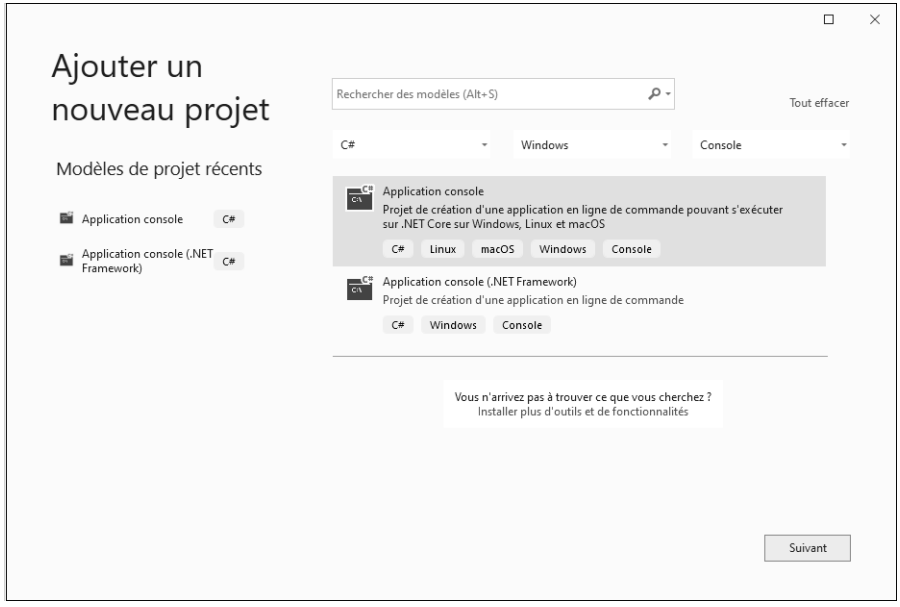


Ainsi, vous pouvez vérifier que ce que vous avez prévu se réalise correctement. On utilise *System.Diagnostics.Debug.Assert* principalement pendant les phases de mise au point pour éventuellement ajouter du code de protection par la suite. Nous verrons d'ailleurs que Visual Studio génère une version "mise au point" (*Debug*) et une version "production" (*Release*). *System.Diagnostics.Debug.Assert* n'a aucun effet sur un code compilé en mode "production".

Introduction à System.Console

Nous l'avons déjà utilisée au chapitre Introduction à .NET 6 et à VS pour afficher le classique *Hello World* à l'écran ; la console va servir de support à plusieurs exemples à suivre. Cet environnement d'exécution très sommaire présente l'avantage de pouvoir simplement afficher des chaînes à l'écran et lire des entrées clavier.

Comme nous l'avons vu, le type **Application console** se choisit à la création du projet.



Voici les principales commandes qui seront utilisées :

Affichage d'une chaîne suivie d'un changement de ligne

```
Console.WriteLine("Message à afficher...");
```

Affichage d'un type primitif sans changement de ligne

```
      int j = 358;
      Console.Write(j);
```

Affichage d'une composition

```
int k = 2;
int l = 3;
Console.WriteLine($"k contient {k} et l contient {l}");
```

Lecture d'une chaîne de caractères saisie au clavier
et terminée par la touche [Entrée]

```
string saisie = Console.ReadLine();
```

Ces présentations étant faites, nous pouvons passer à la suite...

2. "Tout le monde hérite de System.Object"

Le type *System.Object* est la base directe ou indirecte de tous les types du .NET, ceux existants et ceux que vous allez créer (la notion d'héritage a déjà été un peu abordée dans les premiers chapitres). L'héritage d'*Object* étant implicite, sa déclaration est inutile. Tous les types héritent de ses méthodes et peuvent même en substituer certaines.

C'est ce que fait *System.ValueType* qui, dans la hiérarchie des types du .NET, devient la base de la famille "Valeurs" en adaptant les méthodes de *System.Object*.

2.1 Les types Valeurs

La famille "Valeurs" se divise en plusieurs parties :

– les énumérations
– les structures
– les records (s'ils sont instanciés en type Valeur)

Les structures sont elles-mêmes sous-divisées en :

– types numériques :
  – les types intégraux :

Type	Taille
sbyte	Entier signé sur 8 bits
byte	Entier non signé sur 8 bits
char	Caractère UNICODE 16 bits
short	Entier signé sur 16 bits
ushort	Entier non signé sur 16 bits
int	Entier signé sur 32 bits
uint	Entier non signé sur 32 bits
long	Entier signé sur 64 bits
ulong	Entier non signé sur 64 bits

  – les types à virgule flottante :

Type	Précision
float	7 chiffres
double	15-16 chiffres

  – le type décimal (adapté aux calculs financiers) :

Type	Précision
decimal	28-29 chiffres