Un livre de Wikilivres.

Programmation C sharp

Introduction

C sharp

Le langage de programmation C# (*C dièse* en français, ou prononcé *C-sharp* en anglais) a été développé par la société Microsoft, et notamment un de ses employés, Anders Hejlsberg, pour la plateforme *.NET* (*point NET* / *dot NET*).

Ce langage est orienté objet, avec un typage fort. Il est très proche du langage Java.

Il est précompilé en MSIL (*Microsoft Intermediate Language*), puis exécuté sur une machine virtuelle, ou compilé en code natif à l'exécution. Il dispose d'un ramasse-miettes (*garbage collector*). Il utilise l'API .NET en remplacement des MFC (*Microsoft foundation class*). Il semble être le nouveau langage pour développer des applications Windows, avec Visual Basic et C++.

Caractéristiques partagées

Le langage C# possède un certain nombre de caractéristiques communes avec d'autres langages de programmation. Il sera donc plus facile de l'apprendre si ces autres langages sont connus.

Caractéristiques partagées avec le langage Java

- Syntaxe : les mots clés communs avec Java s'utilisent dans les mêmes circontances et de la même manière : `public`, `private`, `protected`, `abstract`, `class`, `interface`, `try`, `catch`, `finally`, `throw`, `new`,

`return`, `this`, `if`, `while`, `do`, `for`, `foreach`, `enum`... et le mot clé `lock` est l'équivalent C# du mot clé Java `synchronized`;
- *Garbage collector* : les objets qui ne sont plus référencés sont traités par le ramasse-miettes afin de libérer la mémoire qu'ils occupent ;
- Références : les objets sont en fait des références ;
- Documentation automatique : cette caractéristique commune utilise cependant une syntaxe différente dans les 2 langages : le langage Java utilise les commentaires spéciaux `/** */` au format HTML, avec des tags commençant par le caractère arobase `@` ; tandis que le langage C# utilise les commentaires /// au format XML ;
- Méthodes courantes : une majeure partie des méthodes de l'API de C# ressemblent à celles de Java, excepté que leur nom commence par une majuscule : Main, Equals, ToString, Length, IndexOf, ...

Caractéristiques partagées avec le langage C++

- surcharge des opérateurs ;
- structures (mot clé `struct`) ;
- énumérations (mot clé `enum`) ;
- pointeurs : il est possible, en mode *unsafe*, d'utiliser des pointeurs au lieu de références.

Caractéristiques partagées avec d'autres langages

- propriétés (Delphi) : une propriété est un couple de méthodes (*get* et *set*) appelées lorsque celle-ci est lue ou modifiée ;
- attributs : un attribut est lié à une déclaration et contient des méta-données sur celle-ci (méthode obsolète, importée d'une DLL, ...) ;
- *delegate* : un *delegate* est un modèle de méthode, qui lorsqu'il est appelé, appelle toutes les méthodes qui lui ont été associées. Pour faire un parallèle avec le C++, les *delegates* peuvent être comparés à des pointeurs de fonction. Leur sémantique est toutefois nettement plus riche qu'en C++.

Compilation

Les fichiers sources

Un fichier source C# porte l'extension "`.cs`". Il s'agit d'un fichier texte.

Le programme compilé porte l'extension "`.exe`".

Une bibliothèque rassemble des classes et porte l'extension "`.dll`". Elle peut être utilisée par un programme ou une autre bibliothèque.

La compilation

Un compilateur C# permet la traduction du programme source en instructions .Net.

Contrairement à d'autres compilateurs, ces instructions sont produites pour un processeur virtuel et ne sont donc pas directement interprétées par le processeur, mais interprétés par le moteur .Net.

Sous Windows, le compilateur produit un exécutable appelant l'interpréteur .Net.

Sous Linux, le programme produit n'est pas directement exécutable et doit être lancé en argument de l'interpréteur `mono`.

Compilateurs

Plusieurs compilateurs C# existent selon la plateforme utilisée.

Il existe plusieurs versions des spécifications du framework : 1.0, 1.1, 2.0. Ces versions successives ajoutent de nouvelles fonctionnalités au langage.

Il existe également différentes versions de la spécification du langage C# :

- C#1.0 : version initiale du langage,
- C#2.0 : ajoute de nouvelles classes à l'API (compression de données, collections génériques, ...), permet l'utilisation de types génériques, facilite la création d'énumération avec le mot clé `yield`,
- C#3.0 : ajoute de nouvelles facilités de syntaxe : types non spécifiés pour les variables locales (déduit d'après la valeur d'initialisation), intégration des requêtes SQL dans le langage (LINQ [1]), ajout de nouvelles méthodes à une classe existante, expressions lambda pour les *delegates* anonymes, initialisation des membres d'un objet à la déclaration.
- C#4.0 : ajoute de nouvelles fonctionnalités : covariance et contrevariance, binding dynamique, paramètres nommés et optionnels, meilleure interopérabilité avec COM.

Compilateur pour Windows

Le framework .NET est disponible gratuitement pour les utilisateurs de Windows.

Si le framework Microsoft .NET est installé, le compilateur nommé `csc.exe` doit se situer dans l'un des deux répertoires suivants (à ajouter à la variable d'environnement PATH) :

```
C:\WINDOWS\Microsoft.NET\Framework\v numéro_de_version_du_framework
C:\WINDOWS\Microsoft.NET\Framework64\v numéro_de_version_du_framework
```

Le répertoire Framework64 contient la version 64 bits du Framework, disponible seulement sur les versions 64 bits de Windows. Afin de profiter de toutes la puissance du matériel, il est préférable d'utiliser la version 64 bits si Windows est également 64 bits.

Il est également possible d'utiliser Visual Studio .NET pour développer, compiler et débogguer les applications C#. L'édition Express de Visual Studio est téléchargeable gratuitement sur le site de Microsoft [2], mais possède moins d'outils que la version complète.

Une autre possibilité est d'utiliser SharpDevelop qui a l'avantage d'être un logiciel libre [3].

Compilateur pour Linux

Mono est une implémentation libre de la plate-forme de développement Microsoft .NET. Le compilateur est nommé `msc`. L'interpréteur est nommé `mono`[4].

`gmcs` est le nouveau compilateur C# 2.0. Il est recommandé d'utiliser ce dernier.

Références

1. http://msdn.microsoft.com/data/ref/linq/
2. http://msdn.microsoft.com/vstudio/express/visualcsharp
3. http://www.icsharpcode.net/OpenSource/SD/
4. http://www.mono-project.com

Compilation et exécution

La compilation et l'exécution peuvent se faire à l'aide d'un environnement graphique. Il est toutefois utile de connaître la ligne de commande à utiliser en cas d'absence d'environnement graphique ou si une option spéciale de compilation doit être utilisée, ...

Pour utiliser les lignes de commandes de compilation :

1. Ouvrir une fenêtre console,
2. Faire du répertoire du programme à compiler ou exécuter le répertoire courant,
3. Vérifier que la variable d'environnement PATH contient le chemin vers le compilateur, en tapant le nom du compilateur (`csc` sous Windows, `gmcs` sous Linux (Mono)).

Si le compilateur ne se lance pas (programme non trouvé) :

- Vérifier qu'un compilateur C# est installé,
- Vérifier que le chemin complet du répertoire du compilateur est contenu dans la variable d'environnement PATH,
- Essayer de lancer le compilateur en donnant son chemin complet.

Le compilateur peut produire soit directement un exécutable, soit un module (une bibliothèque DLL) utilisable par d'autres programmes.

Si les sources d'un exécutable ou d'un module sont répartis entre différents fichiers (même répertoire ou non), il faut spécifier leur chemin au compilateur.

Exemple : Les sources de l'exécutable prog.cs sont répartis dans deux fichiers : `C:\Prog\Exemple\main.cs` et `D:\Prog\Util\util.cs`. Il faut donc utiliser la ligne de commande suivante :

Windows	`csc C:\Prog\Exemple\main.cs D:\Prog\Util\util.cs`
Linux (Mono)	`gmcs C:\Prog\Exemple\main.cs D:\Prog\Util\util.cs`

Il est également possible d'utiliser les caractères génériques * et ? pour spécifier plusieurs fichiers.

Les syntaxes des commandes à utiliser pour les différentes tâches du développement sont résumées dans le tableau suivant :

Tâche	Windows	Linux
compiler un programme (.exe) avec fenêtre console	`csc` *`fichier`*`.cs`	`gmcs` *`fichier`*`.cs`
compiler un programme (.exe) sans fenêtre console	`csc /t:winexe` *`fichier`*`.cs`	`gmcs -target:winexe` *`fichier`*`.cs`
compiler une bibliothèque (.dll)	`csc /t:library` *`fichier`*`.cs`	`gmcs -target:library` *`fichier`*`.cs`
utiliser une bibliothèque (.dll)	`csc /r:`*`fichier`*`.dll ...`	`gmcs -r:`*`fichier`*`.dll ...`
exécuter un programme	*`programme arguments...`*	`mono` *`programme`*`.exe` *`arguments...`*

Un premier programme

Hello world

Cet exemple typique de programmation est censé montrer la syntaxe du langage.

Le fichier source

Enregistrez le listing suivant dans un document texte, intitulé par exemple « helloworld.cs » :

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

Test

Compilez-le, puis lancez l'exécutable produit. Le programme doit afficher :

```
Hello world !
```

Détails

Voici le détail ligne par ligne du fichier source, certains concepts étant expliqués dans les chapitres suivants :

```
1 using System;
```

Le fichier source utilise l'espace de noms nommé « *System* ».

```
public class HelloWorld
```

```
2   3 {
```

Déclaration d'une classe nommée « `HelloWorld` ».

```
4     public static void Main()
5     {
```

Déclaration d'une méthode statique nommée « `Main` » dans la classe `HelloWorld`. Cette méthode est celle qui est appelée au démarrage du programme.

```
6         Console.WriteLine("Hello world !");
```

Affichage de la ligne « `Hello world !` » sur la console. `Console` désignant la console, appartient à l'espace de nom `System`.

```
7         Console.ReadLine();
```

Attendre que la touche entrée soit frappée avant de poursuivre. Cette ligne de code n'est pas nécessaire si vous lancez le programme depuis une console déjà ouverte. Dans le cas contraire (double-clic sous Windows), cette ligne de code permet de maintenir la fenêtre de console ouverte, car celle-ci se ferme une fois le programme terminé (option par défaut).

```
8     }
```

Fin de la méthode Main.

```
9  }
```

Fin de la classe HelloWorld.

Hello world en bibliothèque

Le programme de cet exemple a le même but que le précédent : afficher le message « Hello world ! » sur la console. Cette version utilisera une seconde classe compilée en bibliothèque.

Les fichiers sources

Enregistrez ce premier listing dans un document texte, intitulé par exemple « helloworld.cs » :

```
using System;
public class HelloWorld
{
    public static void Afficher()
```

```
    {
        Console.WriteLine("Hello world !");
        Console.ReadLine();
    }
}
```

Enregistrez le second listing dans un document texte, intitulé par exemple « premierprog.cs » :

```
public class PremierProg
{
    public static void Main()
    {
        // appel à la méthode statique Afficher() de la classe HelloWorld
        HelloWorld.Afficher();
    }
}
```

Test

Compilez la bibliothèque :

Windows	`csc /t:library helloworld.cs`
Linux (Mono)	`gmcs -target:library helloworld.cs`

Le compilateur a créé un fichier nommé `helloworld.dll`. Cette bibliothèque doit être utilisée lors de la compilation du programme principal.

Compilez le programme :

Windows	`csc /r:helloworld.dll premierprog.cs`
Linux (Mono)	`gmcs -r:helloworld.dll premierprog.cs`

Le compilateur a produit un fichier nommé `premierprog.exe`.

Puis lancez l'exécutable produit :

Windows	`premierprog.exe`
Linux (Mono)	`mono premierprog.exe`

Le programme doit afficher :

```
Hello world !
```

Les commentaires

Tout bon programme a des fichiers sources bien commentés pour expliquer comment cela fonctionne et pourquoi certains choix ont été faits. Ce qui évite une perte de temps lorsque le code source est repris et modifié, soit par un autre développeur, soit par l'auteur qui ne se souvient pas forcément de son projet s'il n'y a pas touché depuis longtemps.

Bloc de commentaire

Un bloc de commentaire est délimité par les signes slash-étoile `/*` et étoile-slash `*/` comme en Java et en C++. Exemple :

```
/*
    Un commentaire explicatif
    sur plusieurs lignes...
*/
```

Les blocs ne peuvent être imbriqués car dès que le compilateur trouve slash-étoile `/*`, il recherche la première occurrence d'étoile-slash `*/` terminant le commentaire.

⚠ Ce code contient **une erreur volontaire** !

```
/* : début du commentaire

    /* : ignoré

    fin du commentaire : */

erreur ici car le commentaire est fini : */
```

Commentaire de fin de ligne

Un commentaire de fin de ligne débute par un double slash `//` et se termine au prochain retour à la ligne. Exemple :

```
x++;  // augmenter x de 1
```

Astuce : Visual Studio utilise ce type de commentaire pour les commandes commenter/décommenter le groupe de lignes sélectionnées (raccourcis claviers : [Ctrl-E] [C] pour commenter, [Ctrl-E] [U] pour décommenter, ou [Ctrl-K] [C] et [Ctrl-K] [U] avec les versions de Visual Studio .NET antérieures à 2005).

Commentaire de documentation

Le langage C# utilise une forme spéciale des commentaires pour documenter les classes. Ces commentaires commencent par un triple slash `///` et se terminent au prochain retour à la ligne. Alternativement, ils peuvent être encadrés par `/**` et `*/`.

Le contenu de ces commentaires est au format XML. Il est possible d'utiliser plusieurs lignes. Cette documentation se rapporte toujours à un des éléments de la déclaration qui suit.

Exemple :

```
/// <summary>
/// Une classe pour démontrer
/// les commentaires de documentation
/// </summary>
public class Exemple
{
    ...
}
```

Certaines balises XML sont standards, mais il est possible d'utiliser des balises supplémentaires. Pour plus de détails, voir le chapitre nommé Documentation XML des classes.

Les espaces de noms

Définition

Un espace de nom regroupe plusieurs déclarations sous un même nom ajouté à ceux des éléments déclarés (classes, interfaces, espace de nom imbriqué).

Exemple :

```
namespace MesClasses
{
    public class ClasseUne
    { ... }

    public class ClasseDeux
    { ... }
}
```

Utiliser les membres d'un espace de nom

Les classes définies à l'intérieur de cet espace de noms peuvent se faire référence par leur nom simple : `ClasseUne`, `ClasseDeux`.

Une classe membre d'un autre espace de nom ou d'aucun doit forcément utiliser le nom absolu des classes, c'est à dire : `MesClasses.ClasseUne`, `MesClasses.ClasseDeux` ou bien utiliser la directive using :

```
using namespace;
```

Exemple :

```
using MesClasses;

public class ClasseTest
{
    ClasseUne objet1;
    ClasseDeux objet2;
}
```

Équivaut à :

```
public class ClasseTest
{
    MesClasses.ClasseUne objet1;
    MesClasses.ClasseDeux objet2;
}
```

L'exemple du chapitre "Un premier programme" utilise déjà un espace de nom : la classe `Console` est définie dans l'espace de noms `System`.

Imbrication des espaces de noms

L'imbrication des espaces de noms est possible :

```
namespace MesClasses
{
    namespace Calcul
    {
        ... // espace de nom  MesClasses.Calcul
    }
}
```

Ou directement :

```
namespace MesClasses.Calcul
{
    ... // espace de nom  MesClasses.Calcul
}
```

Espace de noms en plusieurs parties

Plusieurs fichiers sources peuvent contribuer à ajouter des déclarations à un même espace de noms.

Exemple :

images.cs
Classes de gestion des images dans l'espace de noms Exemple.Graphisme

```
using System;
namespace Exemple.Graphisme
{
    public class Image { ... }
    public class ImageBitmap : Image { ... }
    ...
}
```

couleur.cs
Classes de gestion des couleurs dans l'espace de noms Exemple.Graphisme

```
using System;
namespace Exemple.Graphisme
{
    public class Couleur { ... }
    public class ConversionCouleur { ... }
    ...
}
```

Alias d'espace de nom

Le mot clé `using` peut servir à assigner un autre nom (plus court en général) à un espace de nom.

Exemple :

```
using coll = System.Collections;

public class Exemple
{
    coll::Hashtable donnees = new coll::Hashtable();
}
```

Créer un alias d'espace de nom est également utile pour utiliser deux classes portant le même nom, situées dans deux espaces de nom différents.

Exemple :

```
namespace Gestion.Transport
{
    public class Ligne { ... }
}

namespace Graphique.Dessin
{
    public class Ligne { ... }
}
```

Si l'instruction `using` est utilisée deux fois, le nom `Ligne` est ambigu, comme l'explique l'erreur produite par le compilateur :

```
using Gestion.Transport;
using Graphique.Dessin;

public class Exemple
{
    Ligne exemple;
    // -> error CS0104: 'Ligne' is an ambiguous reference between
    //    'Gestion.Transport.Ligne' and 'Graphique.Dessin.Ligne'.
```

```
}
```

Dans ce cas, on peut utiliser l'instruction `using` pour l'un des deux espaces de nom et utiliser le nom complet pour l'autre espace de nom :

```
using Gestion.Transport;

public class Exemple
{
    Ligne transport;
    Graphique.Dessin.Ligne trace;
}
```

Il est également possible de définir deux alias d'espace de nom afin d'abréger les références:

```
using transport=Gestion.Transport;
using dessin=Graphique.Dessin;

public class Exemple
{
    transport::Ligne transport;
    dessin::Ligne trace;
}
```

Conflit de nom

Il est possible que le code source utilise des noms déjà utilisé comme espace de noms.

Dans l'exemple de code suivant, les noms `System` et `Console` font référence à des attributs de la classe `Classe`.

```
public class Classe
{
    private string System = "system";
    private string Console = "console";

    public void Afficher()
    {
        // System.Console.WriteLine("Exemple"); // provoque une erreur
    }
}
```

Il est toutefois possible d'utiliser `System.Console` en utilisant le préfixe `global::` qui permet de spécifier que l'on accède à l'espace de nom racine de tous les autres (implicite par défaut) :

```
public class Classe
{
    private string System = "system";
    private string Console = "console";

    public void Afficher()
    {
        global::System.Console.WriteLine("Exemple"); // OK
    }
```

```
}
```

Alias d'assemblages

Lorsqu'une application utilise des assemblages dont les noms de classes complets (espace de nom inclus) sont identiques (deux versions/implémentations d'une même classe), l'ambiguïté doit être résolue en utilisant des alias d'assemblages.

Pour utiliser deux versions d'une même classe définies dans les assemblages `grid.dll` (version 1.0, alias "GridV1") et `grid20.dll` (version 2.0, alias "GridV2"), il faut définir les alias à utiliser dans la ligne de commande de compilation :

Windows	`csc /r:GridV1=grid.dll /r:GridV2=grid20.dll ...`
Linux (Mono)	`gmsc -r:GridV1=grid.dll -r:GridV2=grid20.dll ...`

Cela crée les alias externes GridV1 et GridV2. Pour utiliser ces alias à partir d'un programme, il faut les référencer à l'aide du mot clé `extern` :

```
extern alias GridV1;
extern alias GridV2;
```

Ces deux instructions créent deux nouvelles racines d'espace de nom en plus de `global` :

```
extern alias GridV1;
extern alias GridV2;

public class Exemple
{
    GridV1::Grid grid_v1; // implémentation V1.0 (grid.dll)
    GridV2::Grid grid_v2; // implémentation V2.0 (grid20.dll)
}
```

Les variables et les constantes

Les variables

Une variable réserve une place en mémoire pour stocker des données : résultats d'expressions, données lues depuis un fichier, ... Elle est associée à un nom.

Déclaration

Une variable possède un nom et un type. Ce type détermine ce que peut stocker la variable : un nombre, une chaîne de caractère, un objet d'une classe particulière ...

La syntaxe est la suivante :

```
type nom [ = expression ] ;
```

Exemples :

```
double prix_unitaire;
int quantite = 50;
string article = "Pommes";
bool rabais = false;
```

Assigner une variable (initialiser) à la déclaration n'est pas obligatoire. Dans ce cas, elle possède la valeur par défaut correspondant au type :

- la valeur par défaut des types numériques et caractères (`char`) est zéro (0),
- la valeur par défaut du type booléen (`bool`) est `false`,
- la valeur par défaut des types références (objet, chaîne de caractère, tableaux) est `null`,
- la valeur par défaut des types énumérés (`enum`) est celle qui correspond au type sous-jacent (`int` par défaut).

Il est possible de regrouper la déclaration de plusieurs variables du même type en utilisant une virgule pour séparer les différentes variables.

Exemple :

```
int quantite = 50, nombre, quantite_en_stock = 100 ;
```

Ou en plaçant une variable par ligne pour plus de clarté :

```
int quantite = 50,
    nombre,
    quantite_en_stock = 100 ;
```

Utilisation

Une variable est utilisée pour stocker des résultats intermédiaires, des données qui serviront ultérieurement. La valeur d'une variable peut être modifiée autant de fois que nécessaire.

Exemple :

```
prix_unitaire = 1.50;
if (rabais) prix_unitaire = prix_unitaire - 0.20;
double prix_total = prix_unitaire * quantite ;
```

Portée d'une variable

La portée d'une variable est l'ensemble des emplacements dans le code où cette variable est accessible. En dehors de cette portée, l'utilisation du même nom correspondra à une autre variable ou plus souvent à aucune variable (erreur de compilation dans ce cas).

La portée d'une variable correspond au bloc d'accolades où elle est déclarée. Une variable membre d'une classe est déclarée au niveau de la classe et est accessible depuis les autres membres de la classe (méthodes,

indexeurs, propriétés, ...). Une variable locale est déclarée à l'intérieur d'un bloc d'accolades (celui de la méthode/propriété membre, ou un bloc interne à celle-ci).

Exemple :

```
public class Exemple
{
    // variable membre de la classe :
    int nombre = 5; // portée : la classe Exemple

    public int MethodeUne()
    {
        // variable locale à la méthode :
        int nombre = 10; // portée : la méthode MethodeUne()

        return nombre +     // la variable locale
               this.nombre; // la variable membre
        // retourne 10 + 5 , c'est à dire 15
    }

    public int MethodeDeux()
    {
        System.Console.WriteLine("Nombre = " + nombre); // la variable membre
        // Nombre = 5
        {

            int nombre = 20; // variable locale au bloc d'accolades
            System.Console.WriteLine("Nombre = " + nombre); // la variable locale
            // Nombre = 20

        } // la variable locale est hors de portée
        System.Console.WriteLine("Nombre = " + nombre); // la variable membre
        // Nombre = 5
    }
}
```

Durée de vie d'une variable

La durée de vie d'une variable est la période durant laquelle une variable existe, c'est à dire conserve son emplacement en mémoire et durant laquelle elle peut donc stocker des données.

Variable en lecture seule

Une variable peut être déclarée en lecture seule en utilisant le mot-clé `readonly`.

Exemple :

```
readonly double taux_tva = 19.6;
```

Il n'est pas obligatoire d'initialiser une variable en lecture seule lors de sa déclaration. Ce qui permet de déterminer la valeur en fonction d'autres données, ou de lire sa valeur depuis un fichier par exemple, et d'empêcher sa modification après affectation.

Exemple :

```
class Facture
```

```
{
    public readonly double taux_tva;
    public Facture(bool taux1)
    {
        if (taux1) taux_tva = 19.6;
        else taux_tva = 5.5;
    }
}
```

Les constantes

Une constante nommée est associée à une valeur pour toute la durée de l'application. Sa valeur ne peut changer.

Déclaration

La syntaxe est similaire à celle de la déclaration d'une variable, excepté que le mot clé `const` précède la déclaration, et que l'initialisation à la déclaration est obligatoire :

```
const type nom = expression ;
```

Exemple :

```
const double prix_unitaire_unique = 1.50 ;
const double taux_euro_en_francs = 6.55957 ;
```

Comme les variables il est également possible de regrouper la déclaration de plusieurs constantes du même type :

```
const double
    prix_unitaire_unique = 1.50 ,
    taux_euro_en_francs = 6.55957 ;
```

N.B.: Une constante est implicitement statique. Il est donc inutile d'ajouter le mot-clé `static`, sinon le compilateur génère une erreur.

Types des constantes

Le mot `const` ne s'applique qu'aux types standards numériques, booléens, `string`, et énumérations.

Portée et durée de vie d'une constante

Les mêmes règles que celles pour les variables s'appliquent également aux constantes.

Les types de base et les déclarations

En C#, il existe deux catégories de type :

- Les types *valeurs* qui stockent directement la valeur des données (un entier, une chaîne de caractères, une structure),
- Les types *références* qui stockent une référence vers la valeur des données (un tableau, un objet, une interface). Les variables de ce type se distinguent par le fait qu'elles valent initialement `null` et qu'il faut explicitement leur allouer de l'espace mémoire avec `new`. Les données référencées par ces variables sont donc soumises au *garbage collector* quand plus aucune référence n'existe, afin de libérer la place mémoire occupée.

Un type de base est un type simple *valeur*.

Liste des types de base

Type	Classe	Description	Exemples
bool	System.Bool	Booléen (vrai ou faux : `true` ou `false`)	`true`
char	System.Char	Caractère Unicode (16 bits) (ou plus précisément, une unité de code (code unit) Unicode [1]).	`'A' 'λ' 'ω'`
sbyte	System.SByte	Entier signé sur 8 bits (1 octet)	`-128`
byte	System.Byte	Entier non signé sur 8 bits (1 octet)	`255`
short	System.Int16	Entier signé sur 16 bits	`-129`
ushort	System.UInt16	Entier non signé sur 16 bits	`1450`
int	System.Int32	Entier signé sur 32 bits	`-100000`
uint	System.UInt32	Entier non signé sur 32 bits	`8000000`
long	System.Int64	Entier signé sur 64 bits	`-2565018947302L`
ulong	System.UInt64	Entier non signé sur 64 bits	`8000000000000L`
float	System.Single	Nombre à virgule flottante sur 32 bits	`3.14F`
double	System.Double	Nombre à virgule flottante sur 64 bits	`3.14159`
decimal	System.Decimal	Nombre à virgule flottante sur 128 bits	`3.1415926M`
string	System.String	Chaîne de caractères unicode[1]	`"C♯ ≈ C#"` `"€ ≠ ℕ"` `"Chaîne ♫ ♬ ♭ ♮"` `"συμβόλων"` `"C:\\windows \\system32"` `@"C:\windows \system32"`

Syntaxe des chaînes de caractères

Comme illustré par l'exemple du tableau ci-dessus, une chaîne de caractères peut avoir deux syntaxes différentes :

- Syntaxe avec caractères d'échappement : La chaîne est entourée par les guillemets et l'anti-slash introduit un caractère spécial (`\n`, `\t` ...) ou empêche son interprétation (guillemet `\"` et anti-slash `\\`)
- Syntaxe *verbatim* : La chaîne est précédée d'un caractère arobase, et l'anti-slash n'est pas interprété. Si un guillemet doit faire partie de la chaîne de caractère, il faut le doubler `""`.

Exemples :

```
"un guillemet \", un anti-slash \\, un retour à la ligne\n"
@"un guillemet "", un anti-slash \, un retour à la ligne
"
```

La syntaxe *verbatim* simplifie la frappe des chemins de fichiers qui utilisent le caractère anti-slash comme séparateur :

```
@"C:\program files\monapp\monfichier.txt"
```

Auto-boxing/unboxing

Chaque type de données correspond à une classe de l'espace de nom `System`. La conversion entre le type et une instance de cette classe est implicite et invisible.

Ainsi, on peut appeler des méthodes sur les types simples, affecter une constante où une classe de type est attendue (et *vice versa*), ...

Exemple :

```
int a = 25;
string message = 36.ToString(); // convertit 36 en chaîne de caractères,
                                // méthode définie dans la classe System.Int32
UInt32 b = 50;
```

Types *nullable*

Le langage C#2.0 introduit la possibilité pour les types simples de valoir `null`. Cette fonctionnalité permet une meilleure interopérabilité avec les bases de données qui utilisent ce type de données.

Pour qu'une variable puisse valoir `null` (*nullable*), il faut que le type soit suivi d'un point d'interrogation. Par exemple :

```
int? numero = null;
```

Il est donc possible de tester qu'une variable de ce type vaut `null` :

```
if (numero==null) numero = 50;
else numero++;
```

Le nouvel opérateur `??` (double point d'interrogation) permet de sélectionner l'opérande de gauche s'il ne vaut pas `null`, ou l'opérande de droite sinon :

```
valeur_ou_null??valeur_si_null
```

Cet opérateur est donc pratique à utiliser avec les types *nullables* pour obtenir une valeur par défaut :

```
Console.WriteLine("Numéro : "+( numero??50 ));
```

Valeur par défaut

L'opérateur `default` retourne la valeur par défaut du type spécifié. Il s'agit de la valeur quand une variable de ce type n'est pas initialisée (`0` pour les nombres, `null` pour les types références).

Exemple:

```
public int absolu(int? valeur)
{
    if (valeur==null) return default(int);
    else return (valeur<0) ? -valeur : valeur;
}
```

L'utilité de cet opérateur est plus évident avec l'utilisation de types génériques.

Obtenir le type

L'opérateur `typeof` retourne une instance de la classe System.Type pour le type spécifié entre parenthèses.

Exemple:

```
Type t=typeof(int);
```

L'utilité de cet opérateur est plus évident avec l'utilisation de types génériques.

Notes de bas de page

1. Unicode différence l'unité de code (occupant un nombre déterminé de bit), du caractère lui-même (pouvant être composé de plusieurs points de code, et de plusieurs unités de code). Ces notions sont décrites de manière détaillée dans le livre À la découverte d'Unicode

Les tableaux

Un tableau regroupe plusieurs données d'un même type, dans un ensemble ordonné et indicé par un entier. En C#, comme avec la plupart des langages de programmation modernes, le premier élément porte l'indice 0.

Déclaration d'un tableau

Les crochets ajoutés à la fin d'un type indique qu'il s'agit d'un tableau.

Exemple :

```
int[] eurepetea;
```

La variable `entiers` est un tableau de nombres entiers. Le nombre d'éléments du tableau n'est pas spécifié à la déclaration, mais lors de l'allocation du tableau.

Allocation d'un tableau

Les tableaux font partie des types références. Il n'est donc pas alloué par défaut (référence `null`).
L'allocation se fait en utilisant le mot clé `new` :

```
new type[taille]
```

Exemple :

```
int[] eurepetea= new int[10]; // un tableau de 10 entiers
```

Il est également possible de l'allouer ailleurs que dans la déclaration :

```
int[] eurepetea;

...

eurepetea= new int[10]; // un tableau de 10 entiers
```

Une fois la place du tableau réservée en mémoire, celui-ci ne contient que des valeurs par défaut, c'est à dire que chaque élément d'un tableau de numériques (entiers, réels, ...) vaut `0`, pour un tableau de `bool` chaque élément vaut `false`, et pour un tableau de références (objet, interface, tableau) chaque élément vaut `null`.

Pré-initialisation

Il est également possible de définir directement les valeurs que le tableau contient. Ces valeurs doivent être comprises entre des accolades et séparées par une virgule. Le compilateur détermine le nombre d'éléments à allouer d'après la liste d'éléments spécifiée à la suite de l'instruction d'allocation.

Exemples :

- A la déclaration du tableau :

```
int[] eurepetea= new int[] { 10,15,20,25,30,35,40,45 };
```

- Hors déclaration :

```
eurepetea= new int[] { 10,15,20,25,30,35,40,45 };
```

Seule la déclaration peut omettre l'instruction d'allocation du tableau avant les accolades :

```
int[] eurepetea= { 10,15,20,25,30,35,40,45 };
```

Dans ce cas, le compilateur alloue implicitement un tableau du même type que la variable déclarée, pour le nombre d'éléments placés entre les accolades.

Accès aux éléments

L'accés (lecture et écriture) aux éléments du tableau se fait en utilisant le nom du tableau suivi des crochets encadrant l'indice de l'élément accédé :

```
eurepetea[0] = 10;
eurepetea[1] = 15;
Console.WriteLine("1er entier = " + eurepetea[0] );
```

L'indice spécifié est en fait une expression. Il est donc possible d'utiliser une variable comme indice, par exemple, pour parcourir les éléments du tableau :

```
for(int i=0 ; i<2 ; i++)
    Console.WriteLine("entier n°" + i + " = " + eurepetea[ i ] );
```

Taille d'un tableau

L'attribut `Length` donne la taille du tableau. Ce qui est pratique pour le parcourir :

```
Console.WriteLine("Le tableau contient " + entiers.Length + " entiers :");

for(int i=0 ; i<entiers.Length ; i++) // i : indice dans le tableau
    Console.WriteLine("  ["+i+"] = " + entiers[i] );
```

Si l'indice n'est pas nécessaire durant le parcours du tableau, il est plus simple d'utiliser `foreach` :

```
Console.WriteLine("Le tableau contient " + entiers.Length + " entiers :");

foreach(int n in entiers) // n : élément du tableau
    Console.WriteLine("  " + n );
```

Tableaux multi-dimensionnels

Les tableaux vus jusqu'à présent étaient des tableaux uni-dimensionnels : ils n'ont qu'une seule dimension, c'est à dire un seul indice.

Certaines applications nécessitent des tableaux à deux indices ou davantage. Par exemple, une image est représentée par un tableau de couleurs indicé par l'abscisse (x) et l'ordonnée (y).

Un tableau multi-dimensionnel utilise la même syntaxe, en séparant les indices par une virgule.

```
int[,] image = new int[ 32, 16 ];
```

```
// un tableau de 32*16 entiers, soit 512 entiers
image[1,5] = 0;
```

Un tableau de tableaux (appelé *jagged array* ou tableau déchiqueté) utilise la syntaxe des tableaux uni-dimensionnels en utilisant un tableau comme type de base :

```
int[][] ensembles = new int[10][]; // un tableau de 10 tableaux d'entiers
ensembles[0] = new int[11];        // dont le premier contient 11 entiers
ensembles[1] = new int[] { 2, 3 }; // le deuxième : 2 entiers (2 et 3)
// ensembles[2] à ensembles[9] valent null car non alloués
ensembles[1][1] = 4; // remplace la valeur 3 par 4
```

Chacun des sous-tableaux doit être alloué séparemment, et rien n'oblige à ce qu'ils aient tous la même dimension (d'où l'appellation de tableau déchiqueté).

Il est également possible de combiner les possibilités :

```
int[,][] donnees = new int[10,20][];
donnees[1,5] = new int[25];
donnees[1,5][10] = 12;
```

Les objets

Un objet est un type référence. Il s'agit d'une instance de classe.

Le type object

La classe `System.Object` est équivalente au type `object`. Il s'agit de la classe de base de toutes les autres classes.

L'auto-boxing/unboxing permet d'utiliser le type `object` pour n'importe quel type de valeur :

```
object obj;
obj = 10;           // int -> Int32 -> object
obj = "Une chaîne"; // string -> String -> object
```

Créer un nouvel objet

Créer un nouvel objet est appelé *instancier une classe* et utilise l'opérateur `new` :

```
classe variable=new classe(arguments...);
```

Cet opérateur effectue les opérations suivantes :

- allocation d'une zone mémoire d'une taille suffisante pour accueillir les attributs définis dans la classe spécifiée,

- initialisation des attributs à leur valeur par défaut (définie dans la classe ou 0 sinon),
- appel au constructeur acceptant les arguments spécifiés.

L'appel peut échouer pour plusieurs raisons :

- la classe spécifiée n'a pas pu être chargée,
- l'espace mémoire libre est insuffisant,
- une exception a été lancée par le constructeur.

Dans tous les cas une exception est lancée.

Les classes

La notion de classe est à la base de la programmation orientée objet. Elle définit un type d'objet. Ce type rassemble des variables (créées pour chaque objet et appelées « attributs ») et des fonctions (appelées dans ce cas « méthodes »).

Ces attributs et méthodes sont les « membres » de la classe. Ils peuvent avoir chacun un niveau de protection différent.

Déclarer une classe

La déclaration d'une classe utilise le mot clé `class` :

```
class nom_de_la_classe
{
}
```

Les membres sont définis à l'intérieur des accolades.

Exemple :

```
class Fraction
{
    int numerateur,
        denominateur;

    void multiplier(Fraction autre)
    {
        this.numerateur *= autre.numerateur;
        this.denominateur *= autre.denominateur;
    }
}
```

Membres d'une classe

Les membres d'une classe sont de différents types :

- Les fonctions, appelées méthodes, traitent les données, appellent d'autres méthodes, retournent éventuellement une valeur ;

- Les variables, appelées attributs, stockent les données ;
- Les propriétés sont des méthodes spéciales utilisées comme des variables.

Membres statiques de classe et membres d'instance

Un membre statique de classe est un membre déclaré avec le mot clé `static`. Il est défini une seule fois en mémoire, et il n'est pas nécessaire de créer un nouvel objet pour qu'il soit créé et utilisable.

Un membre d'instance est déclaré sans le mot clé `static`. Il n'est créé que pour chaque objet (une instance) créé par l'opérateur `new`.

Exemple :

```
public class UnExemple
{
    int numero; // Défini pour chaque objet
    static int numero_suivant; // Défini pour la classe
}
```

```
Console.WriteLine("N° suivant = " + UnExemple.numero_suivant );
```

```
Console.WriteLine("N° = " + UnExemple.numero );
// Erreur de compilation, car la variable n'est pas définie pour la classe
```

```
UnExemple InstanceDUnExemple = new UnExemple();
Console.WriteLine("N° = " + InstanceDUnExemple.numero );
```

Classe statique

Une classe statique ne contient que des membres statiques, et ne peut être instanciée. Le mot clé `static` précède la déclaration de cette classe.

Exemple :

```
public static class UneClasseStatique
{
    public static void Afficher(string message)
    {
        // ...
    }
}
```

```
UneClasseStatique.Afficher("Un exemple de message");
```

Instance d'une classe

L'instance d'une classe est aussi appelée un objet. Une variable de ce type est un type référence, c'est à dire que la référence est nulle par défaut. Il faut donc utiliser le mot clé `new` pour allouer une instance de la classe.

Exemple :

```
UnExemple objetExemple; // par défaut : null
// Equivaut à   UnExemple objetExemple = null;
objetExemple = new UnExemple();
```

Le mot clé new est suivi du nom de la classe et d'une liste de paramètres entre parenthèses. Il s'agit en fait d'un appel au constructeur (abordé plus loin dans ce chapitre) initialisant l'objet créé.

La référence nulle

Toute variable de type objet est en fait une référence initialisée à `null` par défaut. Cette référence peut être utilisée quel que soit le type d'objet.

Il est souvent nécessaire de tester si la référence est nulle avant d'accéder à un membre de l'objet référencé. Si le test n'est pas fait et que la référence est nulle, une exception est levée.

Niveaux de protection

Le niveau de protection est spécifié par un mot clé placé avant la déclaration d'une classe ou d'un membre de classe (attribut, méthode, propriété, ...) :

`public`
: Accès autorisé à tous ;

`private`
: Accès depuis la classe seulement ;

`protected`
: Accès depuis la classe et ses sous-classes seulement ;

`internal`
: Accès depuis l'assembly seulement.

Exemple :

```
public class Fraction
{
    private int
        numerateur,
        denominateur;

    public void multiplier(Fraction autre)
    {  }
}
```

En général, les règles de l'encapsulation sont les suivantes :

- Toute méthode est publique, à moins qu'elle ne soit destinée à être exclusivement appelée dans la classe, auquel cas est peut être privée ou protégée,
- Les attributs sont privés. Si un accès est nécessaire, il faut ajouter des méthodes d'accès («accesseurs»), voire une propriété.

Constructeur

Un constructeur est une méthode spéciale portant le nom de la classe, ne retournant aucune valeur, chargée d'initialiser l'instance. Un constructeur peut avoir des paramètres permettant de définir les valeurs initiales des attributs du nouvel objet.

Une classe possède souvent plusieurs constructeurs. Chacun possédant une liste d'arguments différente.

Exemple

Pour initialiser une fraction :

```
public class Fraction
{
    private int
        numerateur,
        denominateur;

    // Constructeur
    public Fraction(int numerateur, int denominateur)
    {
        // this est utilisé pour désigner les attributs de l'objet
        // plutôt que les paramètres
        // recopie des paramètres :
        this.numerateur = numerateur;
        this.denominateur = denominateur;
    }

    public void multiplier(Fraction autre)
    {  }
}
```

Utilisation :

```
Fraction estimation_pi = new Fraction( 355 , 113 );
```

Constructeur par défaut

Le constructeur par défaut est celui qui n'a aucun paramètre. Si aucun constructeur n'est déclaré dans une classe, le compilateur crée un constructeur par défaut qui ne fait rien, laissant les attributs à leur valeur par défaut (dépendant du type : 0 pour les numériques, null pour tous les autres).

En fait, ce constructeur par défaut appelle le constructeur sans argument (qu'il soit par défaut ou explicitement implémenté) de la classe de base (`System.Object` par défaut). Voir le chapitre sur l'héritage de classe.

Constructeur statique

Il est possible d'exécuter un code lors de la première utilisation de la classe. Cette initialisation peut être vue comme un constructeur statique :

```
public class Fraction
{
    private static int valeur;

    static Fraction()
```

```
    {
        valeur = 1;
    }
}
```

Les fonctions

Une fonction étant toujours déclarée dans une classe, elles sont plutôt appelées « méthodes ».

Déclarer une méthode

Une méthode possède un nom, une liste de paramètres, et un type de retour.

La syntaxe de déclaration est la suivante :

```
[accès] type_retour nom_méthode ( type argument, ... )
{
    instructions
}
```

Où *accès* est optionnel et définit le mode d'accès (public, privé ou protégé), le type de méthode (statique ou d'instance). Voir le chapitre sur les classes.

type_retour spécifie le type de la valeur retournée par la fonction. Ce type peut être *void* (vide) pour indiquer que la fonction ne retourne aucune valeur.

La fonction retourne une valeur en utilisant le mot clé `return`. L'instruction `return` interrompt l'exécution de la fonction, retourne la valeur de l'expression qui suit, et l'exécution du code ayant appelé la fonction se poursuit.

La syntaxe est :

```
return expression;
```

Si la fonction ne retourne rien, l'instruction `return` n'est pas obligatoire, mais si elle est utilisée, aucune expression ne doit être spécifiée. Si la fonction ne retourne rien on appelera cette fonction une procédure

Exemple :

```
// Fonction qui retourne la concaténation
// de la chaîne de caractères a à la chaîne de caractères b
string Concatener(string a, string b)
{
    return a+b;
}
```

Appeler une méthode

L'appel à une méthode se fait en spécifiant son nom, suivi des arguments entre parenthèses :

```
nom_methode ( expression, ... )
```

Si la fonction retourne une valeur, l'appel à celle-ci peut être utilisé dans une expression.

Exemple :

```
string c = Concatener ( "abc" , "def" );
// c vaut "abcdef"
```

Dans cet exemple, la méthode appelée est nommée `Concatener` car située dans la même classe que l'appel. Cependant, toute méthode s'applique à un objet. Celui-ci doit être spécifié avant le nom de la fonction, et séparé par le caractère point (.). Par défaut, il s'agit de l'objet `this` (objet courant).

Passage de paramètres à une méthode

Les paramètres passés à une méthode peuvent être passés de trois manière différentes :

- un paramètre en lecture seule (transmission d'information) : passage par valeur,
- un paramètre référençant une variable pour y mettre une valeur de retour : paramètre out,
- un paramètre référençant une variable pour transmettre une information et y mettre une valeur de retour : paramètre ref.

Passage par valeur

Il s'agit d'un paramètre servant à transmettre une information à la méthode appelée, comme dans les exemples précédents. Si la méthode modifie sa valeur, la modification est locale à la fonction uniquement. Un paramètre normal n'est précédé d'aucun mot-clé particulier.

Exemple :

```
private void methodeTest(int nombre)
{
    Console.WriteLine("fonctionTest : le nombre vaut " + nombre);
    nombre = 100;
    Console.WriteLine("fonctionTest : le nombre vaut " + nombre);
}

private void testAppel()
{
    int n = 5;
    Console.WriteLine("testAppel : le nombre vaut " + n);

    methodeTest(n); // Appel à la méthode

    Console.WriteLine("testAppel : le nombre vaut " + n);

    // On peut passer une expression ou une constante
    methodeTest( 25 + 5 ); // Appel à la méthode
}
```

Ce programme affiche :

```
testAppel : le nombre vaut 5
fonctionTest : le nombre vaut 5
fonctionTest : le nombre vaut 100
testAppel : le nombre vaut 5
fonctionTest : le nombre vaut 30
fonctionTest : le nombre vaut 100
```

Paramètre out

Un paramètre `out` ne sert à la fonction qu'à retourner une valeur. L'argument transmis doit référencer une variable ou un élément de tableau. Ce paramètre est précédé du mot-clé `out` à la fois lors de sa déclaration, et lors de l'appel à la méthode.

La variable référencée n'a pas besoin d'être initialisée auparavant. La fonction doit obligatoirement affecter une valeur à la variable référencée.

Exemple :

```
private void methodeTest(out int resultat)
{
    //Console.WriteLine("fonctionTest : le nombre vaut " + resultat);
    resultat = 100;
    //Console.WriteLine("fonctionTest : le nombre vaut " + resultat);
}

private void testAppel()
{
    int n;
    methodeTest(out n); // Appel à la méthode

    Console.WriteLine("testAppel : le nombre vaut " + n);

    // On ne peut pas passer une expression ou une constante
    //methodeTest( 25 + 5 ); // <- erreur de compilation
}
```

Ce programme affiche :

```
testAppel : le nombre vaut 100
```

Paramètre ref

Un paramètre `ref` est une combinaison des deux types de paramètres précédents. L'argument transmis doit référencer une variable ou un élément de tableau qui doit être initialisé auparavant. Ce paramètre est précédé du mot-clé `ref` à la fois lors de sa déclaration, et lors de l'appel à la méthode. La méthode n'est pas obligée de modifier la valeur contenue dans la variable référencée.

Exemple :

```
private void methodeTest(ref int resultat)
{
    Console.WriteLine("fonctionTest : le nombre vaut " + resultat);
    resultat = 100;
    Console.WriteLine("fonctionTest : le nombre vaut " + resultat);
```

```
}

private void testAppel()
{
    int n = 5;
    Console.WriteLine("testAppel : le nombre vaut " + n);

    methodeTest(ref n); // Appel à la méthode

    Console.WriteLine("testAppel : le nombre vaut " + n);

    // On ne peut pas passer une expression ou une constante
    //methodeTest( 25 + 5 ); // <- erreur de compilation
}
```

Ce programme affiche :

```
testAppel : le nombre vaut 5
fonctionTest : le nombre vaut 5
fonctionTest : le nombre vaut 100
testAppel : le nombre vaut 100
```

C# 4.0 Méthode COM : ref optionnel

L'appel à une méthode d'interface COM utilisant un paramètre `ref` peut ne pas utiliser explicitement de variable.

Exemple :

La méthode COM suivante :

```
void Increment(ref int x);
```

Peut être appelée sans le mot clé `ref` :

```
Increment(0);
```

Ce qui est équivalent à :

```
int x = 0;
Increment(ref x);
```

Un paramètre `ref` est en général modifié par la méthode appelée. Ne pas utilier le mot clé `ref` lors de l'appel signifie que la nouvelle valeur retournée par la méthode est ignorée.

Nombre variable de paramètres

Une méthode peut posséder un nombre variable de paramètres, comme, par exemple, la méthode statique Format de la classe String :

```
string message = String.Format("A={0} et B={1}", a, b);
```

Ce genre de méthode possède un nombre de paramètres obligatoires, comme une méthode normale, suivis d'un nombre variable de paramètres (voire aucun). Ces paramètres en nombre variable sont en fait transmis sous la forme d'un tableau.

Une telle méthode utilise le mot clé `params` pour le dernier paramètre qui doit être un tableau.

Exemples de méthodes :

```
public string MessageDeLog(string format, params object[] parametres)
{
    return "LOG: "+String.Format(format, parametres);
}
```

```
public double Moyenne(params double[] nombres)
{
    double sum=0;
    if (nombres.Length==0) return 0;

    foreach(double d in nombres)
        sum += d;

     return sum / nombres.Length;
}
```

Pour le paramètre marqué `params`, il est possible de transmettre soit une liste d'argument, soit un tableau contenant ces arguments.

Exemples d'appels :

```
double a = Moyenne(3.0, 2.0, 5.14, 8.22, 6.37);
// Equivaut à :
double b = Moyenne( new double[]{ 3.0, 2.0, 5.14, 8.22, 6.37 } );
```

C# 4.0 Paramètres optionnels

Les paramètres peuvent être optionnels lors de l'appel à la méthode lorsque ceux-ci sont définis avec une valeur par défaut.

Exemple :

```
public void afficher(string texte, int largeur = 80, bool afficher_sommaire = false)
{ /* ... */ }
```

Cette méthode peut être appelée de différentes façons :

```
afficher("Ceci est un exemple", 70, true);
afficher("Ceci est un exemple", 70);  // 70,false
afficher("Ceci est un exemple");      // 80,false
```

Cependant, l'appel suivant ne compile pas :

```
afficher("Ceci est un exemple", true); // largeur = true  incorrect
```

La solution est d'utiliser les paramètres nommés.

C# 4.0 Paramètres nommés

Lors de l'appel à une méthode, les paramètres peuvent être nommés afin de pouvoir les passer dans un ordre quelconque et permettre également de ne pas fournir certains paramètres optionnels.

La méthode `afficher` de l'exemple de la section précédente peut donc être appelée de cette façon :

```
afficher("Ceci est un exemple", afficher_sommaire: true);
afficher(afficher_sommaire: true, texte: "Ceci est un exemple");
```

Surcharge de méthode

Une méthode peut être surchargée (*overload* en anglais), c'est à dire qu'il peut exister au sein de la même classe plusieurs méthodes portant le même nom, à condition qu'elles soient différenciables par leur signature. La signature d'une méthode correspond aux types et nombre de paramètres acceptés par celle-ci.

Exemple :

```
public int Ajouter ( int valeur1, int valeur2 )
{
    return valeur1 + valeur2;
}

public double Ajouter ( double valeur1, double valeur2 )
{
    return valeur1 + valeur2;
}
```

Le compilateur détermine la méthode à appeler en fonction du type des arguments passés à la méthode.

Exemple :

```
Console.WriteLine("(entiers)  2 + 5 = " + Ajouter( 2, 5 ));
Console.WriteLine("(réels)  2.0 + 5.0 = " + Ajouter( 2.0, 5.0 ));
```

Le type de retour ne fait pas partie de la signature, car la valeur retournée par une méthode appelée peut être ignorée. Si deux méthodes portent le même nom et ont la même signature, le compilateur génère une erreur.

Exemple :

⚠ Ce code contient **une erreur volontaire** !

Les deux méthodes ont la même signature.

```
public double Longueur ( string chaine )
```

```
{
    return (double) chaine.Length;
}

public int Longueur ( string chaine )
{
    return (int) chaine.Length;
}
```

Propriétés et indexeurs

Les propriétés

Une propriété est une valeur qui peut être lue ou modifiée, comme une variable. Ces deux opérations sont en fait réalisées par les accesseurs `get` et `set`. Si l'un de ces deux accesseurs est manquant, la propriété sera alors soit en lecture seule, soit en écriture seule.

Syntaxe

```
Type Nom_de_la_propriété
{
    get // propriété lue
    {
        code retournant une valeur du Type spécifié
    }
    set // propriété modifiée
    {
        code utilisant le paramètre prédéfini "value"
    }
}
```

Le code contenu dans chaque bloc est le même que celui que l'on placerait pour implémenter les méthodes suivantes :

```
Type getNom_de_la_propriété()
{
    code retournant une valeur du Type spécifié
}
void setNom_de_la_propriété(Type value) // propriété modifiée
{
    code utilisant le paramètre prédéfini "value"
}
```

Exemple

```
private string _message;
public string Message
{
    get
    {
        return _message;
```

```
    }
    set
    {
        _message = value;
    }
}
```

Utilisation :

```
Message = "Test de la propriété" ; // <- accesseur set
Console.WriteLine( Message ); // <- accesseur get
Message += " et message ajouté"; // <- accesseurs get et set
```

Noms réservés

Depuis la version 2003 du compilateur, lorsqu'une propriété est créée, deux noms de méthodes sont réservés pour les deux accesseurs :

- *type* get_*Nom_propriété*()
- void set_*Nom_propriété*(*type* value)

La classe ne peut donc avoir de méthodes portant l'une de ces deux signatures.

Les indexeurs

Un indexeur est une propriété spéciale qui permet d'utiliser une instance de la classe comme un tableau, en utilisant les crochets.

Syntaxe

```
Type_élément this[ Type_index index ]
{
    get // élément [index] lu
    {
        Code retournant une valeur du Type_éléments spécifié
        dont l'index est dans le paramètre index
    }
    set // élément [index] modifié
    {
        Code utilisant le paramètre prédéfini "value"
        pour le stocker à l'index spécifié par le paramètre index
    }
}
```

Type_élément
: Type de chaque élément du tableau virtuel.

Type_index
: Type de l'indice spécifié entre crochets.

index
: Variable contenant la valeur de l'indice de l'élément lu ou modifié.

L'index peut avoir un autre type que `int`. C'est le cas des tables de hashage de l'espace de nom `System.Collections`.

Exemple

```
public class TableauVirtuel
{
    public string this[int index]
    {
        // lecture seule car pas d'accesseur set
        get
        {
            return "Elément"+index.ToString();
        }
    }
}
...
TableauVirtuel tab=new TableauVirtuel();
Console.WriteLine("tab[15] = " + tab[15] );
// affiche Elément15
```

Les opérateurs

Les opérateurs sont utilisés dans les expressions :

- la plupart d'entre eux nécessite deux opérandes comme l'addition (`a + b`) et sont appelés opérateurs binaires ;
- d'autres ne nécessitent qu'un seul opérande situé juste après celui-ci tel la négation booléenne (`!ok`) et sont appelés opérateurs unaires ;
- enfin, l'opérateur `?:` est le seul opérateur ternaire, c'est à dire nécessitant trois opérandes.

Les opérateurs arithmétiques

Parmi les opérateurs arithmétiques, il y a les quatre opérateurs de base classiques :

- l'addition : `5 + 2` (7)
- la soustraction : `7 - 5` (2)
- la multiplication : `3 * 4` (12)
- la division : `12 / 4` (3)

Cependant, la division sur deux nombres entiers produit un nombre entier. Le reste de cette division est donné par l'opérateur modulo : `13 % 3` (1, car 13 = 3*4 + **1**) La division donnera un nombre à virgule flottante si au moins l'un des deux opérandes est de ce type : `12.0 / 5` donnera `2.4`.

Les opérateurs de comparaison

Chaque opérateur de comparaison retourne un booléen (`true` ou `false`) déterminant si la condition est vraie ou fausse :

- L'opérateur d'égalité : `a == b`
- L'opérateur d'inégalité : `a != b`
- L'opérateur *inférieur à* : `a < b`
- L'opérateur *supérieur à* : `a > b`

- L'opérateur *inférieur ou égal à* : `a <= b`
- L'opérateur *supérieur ou égal à* : `a >= b`

Les opérateurs booléens

Une expression booléenne (telle les comparaisons) sont des conditions qui peuvent être combinées avec les opérateurs suivants :

- L'opérateur *non* retourne le contraire de l'opérande (vrai pour faux, et faux pour vrai). Exemple : `!(a==5)`
- L'opérateur *et* ne retourne vrai que si les deux opérandes sont vrais. Exemple : `(a==5) && (b==0)`
- L'opérateur *ou* retourne vrai si l'un des deux opérandes est vrai. Exemple : `(a==5) || (b==0)`
- L'opérateur *ou exclusif* retourne vrai si un seul des deux opérandes est vrai, c'est à dire qu'il retourne vrai pour les couples d'opérandes (vrai, faux) et (faux, vrai). Comme dans de nombreux langages, il n'existe pas d'opérateur ou-exclusif spécifique. Mais on peut utiliser l'opérateur de comparaison différent pour comparer les valeurs booléennes (false/true). Exemple : `(a==5) != (b==0)`

Les opérateurs de manipulation des bits

Les exemples montrent la représentation binaire des opérandes de type octet (`byte`), donc sur 8 bits.

Les opérateurs suivants manipulent les bits :

- L'opérateur *non* retourne l'inverse des bits de l'opérande (0 pour 1, et 1 pour 0). Exemple : `~ 0x93`

```
~ 1001 0011  (0x93)
= 0110 1100  (0x6C)
```

- L'opérateur *et* ne retourne que les bits à 1 communs aux deux opérandes. Exemple : `0x93 & 0x36`

```
  1001 0011  (0x93)
& 0011 0110  (0x36)
= 0001 0010  (0x12)
```

- L'opérateur *ou* retourne les bits à 0 communs aux deux opérandes. Exemple : `0x93 | 0x36`

```
  1001 0011  (0x93)
| 0011 0110  (0x36)
= 1011 0111  (0xB7)
```

- L'opérateur de décalage de bits vers la gauche, comme son nom l'indique, décale les bits de l'opérande vers la gauche, du nombre de bits spécifié par le second opérande. Les bits les plus à gauche sont donc perdus. Exemple : `0x93 << 2`

```
  10010011  (0x93)
  010011    <<2
= 01001100  (0x4C)
```

- L'opérateur de décalage de bits vers la droite, comme son nom l'indique, décale les bits de l'opérande vers la droite, du nombre de bits spécifié par le second opérande. Les bits les plus à droite sont donc perdus.

Exemple : `0x93 >> 2`

```
  10010011  (0x93)
    100100  >>2
= 00100100  (0x24)
```

Le test de type

Tester le type d'un objet permet de savoir si son type est d'une classe particulière ou une de ses sous-classes. Il permet également de savoir si sa classe implémente une interface particulière.

L'opérateur `is`

Cet opérateur permet de tester le type d'un objet. Le premier argument est l'objet à tester, le second doit être un `Type`. L'opérateur retourne une valeur de type `bool` : `true` si l'objet passé est du type spécifié, `false` sinon.

Syntaxe :

```
expression is type
```

Exemple :

```
Nombre nombre = new Entier(150);
if (nombre is Entier)
    Console.WriteLine("nombre entier");
```

Cet opérateur équivaut à comparer le type retourné par la méthode `GetType()` avec le type spécifié, tout en gérant le cas où la référence d'objet est nulle (`null`).

Les opérateurs de conversion

La conversion permet de modifier le type d'une expression ou d'une référence en un autre type (par exemple, convertir l'entier 5 en nombre à virgule flottante). Le langage C# dispose de deux opérateurs différents :

Les parenthèses

Les parenthèses permettent de convertir tout type en un autre. La conversion d'objet d'une classe en une autre n'est possible que si le type réel de l'expression convertie est une sous-classe du type spécifié, ou implémente l'interface spécifiée. Dans le cas contraire, l'opérateur lance une exception. La syntaxe est la suivante :

```
(nouveau_type)expression
```

Exemple :

```
int longueur = (int)( 10.2 * 3.1415 );
```

L'opérateur as

Contrairement à l'opérateur précédent, l'opérateur as ne fonctionne que sur les références d'objets. Si la conversion ne peut être effectuée (la nouvelle classe n'est pas une classe de base de la classe réelle, ou n'est pas une interface implémentée par la classe réelle), alors la valeur null est retournée (aucune exception lancée). La syntaxe est la suivante :

```
expression as nouveau_type
```

Exemple :

```
object o = "Chaîne de caractère dérive de la classe object" as object;
```

Les opérateurs d'affectation

L'affectation consiste à assigner une valeur (constante ou résultat d'une expression) à une variable.

L'affectation simple

L'opérateur = affecte le résultat de l'expression de droite à la variable située à gauche.

Exemples :

```
total = 0;
total = 1 + 2;
total = longueur + largeur;
this.article1.nom = "Livre";
```

N.B.: Cet opérateur est le seul opérateur d'affectation utilisable à la déclaration des variables : *type identifiant_variable = expression*;

Exemples :

```
int total = 0;
double prix_total_ttc = 1.196 * prix_total_ht;
string nom_article = "Livre";
object livre = new Livre("Sans titre");
```

L'affectation avec opération

Un tel opérateur effectue une opération utilisant la valeur actuelle d'une variable et affecte le résultat à cette même variable.

La syntaxe est la suivante :

variable ***opérateur*** **=** *expression*

Cette syntaxe est l'équivalent de la suivante :

variable **=** *variable* ***opérateur*** *expression*

Ces opérateurs d'affectation sont les suivants :

Opérateur	Exemple	Équivalent
+=	`a += 5;`	`a = a + 5;`
-=	`a -= 5;`	`a = a - 5;`
*=	`a *= 5;`	`a = a * 5;`
/=	`a /= 5;`	`a = a / 5;`
%=	`a %= 5;`	`a = a % 5;`
<<=	`a <<= 5;`	`a = a << 5;`
>>=	`a >>= 5;`	`a = a >> 5;`
&=	`a &= 5;`	`a = a & 5;`
^=	`a ^= 5;`	`a = a ^ 5;`
|=	`a |= 5;`	`a = a | 5;`

N.B.: Un tel opérateur d'affectation ne peut être utilisé à la déclaration d'une variable car celle-ci n'existe pas encore lors de l'évaluation de l'expression.

La liste complète des opérateurs du C#

La liste ci-dessous présente les différents opérateurs du language C# avec leur associativité dans l'ordre de leur priorité (du premier évalué au dernier). Les opérateurs situés dans le même bloc ont la même priorité.

Code de couleur :

- Les opérateurs en rouge ne peuvent être surchargés.
- Les opérateurs en bleu ne peuvent être surchargés de la manière classique (mot-clé `operator`), mais d'une autre manière.

Opérateurs	Description	Associativité
`::`	Qualificateur d'alias d'espace de noms	de gauche à droite

`()`
 	
`[]`
`.`
`->`
	Parenthèses pour évaluer en priorité
	
	Tableau
	Sélection d'un membre par un identificateur (structures et objets)
	Sélection d'un membre par un pointeur (structures et objets)	

`++ --`
 	
`+ -`
`! ~`
`(type)`
`*`
`&`
`as`
`is`
`typeof`
`sizeof`
`new`
	Incrémentation post ou pré-fixée
	
	Opérateur moins unaire (change le signe de l'opérande)
	Non logique et Non binaire
	Conversion de type
	Déréférencement
	Référencement (adresse d'une variable)
	Conversion de type référence (pas d'exception lancée)
	Test de type
	Type d'une variable / expression
	Taille d'une variable / d'un type
	Allocation mémoire
		de droite à gauche

`* / %`	Multiplication, division, et modulo (reste d'une division)	de gauche à droite
`+ -`	Addition et soustraction	
`<< >>`	Décalage de bits vers la droite ou vers la gauche	

`< <=`
 	
`> >=`
	Comparaison “ inférieur strictement ” et “ inférieur ou égal ”
	
	Comparaison “ supérieur strictement ” et “ supérieur ou égal ”	

`== !=`	Condition “ égal ” et “ différent ”	
`&`	ET binaire	
`^`	OU exclusif binaire / logique	
`|`	OU binaire	
`&&`	ET logique booléen	
`||`	OU logique booléen	
`c?t:f`	Opérateur ternaire de condition	de droite à gauche

`=`
 	
`+= -=`
`*= /= %=`
`<<= >>=`
`&= ^= |=`
	Affectation
	
	Affectation avec somme ou soustraction
	Affectation avec multiplication, division ou modulo
	Affectation avec décalage de bits
	Affectation avec ET, OU ou OU exclusif binaires	

`,`	Séquence d'expressions	de gauche à droite

Erreurs dans les expressions

Différents types d'erreur peuvent survenir dans une expression :

- division par zéro : le résultat est indéfini,
- débordement du résultat : le nombre de bits du type accueillant le résultat est insuffisant.

Si l'expression où se situe l'erreur est constante (pas de variable dans l'expression), le résultat est évalué à la compilation, et produit en cas d'erreur une erreur de compilation.

Si au contraire, l'expression n'est pas constante, une erreur provoque le lancement d'une exception.

Ce comportement peut être modifié pour le débordement par les mots-clés `checked` et `unchecked`.

Vérification du débordement

Le débordement provoque une exception `System.OverflowException`.

```
checked( expression )
```

Il s'agit du contexte par défaut pour les expressions constantes, c'est à dire celles qui peuvent être évaluées lors de la compilation.

Non vérification du débordement

Le débordement provoque une copie partielle (les bits de poids faibles) du résultat de l'expression.

```
unchecked( expression )
```

Il s'agit du contexte par défaut pour les expressions non constantes, c'est à dire celles qui ne peuvent être évaluées que lors de l'exécution.

Structures de contrôle

Le language C# est un langage de programmation structuré. La structure d'un programme définit l'ordre d'exécution des instructions : condition, boucles, ...

Condition

Une instruction peut s'exécuter lorsqu'une condition est vraie. La syntaxe est la suivante :

```
if (expression)
    instruction
```

expression définit la condition et doit être du type `bool`, c'est-à-dire qu'elle ne peut valoir que `true` (vrai) ou `false` (faux).

L'instruction n'est exécutée que si la condition est vraie (`true`).

Exemple :

```
if (a==10) Console.WriteLine("a vaut 10");
```

Plusieurs instructions

Pour exécuter plusieurs instructions si la condition est vraie, *instruction* peut être remplacé par un bloc d'instructions entre accolades :

```
{
    instruction
    instruction
    ...
}
```

Exemple :

```
if (a==10)
{
    Console.WriteLine("a vaut 10");
    a=9;
    Console.WriteLine("désormais a vaut 9");
}
```

Sinon...

Il est possible d'exécuter des instructions quand une condition est vraie, et d'autres instructions quand elle est fausse.

La syntaxe est la suivante :

```
if (expression)
    instruction
else
    instruction
```

Exemples :

```
if (a==10) Console.WriteLine("a vaut 10");
else Console.WriteLine("a ne vaut pas 10");
```

```
if ((a==10)&&(b==11)) Console.WriteLine("a vaut 10 et b vaut 11");
else Console.WriteLine("a ne vaut pas 10 ou b ne vaut pas 11");
```

Conditions multiples

L'enchaînement des instructions conditionnelles est possible.

Exemple :

```
if (a==10)
    if (b==11) Console.WriteLine("a vaut 10 et b vaut 11");
    else Console.WriteLine("a vaut 10 mais b ne vaut pas 11");
else Console.WriteLine("a ne vaut pas 10");
```

Chaque instruction `else` correspond au `if` qui le précède, s'il n'a pas déjà de `else`. Cependant, pour clarifier le code ou résoudre certains cas ambigus (pas de `else` pour le second `if`, par exemple), il est préférable de mettre des accolades :

```
if (a==10)
{
    if (b==11) Console.WriteLine("a vaut 10 et b vaut 11");
    else Console.WriteLine("a vaut 10 mais b ne vaut pas 11");
}
else Console.WriteLine("a ne vaut pas 10");
```

Autre exemple :

```
if (a==10) Console.WriteLine("a vaut 10");
else if (a==11) Console.WriteLine("a vaut 11");
else Console.WriteLine("a ne vaut ni 10, ni 11");
```

Dans cet exemple, chaque instruction `if` n'est testée que si la précédente est fausse. Si le nombre de cas est important et que chaque condition teste une valeur pour la même expression, il est préférable d'utiliser l'instruction `switch`.

Tests de plusieurs cas

L'instruction `switch` permet de tester la valeur d'une expression avec plusieurs cas.

La syntaxe est la suivante :

```
switch(expression)
{
cas :
cas :
...
    instructions
    fin du cas

cas :
cas :
...
    instructions
    fin du cas

...
}
```

Où *cas* peut avoir l'une des deux syntaxes suivantes :

`case` *`constante`*
: Cas où l'expression vaut la constante spécifiée.

`default`
: Cas où l'expression ne correspond à aucun autre cas (le cas par défaut).

fin du cas est une instruction spéciale terminant le ou les cas. Hormis les instructions interrompant le cours de l'exécution (une instruction interrompant ou continuant une boucle, une instruction `throw` pour lancer une exception, une instruction `return`), il s'agit en général de l'une des deux instructions suivantes :

`break;`
: Fin du test, la prochaine instruction exécutée est celle située après l'accolade fermante de `switch`.

`goto *cas*;`
: L'exécution se poursuit au cas indiqué.

Exemple :

```
switch(a)
{
case 10 :
    Console.WriteLine("a vaut 10");
    break;

case 11 :
    Console.WriteLine("a vaut 11");
    break;

default :
    Console.WriteLine("a ne vaut ni 10, ni 11");
    break;
}
```

La syntaxe permet de rassembler plusieurs cas ensemble pour exécuter les mêmes instructions :

```
switch(a)
{
case 2 :
case 3 :
case 5 :
    Console.WriteLine("a vaut 2, 3 ou 5");
    break;

case 10 :
case 11 :
    Console.WriteLine("a vaut 10 ou 11");
    break;

default :
    Console.WriteLine("a n'est pas dans la liste (2, 3, 5, 10, 11)");
    break;
}
```

Il est également possible de poursuivre le traitement avec les instructions d'un autre cas :

```
switch(a)
{
case 10 :
    Console.WriteLine("a vaut 10");
    goto case 11;
```

```
case 11 :
    Console.WriteLine("a vaut "+a);
    break;

default :
    Console.WriteLine("a ne vaut ni 10, ni 11");
    break;
}
```

Boucles

Une boucle permet d'exécuter plusieurs fois une ou des instructions tant qu'une condition est vraie.

Boucle while

La boucle `while` est la plus simple : tant que la condition est vraie, elle exécute l'instruction ou le bloc d'instructions spécifié.

Sa syntaxe est la suivante :

```
while ( expression )
    instruction_ou_bloc
```

Où *expression* est une expression du type `bool`, c'est-à-dire valant `true` (vrai) ou `false` (faux).

Exemple : rechercher une valeur spécifique dans un tableau d'entiers

```
int i = 0;

// tant que i est un indice correct et que la valeur 0 n'est pas trouvée
while ( (i < tableau.Length ) && ( tableau[i] != 0 ) )
{
    Console.WriteLine(tableau[i]);
    i++;
}
```

N.B.: Si la condition est fausse dès le début, l'instruction ou le bloc n'est pas exécuté, car la condition est testée avant.

Boucle do...while

La boucle `do...while` ressemble à la précédente, excepté que la condition est testée après. C'est-à-dire que le bloc d'instructions est toujours exécuté au moins une fois.

La syntaxe est la suivante :

```
do
{
    instructions
}
while ( expression );
```

Où *expression* est une expression de type `bool`.

Exemple :

```
string fichier;
do
{
    Console.Write("Entrez un nom de fichier ou rien pour quitter : ");
    fichier=Console.ReadLine();
    if (fichier != "") TraiterFichier(fichier);
}
while ( fichier != "" );
```

Boucle for

La boucle `for` regroupe plusieurs phases de la boucle :

- L'initialisation, par exemple un indice dans un tableau,
- La condition, tant qu'elle est vraie la boucle continue,
- Les instructions à exécuter,
- L'instruction (ou les instructions) d'incrémentation, exécutée(s) juste avant de tester à nouveau la condition.

La syntaxe d'une boucle `for` est la suivante :

```
for ( initialisation ; condition ; incrémentation )
    instructions
```

Exemple : recherche d'une valeur nulle dans un tableau d'entiers

```
for ( int i=0 ; (i<tableau.Length)&&(tableau[i]!=0) ; i++ )
    Console.WriteLine( tableau[i] );
```

Boucle foreach

La boucle `foreach` parcours tous les éléments d'un objet implémentant l'interface `IEnumerable`. Il s'agit d'une version simplifiée de la boucle `for`.

La syntaxe est la suivante :

```
foreach ( type variable in objet_IEnumerable)
    instructions
```

Durant la boucle, la variable *variable* (de type *type*) vaut successivement toutes les valeurs retournées par l'objet implémentant l'interface `IEnumerable`.

Les tableaux implémentent cette interface, et peuvent donc être utilisés dans une boucle `foreach`.

Exemple : afficher tous les éléments d'un tableau

```
string[] messages = { "Test", "d'une", "boucle foreach" };
foreach ( string s in messages )
```

```
    Console.WriteLine(s);
```

Interrompre une boucle

L'instruction `break` permet d'interrompre prématurément une boucle.

Syntaxe :

```
break;
```

Exemple :

```
for(int i = 0 ; i < 10 ; i++)
{
    Console.WriteLine( "i vaut " + i );
    if (i==5) break; // Arrêter à 5
}
```

Le code ci-dessus affiche :

```
i vaut 0
i vaut 1
i vaut 2
i vaut 3
i vaut 4
i vaut 5
```

Continuer une boucle

L'instruction `continue` permet de poursuivre une boucle, c'est-à-dire passer immédiatement à l'itération suivante sans exécuter les autres instructions de la boucle.

Syntaxe :

```
continue;
```

Exemple :

```
for(int i = 0 ; i < 10 ; i++)
{
    if (i==5) continue; // Sauter 5
    Console.WriteLine( "i vaut " + i );
}
```

Le code ci-dessus affiche :

```
i vaut 0
i vaut 1
```

```
i vaut 2
i vaut 3
i vaut 4
i vaut 6
i vaut 7
i vaut 8
i vaut 9
```

L'interface `IEnumerable` et le mot clé `yield`

L'interface IEnumerable est définie dans deux espaces de nom différents :

- `System.Collections` : (Framework 1.0) utilise le type `object`,
- `System.Collections.Generic` : (Framework 2.0) utilise un type générique.

Comme la plupart des classes de ces deux espaces de nom, ces deux classes ne diffèrent que par le type utilisé pour chaque élément : `object` ou générique. L'interface IEnumerable ne définit qu'une seule méthode :

- `System.Collections.IEnumerable`:

```
IEnumerator GetEnumerator();
```

- `System.Collections.Generic.IEnumerable`:

```
IEnumerator<T> GetEnumerator();
```

L'implémentation d'une fonction retournant une instance de l'interface `IEnumerator` est simplifiée en utilisant le mot clé `yield`. Celui-ci est utilisé pour retourner chaque élément de l'itération ou pour interrompre l'itération :

- L'instruction `yield return` est suivie de la valeur de l'élément à retourner,
- L'instruction `yield break` interrompt la boucle prématurément.

Exemple :

```
using System.Collections.Generic;

public class Mots
{
    // Tableau contenant des mots :
    public string[] mots;

    // Constructeur
    public Mots(params string[] mots)
    { this.mots = mots; }

    // Énumérer tous les mots jusqu'à trouver
    // le mot contenu dans le paramètre dernier (exclu)
    // ou la fin du tableau.
    public IEnumerable<string> TousLesMotsJusquA(string dernier)
    {
        foreach(string mot in mots)
            if (mot.Equals(dernier)) yield break; // fin prématurée de l'itération
            else yield return mot; // élément de l'itération
```

```
        // fin normale de la boucle d'itération
    }
}

public class TestProgram
{
    public static void Main()
    {
        Mots fruits = new Mots( "pomme", "poire", "abricot", "fraise", "kiwi" );
        foreach( string fruit in fruits.TousLesMotsJusquA("fraise") )
            Console.WriteLine( fruit );
    }
}
```

Ce programme affiche :

```
pomme
poire
abricot
```

Les fruits suivants ne sont pas affichés car le mot "fraise" a été trouvé.

Héritage de classes

Une classe peut hériter d'une autre, c'est à dire posséder les mêmes méthodes et attributs que celle-ci et en avoir des supplémentaires. Cette nouvelle classe est appelée « classe fille » ou « sous-classe ». Elle hérite d'une « classe mère » ou « classe de base » ou « super-classe ».

L'héritage entre deux classes traduit la relation « est un type de ». Par exemple, une automobile est un type de véhicule. Elle possède la même fonctionnalité de transport que n'importe quel véhicule. En même temps, une automobile a des fonctionnalités plus précises qu'un véhicule : transport de personnes principalement, capacité de transport limitée, nombre de portes, puissance du moteur, ...

Ces spécificités de la sous-classe sont les seules qui ont besoin d'être définies dans celle-ci. Les autres fonctionnalités sont héritées de la super-classe.

Classe de base et sous-classe

La classe de base (ou classe mère) est celle qui définit les membres qui seront hérités par les sous-classes. Une sous-classe (ou classe fille) hérite des membres de la classe mère, peut ajouter de nouveaux membres, implémenter de nouvelles interfaces et redéfinir le comportement des méthodes de la classe de base.

Lorsqu'une classe ne déclare aucune classe de base, par défaut elle hérite de la classe `System.Object`. La classe `System.Object` est la seule qui n'hérite d'aucune autre classe.

Syntaxe

La relation d'héritage est définie à la déclaration de la classe par le symbole deux-points suivi du nom de la super-classe.

Exemple de super-classe :

```
public class Vehicule
{
    int roues, places, kilometrage;
    public Vehicule()
    {
    }
}
```

Exemple de sous-classe :

```
public class Automobile : Vehicule
{
    string couleur;
    public Automobile()
    {
    }
}
```

Constructeurs

Le constructeur de la sous-classe appelle toujours celui de la classe de base, implicitement ou explicitement. Si rien n'est spécifié, le compilateur génère un appel implicite au constructeur de la classe de base ne comportant aucun paramètre. C'est pourquoi ce constructeur est nommé « constructeur par défaut ».

Si la classe de base ne possède aucun constructeur par défaut, ou si un autre serait plus approprié, il est possible de spécifier explicitement le constructeur à appeler. Le mot-clé `base` désigne la classe de base.

Exemple :

```
public class Vehicule
{
    int roues, places, kilometrage;
    // Constructeur "protégé" : l'accès est limité
    // à cette classe et aux sous-classes
    protected Vehicule(int roues,int places)
    {
        this.roues = roues;
        this.places = places;
    }
}

public class Automobile : Vehicule
{
    string couleur;
    public Automobile(string couleur)
        : base( 4, 5 ) // appel au constructeur de Vehicule
                       // 4 roues et 5 places
    {
        this.couleur = couleur;
    }

    // Ajouter un constructeur par défaut n'est pas obligatoire
    // mais permet d'illustrer qu'il est possible d'appeler un
    // autre constructeur de la même classe
    public Automobile()
        : this( "indéfinie" ) // couleur indéfinie par défaut
    {
    }
```

```
}
```

Référence d'instance

Une référence à une instance de la sous-classe peut être stockée par une référence du même type :

```
Automobile voiture = new Automobile();
```

Il est également possible d'utiliser une référence du type de la classe de base :

```
Vehicule voiture = new Automobile();
```

Redéfinition de méthodes et de propriétés

Dans cette section, la redéfinition de propriétés est similaire à la redéfinition de méthodes. Ce paragraphe discutera de la redéfinition de méthodes, mais tout ceci s'applique également aux propriétés.

Redéfinition sans polymorphisme

La redéfinition d'une méthode consiste à créer une méthode dans une sous-classe ayant le même nom et les mêmes types d'arguments (la même signature) qu'une méthode de la classe de base, afin de modifier son comportement.

Exemple :

```
public class Vehicule
{
    private int poids;

    public Vehicule(int poids)
    { this.poids = poids; }

    public string Description()
    {
        return "Véhicule de "+poids+" tonnes";
    }
}
```

```
public class Automobile : Vehicule
{
    private string couleur;

    public Automobile(int poids,string couleur)
        : base(poids)
    { this.couleur = couleur; }

    // méthode surchargée
    public string Description()
    {
        return base.Description()+" de couleur "+couleur;
    }
}
```

Cependant, le compilateur génère un avertissement (warning CS0108) dans la classe dérivée. Il faut spécifier que l'on redéfinit une méthode, en utilisant le mot clé `new` :

```
public class Automobile : Vehicule
{
    private string couleur;

    public Automobile(int poids,string couleur)
        : base(poids)
    { this.couleur = couleur; }

    // méthode surchargée
    public new string Description()
    {
        return base.Description()+" de couleur "+couleur;
    }
}
```

Utilisation :

```
Automobile voiture = new Automobile(3, "rouge");
Console.WriteLine( voiture.Description() );
// affiche : Véhicule de 3 tonnes de couleur rouge
```

Par contre, si on utilise une référence à la classe de base Vehicule, la méthode appelée sera celle de la classe de base :

```
Vehicule vehicule = new Automobile(3, "rouge");
Console.WriteLine( vehicule.Description() );
// affiche : Véhicule de 3 tonnes
```

Une conversion de la référence vers la classe réelle de l'objet permet d'appeler la méthode surchargée :

```
Console.WriteLine( ((Automobile)vehicule).Description() );
// affiche : Véhicule de 3 tonnes de couleur rouge
```

Cet exemple n'utilise donc pas le polymorphisme, c'est à dire que pour appeler la méthode `Description`, le compilateur se base sur le type de la référence plutôt que sur le type réel de l'objet référencé.

Redéfinition avec polymorphisme

Le polymorphisme permet d'utiliser le type réel de l'objet référencé plutôt que le type de la référence pour déterminer la méthode, la propriété ou l'indexeur à utiliser. Pour cela, il faut utiliser le mot clé `virtual` dans la déclaration de la méthode de la classe de base.

Exemple :

```
public class Vehicule
{
    private int poids;

    public Vehicule(int poids)
    { this.poids = poids; }
```

```
    public virtual string Description()
    {
        return "Véhicule de "+poids+" tonnes";
    }
}
```

Et il faut utiliser le mot clé `override` dans la classe dérivée :

```
public class Automobile : Vehicule
{
    private string couleur;

    public Automobile(int poids,string couleur)
        : base(poids)
    { this.couleur = couleur; }

    // méthode surchargée
    public override string Description()
    {
        return base.Description()+" de couleur "+couleur;
    }
}
```

Utilisation :

```
Vehicule vehicule = new Automobile(3, "rouge");
Console.WriteLine( vehicule.Description() );
// affiche : Véhicule de 3 tonnes de couleur rouge
```

Si le mot clé `new` est utilisé à la place du mot clé `override`, le polymorphisme n'est pas effectif, comme dans le paragraphe précédent.

Résumé

Pour résumer :

		Méthode de la classe de base	
		normale	**`virtual`**
Méthode surchargée dans la Classe dérivée		Sans polymorphisme, avertissement CS0108	Sans polymorphisme, avertissement CS0114
	`new`	Sans polymorphisme	Sans polymorphisme
	`override`	erreur CS0506	Avec polymorphisme

Classe sans héritière

Le mot-clé sealed empêche la création de classes dérivées. Cette fonctionnalité est également utile pour les classes ne déclarant que des membres statiques.

Syntaxe:

```
sealed class nom_classe
```

Exemple:

```
public sealed class Ferrari : Automobile
{
    private int consommation;
    public Ferrari(int litre_par_km)
        : base(5100, "Rouge")
    { consommation = litre_par_km; }
}
```

Ce type de classe ne peut avoir de méthodes abstraites (`abstract`) ou de membres protégés (`protected`) car aucune classe dérivée ne pourra être créée pour les implémenter / y accéder.

Classe abstraite

Une classe abstraite possède au moins une méthode abstraite ou une propriété abstraite, c'est à dire ne comportant aucune implémentation (pas de code). Une telle classe ne peut être instanciée avec l'opérateur `new`. Il faut utiliser une sous-classe implémentant les méthodes abstraites de la classe de base.

Syntaxe

Le mot clé `abstract` doit précéder la classe abstraite et toutes les méthodes abstraites.

Exemple :

```
// Classe abstraite gérant un ensemble d'objets
public abstract class Ensemble
{
    public bool AjouterSiNeContientPas(object o)
    {
        if ( ! Contient(o) )
            Ajouter(o);
    }

    public bool RetirerSiContient(object o)
    {
        if ( Contient(o) )
            Retirer(o);
    }

    // méthodes abstraites (sans implémentation) :
    public abstract bool Contient(object o);
    protected abstract void Ajouter(object o);
    protected abstract void Retirer(object o);
}
```

Implémenter les méthodes abstraites

La classe implémentant les méthodes abstraites de la classe de base doit utiliser le mot clé `override`, car les

méthodes abstraites sont implicitement virtuelles également :

```
public class EnsembleTableau : Ensemble
{
    private object[] elements = new object[0];

    public override bool Contient(object o)
    {
        // recherche de l'objet
        for(int i = 0  ; i < elements.Length ; i++)
            if (elements[i] == o) return true; // objet trouvé

        // fin de la boucle sans avoir trouvé l'objet
        return false;
    }

    protected override void Ajouter(object o)
    {
        // ...
    }

    protected override void Retirer(object o)
    {
        // ...
    }
}
```

La classe peut ne pas implémenter toutes les méthodes abstraites de la classe de base. Dans ce cas elle est également abstraite, et laisse le soin à ses sous-classes d'implémenter les méthodes abstraites restantes.

Propriétés abstraites

Il est également possible de déclarer des propriétés abstraites.

Exemple :

```
public abstract class Ensemble
{
    public abstract int nombre { get; }
    public abstract string nom { get; set; }
}

public class EnsembleTableau : Ensemble
{
    private string nom_ensemble = null;
    private object[] elements = new object[0];

    public override int nombre
    {
        get { return elements.Count; }
    }

    public override string nom
    {
        get { return nom_ensemble ; }
        set { nom_ensemble = value ; }
    }
}
```

Mode d'accès

Puisque les méthodes et propriétés abstraites doivent être implémentées par les sous-classes, il n'est pas possible de les déclarer avec le mot clé `private`, ou d'utiliser le mot clé `sealed` avec la classe. De plus les méthodes et propriétés abstraites ne peuvent être statiques.

Les exceptions

Une exception est créée et lancée quand une erreur survient. Elle se propage dans la pile d'appel de la manière suivante : à partir du moment où elle est lancée, l'exécution normale est interrompue, et un gestionnaire d'exceptions est recherché dans le bloc d'instruction courant. S'il n'est pas trouvé, la recherche se poursuit dans le bloc englobant celui-ci, ou à défaut, dans le bloc de la fonction appelante, et ainsi de suite... Si la recherche n'aboutit pas, une boîte de dialogue signalant l'exception est affichée.

Attraper une exception

Un gestionnaire d'exception attrape une classe d'exception particulière et gère le cas d'erreur correspondant. Ce gestionnaire encadre les instructions à gérer pouvant lancer une exception.

La syntaxe est la suivante :

```
try
{
    // Une exception peut être lancée
    instructions
}
catch ( classe_d_exception variable )
{
    // Gérer l'erreur en fonction des détails
    // de l'erreur contenus dans la variable
    instructions
}
...autres blocs catch...
finally
{
    // Instructions toujours exécutées
    // exception lancée ou non
    instructions
}
```

Le bloc `try` est suivi d'un nombre quelconque de bloc `catch` (éventuellement aucun) attrapant différents types d'exception, et éventuellement d'un bloc `finally` qui sera toujours exécuté quoi qu'il se passe.

Exemple :

```
try
{
    Console.Write("Entrez un nombre : ");
    int n = int.Parse( Console.ReadLine() );
    Console.WriteLine("  100/nombre = "+( 100/n ));
}
catch ( DivideByZeroException dbzex )
```

```
{
    Console.Error.WriteLine("  Division par zéro");
}
catch ( Exception ex )
{
    Console.Error.WriteLine(
        "  Une autre exception a eu lieu : "+ex.Message);
}
finally
{
    Console.WriteLine(
        "  Quel que soit le résultat, ceci est affiché");
}
```

Libérer des ressources

Un bloc `finally` est utile pour libérer des ressources à la fin d'un traitement, qu'une erreur ait eu lieu ou non.

Exemple :

```
Bitmap bm;
try
{
    bm=new Bitmap(100,100);
    ...
}
finally
{
    bm.Dispose(); // libérer les ressources
}
```

Cependant, les classes implémentant l'interface `IDisposable` ont une méthode `Dispose()`, et peuvent être utilisées avec le mot clé `using` :

```
using( Bitmap bm = new Bitmap(100,100) )  // <- objet IDisposable
{
    ...
} // <- méthode Dispose() appelée automatiquement
```

Lancer une exception

En cas d'erreur dans une méthode d'un programme (arguments invalides, ...), il est possible de lancer une exception en utilisant le mot clé `throw`.

La syntaxe est la suivante :

```
throw objet_exception;
```

Où *objet_exception* est une instance de la classe `Exception` ou de l'une de ses sous-classes.

En général, l'objet exception est alloué en même temps qu'il est lancé :

```
throw new classe_exception(arguments);
```

La pile d'appel est enregistrée dans l'objet exception au moment où il est lancé.

Dans un gestionnaire d'exceptions, il est possible de relancer l'exception attrapée en utilisant l'instruction `throw` sans argument. Ce qui est utile quand le gestionnaire ne gère que partiellement l'erreur qui devra être totalement traitée par un autre gestionnaire d'exceptions. Dans ce cas, la pile d'appel d'origine est conservée.

Exemple :

```
try
{
    Console.Write("Entrez un nombre : ");
    int n = int.Parse( Console.ReadLine() );
    Console.WriteLine("  100/nombre = "+( 100/nombre ));
}
catch ( DivideByZeroException dbzex )
{
    Console.Error.WriteLine("  Division par zéro");
    throw; // relance la même exception, avec la pile d'appel d'origine
    //throw dbzex; // relance avec une nouvelle pile d'appel
}
```

Créer une classe d'exception

Lancer une exception signale une erreur particulière. Si aucune classe d'exception ne convient ou n'est suffisamment précise, ou si l'exception doit comporter des informations supplémentaires, il est possible de créer une nouvelle classe d'exception.

Pour cela, il faut dériver la classe `Exception` ou l'une de ses sous-classes. Par convention, toutes ces sous-classes ont un nom se terminant par `Exception`.

Exemple :

```
public class ErreurDeScriptException : Exception
{
    // Attributs
    private int ligne,colonne;
    private string fichier;

    // Propriétés (en lecture seule)
    public int Ligne { get { return ligne; } }
    public int Colonne { get { return colonne; } }
    public string Fichier { get { return fichier; } }

    // Constructeur
    public ErreurDeScriptException(
        string message,
        string fichier,
        int ligne,
        int colonne )
        : base(message) // appel au constructeur de la classe Exception
    {
        this.fichier = fichier;
        this.ligne = ligne;
        this.colonne = colonne;
```

```
    }
}
```

Ensuite, cette classe peut être utilisée comme n'importe quelle autre classe d'exception :

```
if ( arg==null )
    throw new ErreurDeScriptException(
        "Un argument est nécessaire", fichier_script, 10 , 5);
```

Structures et énumérations

Structure

Une structure rassemble plusieurs champs (des variables) en un seul type. Déclarer une variable de ce type revient à allouer de la place pour tous les champs déclarés.

Syntaxe

```
struct nom_de_structure
{
    type variable;
}
```

Exemple :

```
struct Point
{
    public double x;
    public double y;
}
```

Méthodes

Comme pour une classe, il est possible d'ajouter des méthodes.

Exemple :

```
struct Point
{
    public double x;
    public double y;
    public void MoveTo(double x,double y)
    { this.x=x; this.y=y; }
}
```

Différences avec les classes

Il est possible d'assimiler les structures à des classes, cependant des différences existent :

- Une structure ne peut hériter d'une autre, ou d'une classe ;
- Les membres sont publics par défaut ;
- Une instance de structure n'est pas une référence mais l'espace occupé par ses champs, par conséquent, l'opérateur `new` n'est pas utilisable, et une structure ne peut valoir `null` car allouée à la déclaration.

Exemple pour illustrer ce dernier point :

```
Point origine;   // alloué en mémoire à la déclaration
origine.x = 0.0;
origine.y = 0.0;
```

Passage d'une structure en paramètre

Le passage d'une structure en paramètre d'une fonction peut se faire de deux manières :

- Par valeur, dans ce cas tous les champs de la structure sont passés dans la pile, ce qui peut prendre beaucoup de temps et de mémoire, voire causer un débordement de pile ;
- Par référence (`ref` ou `out`), manière recommandée avec les structures car seule une addresse est passée.

Énumération

Une énumération est un type de données dont les valeurs sont des constantes nommées.

Syntaxe

```
enum nom_énumération
{
    nom, nom ...
};
```

Exemple :

```
enum JourDeSemaine
{ LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE };
```

Type des constantes

Les constantes d'une énumération sont par défaut des entiers (`int`) dont la valeur augmente de un, en commançant par zéro.

Ce qui veut dire que dans l'exemple précédent, `LUNDI` vaut 0, `MARDI` vaut 1, etc. Une constante peut être convertie en entier :

```
Console.WriteLine("Lundi : "+JourDeSemaine.LUNDI);
// affiche    Lundi : LUNDI

Console.WriteLine("Lundi : "+(int)JourDeSemaine.LUNDI);
// affiche    Lundi : 0
```

Il est possible de modifier les valeurs affectées aux constantes :

```
enum JourDeSemaine
{ LUNDI=1, MARDI=2, MERCREDI=3, JEUDI=4, VENDREDI=5, SAMEDI=6, DIMANCHE=7 };
```

Par défaut, chaque constante est associée à la valeur immédiatement supérieur à celle de la constante précédente. L'exemple précédent peut donc également s'écrire :

```
enum JourDeSemaine
{ LUNDI=1, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE };
```

Il est également possible de modifier le type des constantes :

```
enum JourDeSemaine : long
{ LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE };
```

Combinaison de constantes

En affectant des valeurs différentes de puissances de 2 aux constantes, l'opérateur ou (`|`) est utilisable pour combiner plusieurs constantes.

Exemple :

```
[Flags]
enum Droits
{
    LECTURE = 1,
    ECRITURE = 2,
    EFFACER = 4
};
Droits d = Droits.LECTURE | Droits.EFFACER;

Console.WriteLine("droits : "+d);
// affiche    droits : LECTURE, EFFACER

Console.WriteLine("droits : "+(int)d);
// affiche    droits : 5
```

L'attribut `Flags` indique au compilateur que plusieurs constantes peuvent être combinées. La méthode `ToString` de cet `enum` affiche alors toutes les constantes utilisées.

L'opérateur et (`&`) permet de tester si une constante appartient à l'ensemble :

```
if ( (d & Droits.LECTURE) != 0) Console.WriteLine("  lecture autorisée");
if ( (d & Droits.ECRITURE) != 0) Console.WriteLine("  écriture autorisée");
```

Interfaces

Une interface ne fait que décrire une liste de méthodes, sans implémentation. Le code de ces méthodes est fourni par les classes qui implémentent l'interface.

Déclarer une interface

Le mot clé `interface` sert à déclarer une interface. Il est suivi par le nom de l'interface, qui par convention commence généralement par la lettre `I` (i majuscule comme interface).

Exemple :

```
public interface IAffichable
{
    /*
        Liste des méthodes que doivent posséder toutes les classes
        implémentant l'interface IAffichable :
    */
    void Afficher();
    void Afficher(string message);
}
```

Par convention, le nom d'une interface ne comportant qu'une seule méthode est celui de la méthode suivi du suffixe "able". Par exemple, si la seule méthode s'appelle `Draw`, l'interface est nommée `IDrawable`.

Utiliser une interface

L'interface créée est un nouveau type dont les méthodes sont appelables sans connaître la classe qui les implémente.

Exemple :

```
public void Montrer(IAffichable affichable,string message)
{
    affichable.Afficher(message);
}
```

Implémenter une interface

Une interface est implémentée par une classe la déclarant dans sa liste d'implémentation. Cette classe doit alors fournir le code de toutes les méthodes de l'interface, à moins de déclarer ces méthodes comme abstraites. Dans ce dernier cas, l'implémentation devra être effectuée par une sous-classe.

Exemple :

```
using System;

public class Personne : IAffichable
{
    private string nom, prenom;

    public void Afficher()
    {
        Console.WriteLine(nom+" "+prenom);
    }
```

```
    public void Afficher(string message)
    {
        Console.WriteLine(nom+" "+prenom+" : "+message);
    }
}
```

Méthodes, propriétés, indexeurs, events

Une interface peut en fait déclarer des méthodes, des propriétés, des indexeurs et des events.

Note : Lorsque vous utilisez un accesseur pour implémenter une interface, l'accesseur peut ne pas avoir de modificateur d'accès. [...] [[1] (http://msdn.microsoft.com/fr-fr/library/75e8y5dd%28VS.80%29.aspx) MSDN]

Exemple :

```
public interface IExemple
{
    // Méthode à implémenter :
    void UneMethodeAImplementer();

    // Propriétés à implémenter :
    string UneProprieteAImplementer { get; set; }
    string UneProprieteLectureSeuleAImplementer { get; }

    // Tableau de string à implémenter (indexeur) :
    string this [ int index ] { get; set; }

    // Evènement à implémenter :
    event PageRecueDelegate PageRecue;
}
```

Héritage

Une interface peut hériter d'une autre interface. Elle possède donc les mêmes déclarations que l'interface de base en plus de ces propres déclarations. En outre, elle peut être utilisée là ou une implémentation de l'interface de base est requise.

Exemple :

```
public interface IDrawable
{
    void Draw();
}

public interface IPrintable : IDrawable
{
    void Print();
}

public class Cercle : IPrintable
{
    public void Draw()
    {
        ...
    }
```

```
    public void Print()
    {
        ...
    }
}

...
public void Methode()
{
    IDrawable drawable = new Cercle();
    // ^ conversion implicite de IPrintable vers IDrawable
    drawable.Draw();
}
```

Type partiel

Le langage C# permet d'implémenter les classes, les structures et les interfaces sur plusieurs fichiers. Visual Studio .Net utilise cette fonctionnalité pour les fenêtres graphiques : une partie de la classe `Form1` est implémentée dans le fichier `Form1.cs` et l'autre dans le fichier `Form1.Designer.cs`.

Syntaxe

Chaque fichier implémentant une partie d'un type doit faire précéder le mot clé du type (`class`, `struct` ou `interface`) du mot clé `partial` :

```
mode_accès partial type nom ...
{
    ...
}
```

Exemple :

```
public partial class Exemple
{
}
```

Il n'est pas nécessaire que tous les fichiers précisent le mode d'accès ou la classe de base.

Compilation

À la compilation d'un type partiel, tous les fichiers sources concernés doivent être spécifiés sur la ligne de commande d'appel au compilateur.

Surcharge des opérateurs

La surcharge des opérateurs permet d'utiliser les opérateurs sur d'autres types (des classes) que les types

simples.

Certains opérateurs ne sont pas surchargeables. Il s'agit des opérateurs en rouge dans le tableau du chapitre *Les opérateurs*.

Syntaxe

L'opérateur surdéfini doit toujours être statique et public, sinon le compilateur génère une erreur. L'opérateur est déclaré en utilisant le mot clé `operator` suivi de l'opérateur surdéfini.

Exemple :

```
public class NombreComplexe
{
    private double n_reel, n_imag;

    public NombreComplexe() {}

    public NombreComplexe(double r,double i)
    { this.n_reel = r; this.n_imag = i; }

    public static NombreComplexe operator +
        (NombreComplexe a,NombreComplexe b)
    {
        return new NombreComplexe
          ( a.n_reel + b.n_reel , a.n_imag + b.n_imag );
    }
}
```

Implémentation par paire

La surdéfinition de certains opérateurs exige également la surdéfinition d'un autre. Ces opérateurs doivent donc être surdéfinis par paire.

Il s'agit des paires d'opérateurs suivantes :

- `operator <` et `operator >`
- `operator <=` et `operator >=`
- `operator ==` et `operator !=`

En général, l'un des deux peut être défini en fonction de l'autre :

- `operator <` en fonction de `operator >`

```
    public static bool operator < (NombreComplexe a,NombreComplexe b)
    {
        return b > a;
    }
```

- `operator >` en fonction de `operator <`

```
    public static bool operator > (NombreComplexe a,NombreComplexe b)
    {
        return b < a;
```

```
    }
```

- `operator <=` en fonction de `operator >=`

```
    public static bool operator <= (NombreComplexe a,NombreComplexe b)
    {
        return b >= a;
    }
```

- `operator >=` en fonction de `operator <=`

```
    public static bool operator >= (NombreComplexe a,NombreComplexe b)
    {
        return b <= a;
    }
```

- `operator ==` en fonction de `operator !=`

```
    public static bool operator == (NombreComplexe a,NombreComplexe b)
    {
        return !( a != b );
    }
```

- `operator !=` en fonction de `operator ==`

```
    public static bool operator != (NombreComplexe a,NombreComplexe b)
    {
        return !( a == b );
    }
```

Opérateurs de conversions

Les opérateurs de conversions sont déclarés dans une classe C en ajoutant une méthode utilisant la syntaxe suivante :

```
operator type_cible(type_source valeur)
```

où l'un des deux types est la classe C : convertir un objet de classe C vers *type_cible* ou un objet de type *type_source* vers classe C.

Exemple:

```
public class Fraction
{
    private int numerateur, denominateur;

    public Fraction(int n, int d)
    { this.numerateur = n; this.denominateur = d; }

    public double GetValue()
```

```
    { return this.numerateur / this.denominateur; }

    public static implicit operator double(Fraction f)
    {
        return f.GetValue();
    }

    public static implicit operator Fraction(int entier)
    {
        return new Fraction(entier, 1);
    }
}
...
Fraction f=new Fraction(1,3);
double d = f; // -> 0.33333...
...
Fraction f;
f = 5; // -> 5 / 1
```

Explicite/Implicite

L'opérateur de conversion doit être déclaré avec l'un des mots-clés `implicit` ou `explicit` pour qu'il soit utilisé respectivement implicitement ou explicitement.

Exemple:

```
public class Fraction
{
    private int numerateur, denominateur;

    public Fraction(int n, int d)
    { this.numerateur = n; this.denominateur = d; }

    public double GetValue()
    { return this.numerateur / this.denominateur; }

    public static explicit operator Fraction(int entier)
    {
        return new Fraction(entier, 1);
    }
}
...
Fraction f;
f = (Fraction)5; // conversion explicite -> 5 / 1
```

Directives du préprocesseur

Le préprocesseur est le traitement effectué par le compilateur pour modifier le fichier source (en mémoire seulement) avant de le compiler. Il permet une compilation conditionnelle (fonctions de deboggage à exclure quand l'application est finalisée, fonctionnalités selon des options, ...), il modifie donc la façon de compiler les fichiers sources.

Syntaxe générale des directives

Une directive est toujours placée seule sur une ligne de texte et commence par le caractère dièse (`#`). Le premier mot identifie la directive. Il est éventuellement suivi de paramètres.

Exemple :

```
#pragma warning disable
```

La directive `pragma` est suivie des paramètres `warning` et `disable`.

Portée des directives

La portée d'une directive commence à partir de la ligne qui suit cette directive, jusqu'à la fin du fichier source, ou la prochaine contre-directive.

Avertissements du compilateur

Lors de la compilation, des avertissements sont générés pour attirer le programmeur sur certaines instructions du fichier source potentiellement ambiguës ou dangereuses. Si le programmeur, après vérification de l'endroit indiqué, estime que le danger potentiel ne peut arriver, ou qu'il s'agit bien de ce qu'il faut faire, il est possible d'enlever les avertissements dans une zone du fichier source afin que les avertissements inutiles ne soit plus générés.

Ignorer un avertissement

Pour ignorer un avertissement, la directive est la suivante :

```
#pragma warning disable codes
```

Où *`codes`* est la liste des codes des avertissements à ignorer. Ces codes correspondent à ceux affichés par le compilateur, sans le préfixe "`CS`".

Ignorer tous les avertissements

Pour ignorer tous les avertissements, la directive est la même que la précédente, sans aucun code :

```
#pragma warning disable
```

Restaurer un avertissement

Pour restaurer la génération d'un avertissement, c'est à dire ne plus l'ignorer, la directive est la suivante :

```
#pragma warning restore codes
```

Où *`codes`* est la liste des codes des avertissements à restaurer.

Restaurer tous les avertissements

Pour restaurer tous les avertissements, la directive est la même que la précédente, sans aucun code :

```
#pragma warning restore
```

Symboles de compilation

Un symbole est défini avec la directive `#define`. Il possède un nom mais ne peut prendre de valeur en C#, à la différence d'autres langages comme le C ou le C++.

La syntaxe est la suivante :

```
#define nom_du_symbole
```

Le symbole est défini jusqu'à la fin ou jusqu'à la directive `#undef` correspondante. La syntaxe de cette directive est la suivante :

```
#undef nom_du_symbole
```

Exemple :

```
#undef MAX_PAGES
```

Compilation conditionnelle

La syntaxe générale des directives de compilation conditionnelle est la suivante :

```
#if condition
    à compiler si la condition est vraie

#elif condition
    sinon, à compiler si la condition est vraie

...#elif...

#else
    à compiler sinon

#endif
```

Les blocs `#elif` et `#else` sont optionnels.

La condition peut avoir l'une des syntaxes suivantes :

nom_symbole
: Retourne vrai si le symbole spécifié est défini.

!*condition*
: Retourne vrai si la condition est fausse, et *vice-versa*.

condition && condition
: Retourne vrai si les deux conditions sont vraies (ET logique).

condition || condition
: Retourne vrai si au moins l'une des deux conditions est vraie (OU logique).

(*condition*)
: Priorité d'évaluation.

La compilation conditionnelle est souvent utilisée pour les fonctions de déboggage. Par exemple :

```
#if DEBUG
// Fonction de déboggage
public void trace(string message)
{
    Console.WriteLine(message);
}
#endif

public void traitement()
{
#if DEBUG
     trace("Début de la fonction traitement()");
#endif
     ...
#if DEBUG
     trace("Fin de la fonction traitement()");
#endif
}
```

Pour ce genre de condition, l'attribut `Conditionnal` est plus simple à utiliser :

```
// Fonction de déboggage
[Conditionnal("DEBUG")]
public void trace(string message)
{
    Console.WriteLine(message);
}

public void traitement()
{
     trace("Début de la fonction traitement()");
     ...
     trace("Fin de la fonction traitement()");
}
```

Générer une erreur ou un avertissement

Il est possible qu'une certaine combinaison de symboles n'ait pas de sens, ou ne devrait pas être utilisée. Pour prévenir ce risque, en testant la condition correspondante, il est possible de générer une erreur ou un avertissement à la compilation, en utilisant les directives suivantes :

- Pour générer une erreur :

```
#error message
```

- Pour générer un avertissement :

```
#warning message
```

Délimiter une région du code

Délimiter une région sert à regrouper des lignes de codes (méthodes, classes, ...). Les régions de code sont utilisés dans les éditeurs de code spécialisés (Visual Studio, ...) pour permettre leur réduction. Lorsque la région est cachée, trois points de suspension apparaissent à la place des lignes contenues dans la région. Ces lignes apparaissent dans une bulle d'information lorsque le curseur de la souris passe sur ces trois points de suspension.

La syntaxe est la suivante :

```
#region description courte de la région de code
...
#endregion
```

Documentation XML des classes

Le langage C# permet de documenter les classes d'une application en utilisant des commentaires spéciaux juste avant la déclaration de l'élément concerné (espace de noms, classe, interface, méthode, propriété, attribut).

Syntaxe

Les commentaires de documentation commencent par un triple slash `///` et se termine au prochain retour à la ligne.

Le contenu de ces commentaires est au format XML. Il est possible d'utiliser plusieurs lignes.

Exemple :

```
/// <summary>
/// Une classe pour démontrer
/// les commentaires de documentation
/// </summary>
public class Exemple
{
    ...
}
```

Depuis la version 2003 du compilateur C# (pour la normalisation ECMA), il est également possible d'utiliser des blocs de commentaires `/** ... */` :

```
/**
    <summary>
    Une classe pour démontrer
    les commentaires de documentation
    </summary>
```

```
*/
public class Exemple
{
    ...
}
```

Il est possible de mélanger les deux styles, mais non recommandé pour la lisibilité du code :

```
/**
    <summary>
    Une classe pour démontrer
    les commentaires de documentation
*/
/// </summary>
public class Exemple
{
    ...
}
```

Générer la documentation

Le compilateur C# est capable de générer un fichier XML à partir des fichiers sources.

Windows	`csc /doc:fichier.xml fichier_source.cs`
Linux (Mono)	`gmcs -doc:fichier.xml fichier_source.cs`

Exemple :

```
csc /doc:exemple.xml exemple.cs
```

Balises XML standards

Quelques balises XML sont utilisées couramment. Les balises suivantes sont placées en dehors de toute autre :

<summary>
: Description sommaire de l'entité (classe, méthode ou autre).

<remarks>
: Remarque concernant l'entité décrite.

<value>
: Description de la valeur d'une propriété.

<param name="*nom_du_parametre*">
: Description du paramètre de la méthode.

<returns>
: Description de la valeur retournée par la méthode.

<typeparam name="*nom_du_parametre_type*">
: Description du paramètre type générique.

<seealso cref="*membre*">
: Référence à une classe ou un de ses membres en relation avec l'entité décrite.

<exception cref="*classe*">
: Décrit la classe d'exception lancée par la méthode : description de l'erreur, cas où l'exception est lancée, ...

<example>
: Un exemple d'utilisation.

Les balises suivantes sont placées à l'intérieur de celles décrites ci-dessus pour formater le texte :

<para>
: Paragraphe de texte.

<c>
: Extrait de code.

<code>
: Bloc de code.

<paramref name="*nom_du_parametre*"/>
: Référence dans le texte à un paramètre de la méthode.

<typeparamref name="*nom_du_parametre*"/>
: Référence dans le texte à un paramètre type générique.

<see cref="*membre*">
: Référence dans le texte à une classe ou un de ses membres en relation avec l'entité décrite.

<list type="*type_de_liste*">
: Une liste ou un tableau. *type_de_liste* peut valoir `bullet` (liste non ordonnée), `number` (liste numérotée) ou `table` (tableau).

Il est également possible d'ajouter ses propres balises XML.

La balise `<list>` contient les balises suivantes :

<listheader>
: En-tête de tableau.

<item>
: Ligne de tableau.

Ces balises contiennent :

<term>
: Terme défini.

<description>
: Description correspondante.

L'attribut `cref` peut en fait s'appliquer à toute balise XML pour faire référence à une autre entité (classe, méthode, propriété, ...).

Il est également possible d'utiliser un fichier XML séparé pour documenter plusieurs entités, en utilisant la balise `include` :

<include file="*chemin_du_fichier_xml*" path="*chemin_XPath*" />
: Copier les balises XML spécifiées par le chemin XPath dans le fichier spécifié.

Visualiser la documentation

Visualiser directement le fichier XML tel quel n'est pas très pratique. Il est possible d'utiliser des logiciels de

visualisation spécialisés, ou bien d'utiliser un navigateur supportant XML et XSL (Internet Explorer et Firefox par exemple). Pour cela, plusieurs feuilles de style (*stylesheet* en anglais) sont téléchargeables depuis internet :

- http://www.codeproject.com/soap/XMLDocStylesheet.asp

Puis il faut modifier le fichier XML produit en ajoutant la ligne suivante juste après la première balise `<?xml ...?>` :

```
<?xml-stylesheet type="text/xsl" href="''nom_du_fichier''.xsl"?>
```

En double-cliquant sur le fichier XML sous Windows, le navigateur applique la transformation décrite par la *stylesheet* indiquée pour afficher une version HTML plus présentable que le format initial.

Cependant la feuille de transformation fournie par MSDN comporte quelques problèmes :

- Les types génériques ne sont pas correctement présentés et laissés tel qu'ils sont dans le fichier XML initial : un ou deux apostrophes inversées indiquent le nombre ou l'indice des paramètres génériques. Par exemple : `Affiche``2(string, ``1, ``0)` représente `Affiche<T0,T1>(string, <T1>, <T0>)`,
- Les références dans les balises XML `seealso` ne sont pas affichées. Pour corriger ce problème, remplacer `cref` par `@cref` dans la feuille de style :

```
<xsl:template match="seealso">
    <xsl:if test="@cref">
```

En savoir plus

- français Balises recommandées pour les commentaires de documentation (http://msdn2.microsoft.com/fr-fr/library/5ast78ax(VS.80).aspx) [archive]
- français Laboratoire 2 : Commentaires XML (http://www.microsoft.com/france/msdn/articles/vstudio/vsnet_labs2.mspx) [archive]

Attributs

En C#, chaque déclaration de classe, méthode, propriété, ou variable peut être précédée d'un ou plusieurs attributs.

Rôle d'un attribut

Le rôle d'un attribut est d'associer des informations, des méta-données à une déclaration. Ces données peuvent être utilisées par le compilateur, ou le Framework .NET, ou par l'application elle-même.

Syntaxe d'utilisation

Un attribut comporte un nom et éventuellement des paramètres. Il est encadré par des crochets. Par exemple, Visual Studio .NET génère la fonction `Main()` de la manière suivante :

```
...
    [STAThread]
    public static void Main()
    {
        ...
    }
...
```

L'attribut associé à la méthode `Main()` est nommé `STAThread` et ne comporte aucun paramètre.

Nom raccourci ou nom long

Dans l'exemple précédent, l'attribut `STAThread` est en fait géré par une classe nommée `STAThreadAttribute` qui dérive de la classe `System.Attribute`.

L'exemple fonctionne également avec le code source suivant :

```
    [STAThreadAttribute]
    public static void Main()
    {
        ...
    }
```

Par convention, toute classe définissant un attribut doit porter un nom se terminant par `Attribute`. Ce suffixe est optionnel, le compilateur recherchera d'abord le nom indiqué, puis s'il ne trouve pas la classe, il ajoute le suffixe pour effectuer une seconde recherche.

Attribut avec paramètres

Deux exemples d'attribut comportant des paramètres :

Exemple 1:

```
    [Obsolete("Veuillez ne plus utiliser cette méthode.")]
    public void AncienneMethode()
    {
        ...
    }
```

Exemple 2:

```
    [DllImport("User32.dll", EntryPoint="MessageBox")]
    static extern int MessageDialog(int hWnd, string msg, string caption, int msgType);
```

Un attribut possède deux catégories de paramètres :

- Les paramètres ordonnés : ils ne sont pas nommés, et doivent être placés dans le bon ordre, et avant les paramètres nommés ;
- Les paramètres nommés : le nom du paramètre est suivi du signe égal et de la valeur affectée au paramètre.

Plusieurs attributs par déclaration

Plusieurs attributs peuvent être associés à une déclaration. Il est possible de les mettre dans le même bloc de crochets, en les séparant par une virgule :

```
[Obsolete("Veuillez ne plus utiliser cette méthode."),
 DllImport("User32.dll", EntryPoint="MessageBox")]
static extern int MessageDialog(int hWnd, string msg, string caption, int msgType);
```

Ou bien de les mettre dans des blocs différents :

```
[Obsolete("Veuillez ne plus utiliser cette méthode.")]
[DllImport("User32.dll", EntryPoint="MessageBox")]
static extern int MessageDialog(int hWnd, string msg, string caption, int msgType);
```

Cible d'un attribut

Un attribut est associé à la déclaration qui suit (sa cible), mais il existe des cas où la cible est ambiguë. En effet, il existe des attributs globaux associés à l'*assembly* lui-même et ne concerne pas une déclaration particulière. De même, les attributs concernant la valeur de retour d'une méthode sont placés au même endroit que ceux concernant la méthode elle-même.

Pour lever l'ambiguïté, il est possible de préciser la cible des attributs contenus dans le bloc de crochets. Le nom de la cible est suivi du signe deux-points (`:`).

Exemple :

```
[assembly: AssemblyTitle("Essai")]
```

Les cibles possibles sont :

- `assembly` : attributs concernant l'assembly (ne précède aucune déclaration particulière),
- `module` : attributs concernant le module (ne précède aucune déclaration particulière),
- `type` : attributs concernant la classe, la structure, l'interface, l'énumération ou le delegué,
- `method` : attributs concernant la méthode, l'accesseur `get` ou `set` d'une propriété, l'accesseur `add` ou `remove` d'un évènement, ou l'évènement,
- `return` : attributs concernant la valeur retournée par la méthode, le délégue, l'accesseur `get` ou `set` d'une propriété,
- `param` : attributs concernant un paramètre (méthode, délégue, accesseur `set` d'une propriété, accesseur `add` ou `remove` d'un évènement),
- `field` : attributs concernant le champ, ou l'évènement,
- `property` : attributs concernant la propriété ou l'indexeur,
- `event` : attributs concernant l'évènement.

Créer un nouvel attribut

Le langage C# permet de créer de nouveaux attributs pour, par exemple, documenter une partie du code ou associer des données particulières à une déclaration.

Créer la classe

Pour créer un nouvel attribut, il faut créer une classe dérivant de la classe `System.Attribute`, et la nommer

avec le suffixe `Attribute`. Exemple :

```
using System;
public class ExempleAttribute : Attribute
{
}
```

Cet attribut peut déjà être utilisé tel qu'il est, sans paramètres :

```
[Exemple]
public class UneClasse
{
    [method:Exemple]
    [return:Exemple]
    public int UneMethode(int UnParametre)
    {
        return UnParametre;
    }
}
```

Définir les paramètres positionnels

Les paramètres positionnels (ou *ordonnés*) correspondent aux paramètres du constructeur. Exemple :

```
using System;
public class ExempleAttribute : Attribute
{
     private string titre;
     public ExempleAttribute(string titre)
     { this.titre = titre; }
}

[Exemple("Un exemple de classe")]
public class UneClasse
{
    [method:Exemple("Une méthode")]
    [return:Exemple("Retourne le paramètre passé")]
    public int UneMethode(int UnParametre)
    {
        return UnParametre;
    }
}
```

Un attribut peut avoir plusieurs constructeurs différents pour définir différents types de paramètres positionnels. Exemple :

```
using System;
public class ExempleAttribute : Attribute
{
     private string titre,commentaire;
     public ExempleAttribute(string titre)
     {
         this.titre = titre;
         this.commentaire = "Sans commentaire";
     }
     public ExempleAttribute(string titre,string commentaire)
     {
```

```
        this.titre = titre;
        this.commentaire = commentaire;
    }
}

[Exemple("Un exemple de classe",
         "Cette classe est un exemple")]
public class UneClasse
{
    [method:Exemple("Une méthode")]
    [return:Exemple("Retourne le paramètre passé")]
    public int UneMethode(int UnParametre)
    {
        return UnParametre;
    }
}
```

Définir les paramètres nommés

Les paramètres nommés correspondent à des champs ou propriétés publics, ils sont optionnels :

```
using System;
public class ExempleAttribute : Attribute
{
    private string titre;
    public ExempleAttribute(string titre)
    {
        this.titre = titre;
        numero = 0; // valeur par défaut
    }
    public int numero;
}

[Exemple("Un exemple de classe", numero=1)]
public class UneClasse
{
    [method:Exemple("Une méthode")]
    [return:Exemple("Retourne le paramètre passé", numero=2)]
    public int UneMethode(int UnParametre)
    {
        return UnParametre;
    }
}
```

Définir la cible

Par défaut l'attribut concerne tous les types de cibles (`All`).

Pour définir les cibles que l'attribut peut concerner, il faut utiliser l'attribut `System.AttributeUsage` sur la classe de l'attribut. Exemple :

```
using System;
[ AttributeUsage( AttributeTargets.Class |   // cible classe
                  AttributeTargets.Struct,   // ou structure
                  AllowMultiple = false ) ]  // une seule fois par classe ou structure
public class ExempleAttribute : Attribute
{
    private string titre;
    public ExempleAttribute(string titre)
```

```
    {
        this.titre = titre;
        numero = 0; // valeur par défaut
    }
    public int numero;
}

[Exemple("Un exemple de classe", numero=1)]
public class UneClasse
{
    // erreur de compilation pour les 2 attributs suivants
    // car l'attribut ne peut concerner une méthode ou une valeur de retour
    [method:Exemple("Une méthode")]
    [return:Exemple("Retourne le paramètre passé", numero=2)]
    public int UneMethode(int UnParametre)
    {
        return UnParametre;
    }
}

// erreur de compilation car l'attribut est utilisé plus d'une fois
[Exemple("Un autre exemple de classe", numero=1)]
[Exemple("Un exemple de classe", numero=2)]
public class UneAutreClasse
{
}
```

L'énumération `AttributeTargets` possède les valeurs suivantes :

- `All` : toutes les cibles possibles. Cette valeur est la combinaison par *ou logique* de toutes les autres valeurs,
- `Assembly`,
- `Module`,
- `Class` : déclaration d'une classe,
- `Struct` : déclaration d'une structure,
- `Interface` : déclaration d'une interface,
- `Constructor` : constructeur d'une classe,
- `Delegate`,
- `Event`,
- `Enum`,
- `Field`,
- `Method` : déclaration d'une méthode,
- `Property` : déclaration d'une propriété,
- `Parameter`,
- `ReturnValue`,
- `GenericParameter` : paramètre d'un générique (template).

Accès dynamique aux attributs

L'accès aux attributs personnalisés se fait en utilisant la *réflexion* qui permet d'accéder dynamiquement aux différents éléments des déclarations (attributs, méthodes d'une classe, ...).

La classe `Type` représente un type de données : une classe, une structure. L'opérateur `typeof` retourne le `Type` de son argument.

Exemple :

```
Type maclasse = typeof(UneClasse);
```

La classe `System.Attribute` possède une méthode statique nommée `GetCustomAttributes` prenant un `Type` en paramètre (ou tout autre objet de *réflexion* tel que méthode, propriété, ...) et retourne un tableau d'attributs :

```
Type maclasse = typeof(UneClasse);
System.Attribute[] attributs = System.Attribute.GetCustomAttributes(maclasse);
```

L'opérateur `is` permet de tester le type réel de l'attribut, et par exemple retrouver l'attribut `Exemple` définit dans la section précédente :

```
foreach(Attribute attr in attributs)
    if (attr is ExempleAttribute)
    {
        ExempleAttribute ex=(ExempleAttribute)attr;
        Console.WriteLine("Exemple : " + ex.titre );
    }
```

Attributs prédéfinis

Le langage C# définit un nombre d'attributs ayant un rôle spécifique lors de la compilation.

Attribut `Conditional`

L'attribut `System.Diagnostics.ConditionalAttribute` s'applique à une méthode qui ne doit être appelée et définie que si le symbole spécifié est défini. Il peut s'agir par exemple d'une méthode de déboggage. Le symbole `DEBUG` est souvent utilisé pour distinguer les versions *Debug* et *Release* des projets sous Visual Studio.

Exemple :

```
[Conditional("DEBUG")]
public void trace(string message) { ... }
```

Si l'attribut est utilisé plus d'une fois, la méthode n'est appelée et définie que si l'un des symboles est défini (Ou logique) :

```
[Conditional("DEBUG"),Conditional("FORCEDEBUG")]
public void trace(string message) { ... }
```

Cet attribut évite d'encadrer systématiquement chaque appel de la méthode par des directives `#if...#endif`.

Il est également applicable aux classes d'attributs. Dans ce cas, les informations associées à l'attribut ne sont ajoutées que si le symbole est défini.

Par exemple :

```
[Conditional("DEBUG")]
```

```
public class Documentation : System.Attribute
{
    string text;

    public Documentation(string text)
    {
        this.text = text;
    }
}

class ExempleDeClasse
{
    // Cet attribut ne sera inclus que si DEBUG est défini.
    [Documentation("Cette méthode affiche un entier.")]
    static void DoWork(int i)
    {
        System.Console.WriteLine(i.ToString());
    }
}
```

Attribut `Obsolete`

L'attribut `System.ObsoleteAttribute` est utilisé pour marquer une entité dont l'utilisation n'est plus recommandée. Le compilateur peut alors générer une erreur ou un avertissement selon les paramètres de l'attribut.

```
[Obsolete("Utilisez plutôt UneNouvelleMethode()")]
public void UneAncienneMethode()
{ ... }
```

Lors de l'appel à la méthode :

```
UneAncienneMethode();
```

le compilateur génèrera un avertissement comportant le message spécifié :

```
Utilisez plutôt UneNouvelleMethode()
```

Si une valeur booléenne est spécifée à la suite du message, elle indique si une erreur doit être générée au lieu d'un avertissement.

Exemple :

```
[Obsolete("Utilisez plutôt UneNouvelleMethode()",true)]
public void UneAncienneMethode()
{ ... }
```

Lors de l'appel à cette méthode, le compilateur génèrera une erreur comportant le message spécifié :

```
Utilisez plutôt UneNouvelleMethode()
```

Attribut `AttributeUsage`

L'attribut `System.AttributeUsageAttribute` déjà vu précédemment, indique l'utilisation d'une classe attribut.

Le seul paramètre obligatoire est une valeur ou une combinaison de valeurs de l'énumération `System.AttributeTargets` indiquant les cibles acceptables pour l'attribut.

Le paramètre `AllowMultiple` (`bool`, `false` par défaut) indique si l'attribut peut être utilisé plusieurs fois pour la même cible.

Le paramètre `Inherited` (`bool`, `true` par défaut) indique si l'attribut est hérité par les sous-classes.

Attribut `DllImport`

L'attribut `System.Runtime.InteropServices.DllImportAttribute` permet d'importer une fonction définie dans une DLL externe.

Le nom de la DLL où la fonction est définie est le seul paramètre obligatoire.

Attribut `Flags`

L'attribut `System.FlagsAttribute` s'applique aux énumérations pour indiquer que plusieurs valeurs peuvent être combinées avec l'opérateur ou (`|`). Cet attribut indique au compilateur de gérer les combinaisons de constantes dans la méthode ToString() de cette énumération.

Attribut `ThreadStatic`

L'attribut `System.ThreadStaticAttribute` s'applique aux variables membres statiques. Cet attribut indique que la variable est allouée pour tout nouveau thread. Ce qui signifie que chaque thread possède une instance différente de la variable. L'initialisation d'une telle variable à la déclaration n'est effectuée que pour l'instance du thread ayant chargé la classe.

Delegates et events

Les délégués

Un délégué (*delegate* en anglais) est l'équivalent .Net d'un pointeur de fonction. Son rôle est d'appeler une ou plusieurs méthodes qui peuvent varier selon le contexte.

Syntaxe

La déclaration d'un délégué définit la signature d'une méthode, dont le type de retour est précédé du mot clé `delegate`.

Exemple :

```
public delegate int CompareDelegate(object a,object b);
```

Cette déclaration produit en fait une classe `CompareDelegate` dérivant de la classe `System.Delegate`. Il est donc possible de la placer en dehors de toute classe ou espace de noms, comme pour une classe normale.

Utilisation

Ce délégué définit donc un nouveau type qui peut être utilisé en paramètre d'une méthode. L'appel à un délégué utilise la même syntaxe que celle d'une méthode.

Exemple :

```
public void TrierTableau(object[] objects, CompareDelegate compare)
{
    int moves;

    // Vérifier que le délégué pointe sur une méthode :
    if (compare==null) return;

    do
    {
        moves = 0;
        for (int i=1 ; i<objects.Length ; i++)
            // Si objects[i-1] > objects[i]
            //   appel au délégué pour comparer deux objets
            if ( compare( objects[i-1], objects[i]) > 0 )
            {
                 // Échange des deux objets mal ordonnés
                 object o=objects[i-1];
                 objects[i-1]=objects[i];
                 objects[i]=o;

                 moves++; // un échange de plus
            }
    }
    while (moves!=0);
}
```

Instanciation

Une variable du type délégué peut être déclarée, comme avec une classe normale :

```
CompareDelegate comparateur;
```

Initialement, cette variable vaut `null` car ne référence aucune méthode.

Ajout et retrait de fonction

Un délégué est associé à une ou plusieurs méthodes qui possèdent toutes la même signature que celui-ci.

L'ajout ou le retrait de fonction se fait par les opérateurs `=`, `+=` et `-=`.

Exemple : soit les deux méthodes suivantes :

```
void AfficherConsole(string message)
{ ... }
void AfficherFenetre(string message)
```

```
{ ... }
```

et le délégué suivant :

```
delegate void AfficherDelegate(string message);
```

```
AfficherDelegate affichage;
affichage = AfficherConsole;
affichage += AfficherFenetre;
affichage("Un message affiché de deux manières différentes en un seul appel");

affichage -= AfficherFenetre; // Ne plus afficher par fenêtre
affichage("Un message sans fenêtre");

affichage = AfficherFenetre; // Fenêtre seulement (affectation par =)
affichage("Un message dans une fenêtre");
```

Il est également possible d'utiliser la syntaxe suivante équivalente à la précédente :

```
affichage += new AfficherDelegate( AfficherFenetre );
affichage -= new AfficherDelegate( AfficherFenetre );
```

Délégué anonyme

Il est possible de créer dynamiquement la fonction associée à un délégué en utilisant la syntaxe anonyme suivante :

```
delegate( arguments )
{ code }
```

Exemple :

```
affichage += delegate(string m)
    { Console.WriteLine("Nouveau message : "+m); }
```

La spécification des arguments est optionnelle, à condition qu'aucun des arguments ne soit utilisé par la fonction. Ce qui ne peut être le cas si un des arguments est de type `out`, car la fonction doit obligatoirement lui attribuer une valeur.

Exemple :

```
affichage += delegate
    { Console.WriteLine("Fonction qui n'utilise pas les arguments"); }
```

Les délégués du framework .Net

La plupart des délégués du framework .Net sont utilisés comme callback, c'est à dire appelés quand un événement se produit (timer, opération terminée, exécution d'un thread, ...). La signature de ces délégués comporte en général un paramètre nommé `state` du type `object`. Ce paramètre correspond à la valeur

transmise au paramètre `state` de la méthode appelée utilisant le délégué.

Ce paramètre étant du type `object` permet de transmettre toute sorte de valeurs : objets, valeurs numériques (grâce à l'auto-boxing), tableaux de valeurs, ...

Les événements

Un événement (*event* en anglais) est déclenché en dehors de l'application, par l'utilisateur (frappe au clavier, clic d'un bouton de souris, ...), par le système (connexion réseau, ...), par une autre application.

Gestion par délégué

Les délégués sont utilisés pour gérer les événements. Toutefois, cela pose un problème si on utilise un délégué public dans une classe :

```
public delegate void PageRecueDelegate(string url, string contenu);
public class ConnectionHttp
{
    public PageRecueDelegate PageRecue;
}
```

Le code utilisant cette classe peut être le suivant :

```
ConnectionHttp getweb = new ConnectionHttp();
getweb.PageRecue = StockerPageRecue;
getweb.PageRecue += AfficherPageRecue;
getweb.PageRecue("","); // appel bidon
```

Les trois principaux problèmes sont les suivants :

- Si l'objet est partagé par plusieurs classes, fonctions, threads ou avec le système (comme c'est le cas pour les composants de l'interface graphique), plusieurs fonctions pourraient déjà avoir été associées au délégué `PageRecue`, et pourraient être supprimées du délégué par une simple affectation,
- L'ajout et le retrait de fonctions au délégué n'est pas *thread-safe*,
- L'appel au delegate ne devrait pas être possible en dehors de la classe `ConnectionHttp`.

Solution : `event`

Les trois problèmes cités précédemment sont résolus par le mot clé `event` :

```
public delegate void PageRecueDelegate(string url, string contenu);
public class ConnectionHttp
{
    public event PageRecueDelegate PageRecue;
}
```

Ce mot clé protège l'accès au délégué de la manière suivante :

- Il n'est plus possible d'utiliser l'affection seule (opérateur =), il faut utiliser += ou -= ;
- L'ajout et le retrait sont réalisés de manière synchrone,
- Il n'est pas possible d'appeler le `delegate` en dehors de la classe où l'`event` est déclaré.

Fonctionnement interne

La protection est réalisée de la manière suivante :

- Le véritable membre délégué est privé (même si l'`event` est public) ;
- L'utilisation des opérateurs `+=` et `-=` est réalisée par des appels aux accesseurs `add` et `remove` de l'`event`.

Il est possible de remplacer les accesseurs par défaut créés par le compilateur. Pour l'exemple précédent, les accesseurs par défaut sont définis ainsi :

```
public delegate void PageRecueDelegate(string url, string contenu);
public class ConnectionHttp
{
    private PageRecueDelegate pageRecue;
    public event PageRecueDelegate PageRecue
    {
        [MethodImpl(MethodImplOptions.Synchronized)]
        add // paramètre value : fonction à ajouter
        {
            pageRecue += value;
        }
        [MethodImpl(MethodImplOptions.Synchronized)]
        remove // paramètre value : fonction à retirer
        {
            pageRecue -= value;
        }
    }
}
```

Les événements du framework .Net

Les événements sont principalement utilisés dans le framework .Net pour les interfaces graphiques. Le délégué correspondant ne retourne rien (`void`) et possède deux paramètres : un objet indiquant la source de l'événement (le contrôle), et un objet du type *`nom_événement`*`EventArgs` dérivant de la classe `EventArgs`, contenant d'autres informations sur l'événement.

Types génériques

Un type générique est en fait un type non spécifié à l'implémentation d'une méthode ou d'une classe. Le type sera déterminé lors de l'utilisation de cette méthode ou classe. En fait, pour chaque type différent utilisé, le compilateur génèrera une nouvelle version de la méthode ou de la classe.

Les types génériques sont utilisés pour effectuer un même traitement sur différents types de données.

Syntaxe

Les types génériques sont identifiés par un nom, et spécifiés entre les signes < et > placés juste après le nom de la méthode ou la classe.

Exemple de méthode

Soit une méthode statique retournant la valeur maximale entre deux valeurs passées en paramètre :

```
public static T max<T>(T a, T b) // Retourne la valeur maximale
{
    return a > b ? a : b ;
}

int entier = max ( 10 , 22 );
double vmax = max ( 3.14 , 1.618 );
```

Le compilateur détermine le type utilisé pour `T` d'après les valeurs des arguments. Dans le cas précédent, il génère deux versions surchargées de la méthode statique :

- `public static int max(int a, int b)`
- `public static double max(double a, double b)`

Exemple de classe

La syntaxe est similaire. Soit une classe gérant une structure en arbre de valeurs de type quelconque :

```
public class Arbre<T>
{
    public T valeur;
    private Arbre<T> _gauche, _droite;

    public Arbre<T> ArbreGauche
    {
        get { return _gauche; }
    }
}
```

L'utilisation de cette classe exige de spécifier explicitement le type utilisé :

```
Arbre<int> ArbreDesEntiers = new Arbre<int>();
ArbreDesEntiers.valeur = 100;
```

Exemple de structure

Les types génériques sont également utilisables avec les structures :

```
public struct Taille<T>
{
    public T largeur, hauteur;
}
```

Plusieurs types génériques

Il est possible d'utiliser plusieurs types génériques pour une méthode ou une classe :

```
public static void affiche<T,U>(T a, U b) // Affiche a et b
{
```

```
    Console.WriteLine("A vaut {0}, et B vaut {1}", a, b);
}

affiche(10,3.1415926);
```

Contraintes sur les types génériques

Il est possible de restreindre les types utilisables à ceux qui implémentent une ou plusieurs interfaces.

Syntaxe

Pour chaque type à contraindre, il faut ajouter une clause `where` :

```
where type : liste_des_interfaces
```

Exemple :

```
public class TableauTriable<T>
    where T : IComparable
{
    //...
}
```

Il est possible d'utiliser `class` ou `struct` pour limiter le type à une classe ou une structure.

Exemple :

```
public class TableauTriable<T>
    where T : struct
{
    //...
}
```

Il est également possible d'ajouter des contraintes sur les constructeurs du type générique :

```
public class TableauTriable<T>
    where T : new()   // T doit avoir un constructeur sans paramètre
{
    public T Creer()
    {
        return new T(); // appel au constructeur de T
    }
}
```

Opérateur `default`

L'opérateur `default` retourne la valeur par défaut du type générique spécifié. Il s'agit de la valeur quand une variable de ce type n'est pas initialisée (`0` pour les nombres, `null` pour les types références).

Exemple:

```
public T maxAbsolu(T a,T b)
{
    if (a==b) return default(T);
    else return a>b ? a : b;
}
```

Alias

Il est possible de définir un alias d'une classe générique spécifique, en utilisant le mot clé `using` :

```
using Entiers = TableauTriable<int>;
Entiers entiers = new Entiers();
```

Équivaut à :

```
TableauTriable<int> entiers = new TableauTriable<int>();
```

Appel de plateforme

L'appel de plateforme permet au code géré (*managed code*) de faire appel à du code non géré (*unmanaged code*) contenu dans une bibliothèque externe, utilisant du code natif à la plateforme.

Syntaxe

La méthode externe doit être déclarée comme statique (mot clé `static`) et externe (mot clé `extern`) ce qui signifie qu'il ne faut aucun bloc de code.

Cette méthode doit obligatoirement utiliser l'attribut `DllImport` de l'espace de nom `System.Runtime.InteropServices` afin de déclarer le nom de la DLL utilisée et d'autres paramètres optionnels permettant l'interopérabilité avec l'application .Net.

Il est conseillé de déclarer ces méthodes externes dans une classe séparée.

Exemple

```
using System.Runtime.InteropServices;

public class Win32
{
    [DllImport("user32.dll", CharSet=CharSet.Auto)]
    public static extern IntPtr MessageBox
        (int hWnd, String text, String caption, uint type);
}

public class HelloWorld
{
    public static void Main()
```

```
    {
        Win32.MessageBox
            (0, "Hello World", "Platform Invoke Sample", 0);
    }
}
```

En savoir plus

- Présentation détaillée de l'appel de plate-forme (http://msdn2.microsoft.com/fr-fr/library/0h9e9t7d(VS.80).aspx) [archive]

Code non vérifié

Le code produit par la compilation d'un programme C# est géré par l'environnement .Net qui effectue diverses vérifications, gère la mémoire en utilisant un ramasse-miette et peut lancer des exceptions en cas d'erreur : référence nulle, divisions par zéro, variable non initialisée, indexation d'un tableau au delà de ses limites.

Le langage C# permet d'utiliser du code non vérifié pour utiliser des pointeurs et d'autres fonctionnalités non sécurisées. Ce mode de fonctionnement est donc à utiliser avec précautions, et à éviter pour les développeurs débutants.

Le recours à du code non vérifié peut être nécessaire pour utiliser le système d'exploitation, un périphérique accédé par adresse mémoire, ...

Le code non vérifié doit obligatoirement être marqué avec le mot-clé `unsafe`. Ce qui permet d'empêcher son exécution dans un contexte non sûr (code provenant d'une source non fiable).

Déclaration

Le mot-clé unsafe peut être ajouté à la déclaration d'une méthode comme dans l'exemple suivant :

```
public unsafe void Methode()
{
    int iData = 10;
    int* pData = &iData;
    Console.WriteLine("Data contient " + *pData);
    Console.WriteLine("Son addresse est " + (int)pData );
}
```

Il est également possible de l'ajouter pour un bloc d'instruction seul :

```
public void Methode()
{
    int iData = 10;
    unsafe
    {
        int* pData = &iData;
```

```
        Console.WriteLine("Data contient " + *pData);
        Console.WriteLine("Son addresse est " + (int)pData );
    }
}
```

Pointeurs

Un pointeur est un type qui stocke une adresse vers une donnée du type spécifié par le pointeur. La syntaxe d'utilisation est la même que dans les langages C et C++.

Déclaration

La déclaration d'un pointeur utilise un type suivi du caractère étoile.

Exemple :

```
int* pEntier; // Pointeur d'entier
```

Adresse d'une variable

Un pointeur peut recevoir l'adresse d'une variable, en faisant précéder la variable du caractère `&`.

Exemple :

```
int total;
pEntier = &total; // adresse de la variable total
```

Déréférencer un pointeur

Un pointeur utilisé directement donnera l'adresse de la variable. Pour utiliser le contenu pointé par le pointeur il faut le déréférencer en le faisant précéder du caractère étoile `*`.

Exemple :

```
*pEntier = 100; // modification de la variable total
```

Membre d'une classe pointée

Pour accéder à un membre d'un objet pointé il est possible d'utiliser la syntaxe suivante :

```
(*pEntier).ToString(); // accès à la méthode ToString de l'entier pointé
```

Ou d'utiliser l'opérateur flèche équivalent :

```
pEntier->ToString(); // accès à la méthode ToString de l'entier pointé
```

Pointeur et tableau

L'adresse d'un tableau est donnée sans utiliser l'opérateur d'adresse `&`. Toutefois, il n'est pas possible de modifier l'adresse du tableau afin d'éviter de perdre l'adresse de début du tableau. Le pointeur doit utiliser le mot-clé `fixed` pour obtenir l'adresse d'un tableau.

Exemple :

```
int[] valeurs = new int[10];
fixed (int* pEntier = valeurs)
for (int iIndex = 0; iIndex < 10; iIndex++)
    Console.WriteLine( *(pEntier + iIndex) );
```

Gestion de la mémoire

Le mode non vérifié permet de modifier le comportement du ramasse-miettes.

Éviter le déplacement par le ramasse-miettes

Le mot clé `fixed` sert à éviter qu'un tableau ou un objet ne soit déplacé en mémoire par le ramasse-miettes :

- pour empêcher le déplacement durant l'exécution d'une instruction ou un bloc d'instructions :

```
fixed (type* pointeur = adresse) instruction
```

- pour déclarer un tableau de taille fixe (non déplacé en mémoire) :

```
fixed type[nombre] variable;
```

Exemple :

```
protected fixed int[12] jours_par_mois;
```

Taille d'une variable ou d'un type

L'opérateur `sizeof` s'utilise dans un contexte de code non vérifié, comme une fonction renvoyant le nombre d'octets occupés par la variable ou le type spécifié.

Exemple :

```
int a;
unsafe
{
    System.out.println("Taille de a : "+ sizeof(a) +" octets");
    System.out.println("Taille d'un entier : "+ sizeof(int) +" octets");
}
```

Allocation sur la pile

Le mot-clé `stackalloc` permet d'allouer un objet ou un tableau sur la pile plutôt que sur le tas. Dans ce cas, cet objet ou tableau n'est pas géré par le ramasse-miettes. Il est donc possible d'utiliser un pointeur sans

utiliser le mot-clé `fixed`.

Syntaxe : le mot-clé `stackalloc` s'utilise à la place du mot-clé `new` pour initialiser des pointeurs locaux.

Exemple :

```
unsafe
{
    // allouer 10 entiers sur la pile
    int* pEntier = stackalloc int[10];
    ...
}
```

Interfaces graphiques

Les applications de la plateforme .Net construisent leur interface graphique à partir de *forms*.

Espaces de nom

Les deux principaux espaces de nom de l'interface graphique sont :

`System.Windows.Forms`
: Cet espace de nom contient les classes correspondant aux divers composants de l'interface graphique (fenêtre, bouton, label, ...).

`System.Drawing`
: Cet espace de nom gère le dessin dans un composant (ligne, rectangle, texte, image, ...).

Compilation

Si l'application possédant une interface graphique n'a pas besoin de console, il est possible de la supprimer en spécifiant `winexe` pour le paramètre `target` dans la ligne de commande du compilateur :

Windows	`csc /t:winexe fichier.cs`
Linux (Mono)	`gmsc -target:winexe fichier.cs`

Une première fenêtre

La fenêtre est gérée par la classe `Form` de l'espace de nom `System.Windows.Forms`. En général, l'application dérive de cette classe pour ajouter des attributs de type composants, et gérer les évènements.

Cette première fenêtre comportera un label « Hello world ! » et un bouton « Fermer ».

Source :

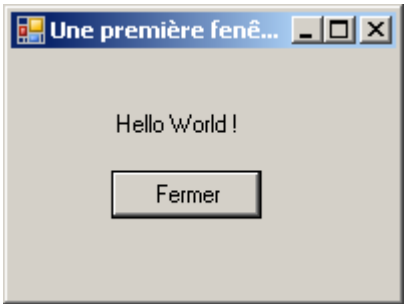
```
using System;
```

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

public class PremiereFenetre : Form
{
    private Label message;
    private Button fermer;

    public PremiereFenetre()
    {
        SuspendLayout();
        Text = "Une première fenêtre";    // Le titre de la fenêtre
        Size = new Size(200, 150);        // La taille initiale
        MinimumSize = new Size(200, 150); // La taille minimale

        // Le label "Hello world !"
        message = new Label();
        message.Text = "Hello World !";
        message.AutoSize = true;              // Taille selon le contenu
        message.Location = new Point(50, 30);// Position x=50 y=30

        // Le bouton "Fermer"
        fermer = new Button();
        fermer.Text = "Fermer";
        fermer.AutoSize = true;             // Taille selon le contenu
        fermer.Location = new Point(50, 60);// Position x=50 y=60

        fermer.Click += new System.EventHandler(fermer_Click);

        // Ajouter les composants à la fenêtre
        Controls.Add(message);
        Controls.Add(fermer);

        ResumeLayout(false);
        PerformLayout();
    }

    // Gestionnaire d'événement
    private void fermer_Click(object sender, EventArgs evt)
    {
        // Fin de l'application :
        Application.Exit();
    }

    static void Main()
    {
        // Pour le style XP :
        Application.EnableVisualStyles();

        // Lancement de la boucle de messages
        // pour la fenêtre passée en argument :
        Application.Run(new PremiereFenetre());
    }
}
```

Les propriétés des composants sont initialisées. Puis les composants sont ajoutés à la collection `Controls` des composants de la fenêtre.

La gestion de l'évènement du clic sur le bouton se fait à l'aide de l'event `Click`.

Si la compilation se déroule bien, le lancement de l'application affiche la fenêtre suivante :



Cette fenêtre est redimensionnable, mais ne peut être plus petite que la taille minimale spécifiée dans le fichier source par la propriété `MinimumSize`.

L'interface avec un éditeur graphique

un éditeur graphique permet de créer facilement l'interface graphique grâce à une barre d'outils permettant d'ajouter des composants à la fenêtre de l'application, et de modifier leur propriétés.

Les ressources associées à l'interface graphique sont :

- les icônes, dont celle de l'application affichée dans le coin supérieur gauche de la fenêtre,
- les images,
- les curseurs de souris personnalisés,
- les chaînes de caractères (ressources en plusieurs langues).

Les éditeurs disponibles sont :

- Visual Studio et sa version Express gratuite, le plus diffusé
- SharpDevelop, un éditeur développé par une communauté de passionnés
- MonoDevelop, un éditeur développé par le projet Mono

Les composants

Tous les composants ont pour classe de base `System.Windows.Forms.Control`. Cette classe définit les fonctionnalités communes à tous les composants.

Les évènements

La gestion des évènements est assurée par l'utilisation d'events dont la méthode delegate correspondante a la signature suivante :

```
void nom_method(object sender, nom_évènementEventArgs e)
```

Le premier argument indique la source de l'évènement (le contrôle). Le deuxième argument donne des informations sur l'évènement (position de la souris, touche utilisée, ...).

Interfaces graphiques/Graphique vectoriel

Espaces de nom

`System.Drawing.Drawing2D`
: Cet espace de nom contient des classes permettant la conception de graphiques vectoriels avancés en 2D.

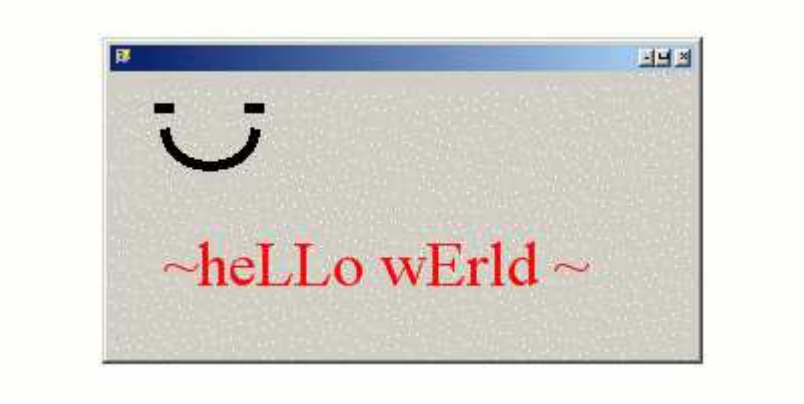
Un premier graphique

```
using System;
using System.ComponentModel;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

namespace test
{
    public partial class Form1 : Form
    {
        private int yPos = 40;
        private PictureBox pictureBox1;
        private Graphics g;

        public Form1()
        {
            this.InitializeComponent();
            this.Show();
        }

        private void InitializeComponent()
        {
            this.pictureBox1 = new PictureBox();
            ((ISupportInitialize) this.pictureBox1).BeginInit();
            this.SuspendLayout();
            //
            // pictureBox1
            //
            this.pictureBox1.Location = new Point(12, 12);
            this.pictureBox1.Name = "pictureBox1";
            this.pictureBox1.Size = new Size(268, 112);
            this.pictureBox1.TabIndex = 0;
            this.pictureBox1.TabStop = false;
            //
            // Form1
            //
            this.ClientSize = new Size(292, 141);
            this.Controls.Add(this.pictureBox1);
            this.Name = "Form1";
            ((ISupportInitialize) this.pictureBox1).EndInit();
            this.ResumeLayout(false);

        }

        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }

        private void Show()
```

```
        {
            Bitmap bmp = new Bitmap(
                pictureBox1.Width, pictureBox1.Height);

            using (Graphics image = Graphics.FromImage(bmp))
            {
                Pen myPen1 = new Pen(Color.Red);
                Pen myPen2 = new Pen(Color.Black);

                myPen1.Width = 5;
                myPen2.Width = 5;

                SolidBrush myBrush = new SolidBrush(Color.Red);
                Font myFont = new Font("Times New Roman", 24);

                // smile
                image.DrawLine(myPen2, 10, 5, 20, 5);
                image.DrawLine(myPen2, 55, 5, 65, 5);
                image.DrawBezier(myPen2, 15, 15, 15, 40, 60, 40, 60, 15);

                image.DrawString("~heLLo wErld ~", myFont, myBrush,
                    10, (this.yPos = this.yPos + 25));

            } // using
            pictureBox1.Image = bmp;
        }
    }
}
```

Les méthodes appliquées ici sont issues de la classe `Graphics` de .NET.

Pour afficher l'image il faut initialiser auparavant un constructeur de cette classe ou l'appeler par un évènement on_click par exemple.



Quelques méthodes

La classe `Graphics` comprend de nombreuses autres méthodes du genre. Dont :

1. DrawArc : Dessine un arc représentant une partie d'une ellipse indiquée par une paire de coordonnées, d'une largeur, et d'une taille.
2. DrawBezier : Dessine une cannelure de Bézier définie par quatre structures de point.
3. DrawBeziers : Tire une série de cannelures de Bézier d'un choix de structures de point.
4. DrawClosedCurve : Dessine une cannelure cardinale fermée définie par un choix de structures de point.
5. DrawCurve : Dessine une cannelure cardinale par un choix indiqué de structures de point.
6. DrawEllipse : Dessine une ellipse définie par un rectangle de bondissement indiqué par une paire des

coordonnées, d'une taille, et d'une largeur.
7. DrawIcon : Dessine l'image représentée par l'icône indiquée aux coordonnées indiquées.
8. DrawIconUnstretched : dessine l'image représentée par l'icône indiquée sans mesurer l'image.
9. DrawImage : Dessine l'image indiquée à l'endroit indiqué et avec le format document.
10. DrawImageUnscaled : Dessine l'image indiquée en utilisant son taille physique originale à l'endroit indiqué par une paire du même rang.
11. DrawImageUnscaledAndClipped : dessine l'image indiquée sans graduation et agrafes il, au besoin, à l'ajustement dans le rectangle indiqué.
12. DrawLine : Trace une ligne reliant les deux points indiqués par les paires du même rang.
13. DrawLines : Dessine une série de ligne segments qui relient un choix de structures de point. dessine un GraphicsPath.
14. DrawPie : Dessine une forme de pâté en croûte définie par une ellipse indiquée par une paire du même rang, une largeur, une taille, et deux lignes radiales.
15. DrawPolygon : Dessine un polygone défini par un choix de structures de point.
16. DrawRectangle : Dessine un rectangle indiqué par une paire du même rang, une largeur, et une taille.
17. DrawRectangles : Dessine une série de rectangles indiqués par des structures de Rectangle. Cordon Surchargé. Dessine la corde indiquée des textes à l'endroit indiqué avec la brosse indiquée et la police objecte.

Fonctions asynchrones

L'appel à une fonction se fait de manière synchrone : aucune autre instruction n'est exécutée avant que la fonction ne retourne une valeur. Cependant certaines fonctions prennent beaucoup de temps, en particulier les opérations d'entrées-sorties, et les fonctions de communication par réseau informatique.

Pour ce genre de fonction, l'API .Net possède souvent deux versions de la méthode :

- Une méthode synchrone qui attend la fin de l'opération avant d'en retourner le résultat,
- Une méthode asynchrone demandant au pool de threads standard d'effectuer l'opération, puis retourne immédiatement à la méthode appelante.

Syntaxe et utilisation des méthodes asynchrones

Une méthode synchrone est déclarée selon la syntaxe suivante :

```
type_retour nom_methode ( arguments... )
```

La version asynchrone utilise deux autres méthodes. La première commence par `Begin` et demande au pool de threads standard d'effectuer l'opération. Une fois l'opération terminée, un delegate est appelé. Celui-ci appelle alors la méthode dont le nom commence par `End` pour récupérer la valeur retournée par la méthode asynchrone.

Ces deux méthodes ont la syntaxe suivante :

```
IAsyncResult Beginnom_methode ( arguments_non_ref...,
     AsyncCallback callback, object state )
```

```
type_retour Endnom_methode ( IAsyncResult, arguments_ref )
```

Les arguments de la méthode synchrone se retrouvent répartis entre les deux méthodes :

- Les paramètres passés par valeur sont passés à la méthode `Begin`,
- Les paramètres de sorties (out et ref) sont passés à la méthode `End`.

La méthode `Begin` possède deux arguments supplémentaires :

- Un delegate de type AsyncCallback appelé quand l'opération est terminée (celui-ci doit alors appeler la méthode `End` correspondante),
- Un objet à transmettre au delegate.

Le delegate `AsyncCallback`

La syntaxe du delegate `AsyncCallback` est la suivante :

```
public delegate void AsyncCallback(IAsyncResult result)
```

Le paramètre `result` est une interface de type `IAsyncResult` correspondant à celui retourné par la méthode `Begin`, et doit être passé à la méthode `End`.

L'interface `IAsyncResult`

L'interface `IAsyncResult` possède les propriétés en lecture seule suivantes :

- Propriété généralement utilisée par le delegué appelé quand l'opération est terminée :
  object AsyncState
  Cet objet correspond à celui transmis à la méthode Begin.
- Propriétés généralement utilisées par le code appelant la méthode Begin de l'opération asynchrone :
  WaitHandle AsyncWaitHandle
  Cet objet de synchronisation est mis dans l'état signalé quand l'opération est terminée.
  bool IsCompleted
  Ce booléen vaut true (vrai) lorsque l'opération est terminée.
  bool CompletedSynchronously
  Ce booléen vaut true (vrai) si l'opération s'est terminée de manière synchrone.

La propriété `AsyncWaitHandle` permet de lancer une opération asynchrone, d'effectuer d'autres traitements durant l'opération en cours, et finalement attendre la fin de l'opération si elle ne s'est pas déjà terminée, en testant la propriété `IsCompleted`.

Threads et synchronisation

Un thread est un contexte d'exécution ayant sa propre pile de paramètres et de variables locales, mais partageant les mêmes variables globales (variables statiques des classes) que les autres threads du même processus (la même instance d'une application créée au moment de son lancement).

Initialement, un processus ne possède qu'un seul thread. En général, celui-ci crée d'autres threads pour le traitement asynchrone de la file de messages provenant du système d'exploitation (gestion des évènements), du matériel géré, ...

Créer et démarrer un thread

Un thread est créé pour effectuer une tâche en parallèle d'une autre. Si l'application possède une interface graphique et doit effectuer une tâche qui prend du temps (calcul, téléchargement, ...), si un seul thread est utilisé, durant la longue tâche l'interface graphique est inutilisable. Il vaut mieux effectuer la tâche longue dans un nouveau thread afin que l'utilisateur puisse continuer à utiliser l'interface graphique (pour annuler cette tâche par exemple).

Les threads sont également nécessaires pour gérer plusieurs tâches indépendantes les unes des autres, comme par exemple pour un serveur, la gestion de connexions simultanées avec plusieurs clients. Dans ce cas, chaque thread effectue les mêmes tâches. Ils sont en général gérés par un ensemble de threads (*Thread pool* en anglais).

Les threads et les outils associés sont gérés par les classes de l'espace de nom `System.Threading`. La classe `Thread` gère un thread. Son constructeur accepte comme premier paramètre :

- soit un délégué de type `ThreadStart` :

```
delegate void ThreadStart()
```

- soit un délégué de type `ParameterizedThreadStart` :

```
delegate void ParameterizedThreadStart(object parameter)
```

Le second paramètre est optionnel : `int maxStackSize`. Il indique la taille maximale de la pile à allouer au nouveau thread.

Une fois le thread créé, il faut appeler la méthode `Start` pour démarrer le thread :

- soit `Start()` si le `delegate` n'a aucun paramètre (`ThreadStart`),
- soit `Start(object parameter)` si le `delegate` accepte un paramètre de type `object` (`ParameterizedThreadStart`).

Exemple 1 :

```
// La tâche qui prend du temps ...
private void longTask()
{
    Console.WriteLine("Début de la longue tâche dans le thread "
        + Thread.CurrentThread.GetHashCode());
    //...
}

// Démarre un nouveau thread pour la méthode longTask()
public void StartLongTask()
{
    Console.WriteLine("Création d'un thread à partir du thread "
        + Thread.CurrentThread.GetHashCode());
```

```
    Thread th = new Thread(
        new ThreadStart( this.longTask )
        );
    th.Start();
}
```

Exemple 2 : Une méthode acceptant un paramètre.

```
// La tâche qui prend du temps ...
private void telecharger(string url)
{
    Console.WriteLine("Début du téléchargement de " + url +
        " dans le thread " + Thread.CurrentThread.GetHashCode());
    //...
}

// Démarre un nouveau thread pour la méthode telecharger()
public void CommencerATelecharger(string url)
{
    Console.WriteLine("Création d'un thread à partir du thread "
        + Thread.CurrentThread.GetHashCode());

    Thread th = new Thread(
        new ParameterizedThreadStart( this.telecharger )
        );
    th.Start( url );
}
```

Exemple 3 : Si la méthode doit accepter plusieurs paramètres, il faut passer un tableau de paramètres. Puisqu'un tableau est egalement un objet, le tableau peut être passé sous la forme d'une référence d'objet, qui sera reconverti en tableau dans la méthode du delegate.

```
// La tâche qui prend du temps ...
private void telecharger(object object_parameters)
{
    object[] parameters = (object[]) object_parameters;
    string url = (string) parameters[0];
    int tentatives = (int) parameters[1];
    Console.WriteLine("Début du téléchargement de " + url +
        " ("+tentatives+" tentatives) dans le thread " +
        Thread.CurrentThread.GetHashCode());
    //...
}

// Démarre un nouveau thread pour la méthode telecharger()
public void CommencerATelecharger(string url, int tentatives)
{
    Console.WriteLine("Création d'un thread à partir du thread "
        + Thread.CurrentThread.GetHashCode());

    Thread th = new Thread(
        new ParameterizedThreadStart( this.telecharger )
        );

    th.Start( new object[] { url, tentatives } );
}
```

Attendre la fin d'un thread

Une fois qu'un thread a démarré, la méthode join peut être appelée pour attendre que la méthode du thread se termine. Cette méthode est surchargée :

- `void Join()` : attend indéfiniment que le thread se termine,
- `bool Join(int millisecondsTimeout)` : attend que le thread se termine dans le temps imparti spécifié en millisecondes. Cette méthode retourne `true` si le thread s'est terminé dans le temps imparti, et `false` sinon,
- `bool Join(TimeSpan timeout)` : cette méthode fonctionne de la même manière que la précédente, excepté que le temps imparti est spécifié par une structure de type `System.TimeSpan`.

Exemple :

```
// Démarre un nouveau thread pour la méthode longTask()
public void startLongTask()
{
    Console.Writeline("Création d'un thread à partir du thread "
        + Thread.CurrentThread.GetHashCode());

    Thread th = new Thread(
        new ThreadStart( this.longTask )
        );

    Console.Writeline("Démarrage du thread " + th.GetHashCode() );
    th.Start();

    Console.Writeline("Attente de la fin du thread " + th.GetHashCode()
        + " pendant 5 secondes ...");
    bool fini = th.Join(5000); // 5000 ms

    if (!fini) // si pas fini
    {
        Console.Writeline("Le thread " + th.GetHashCode()
            + " n'a pas terminé au bout de 5 secondes, attente indéfinie ...");
        th.Join();
    }

    Console.Writeline("Le thread " + th.GetHashCode() + " a terminé sa tâche.");
}
```

Suspendre et Arrêter un thread

Suspendre un thread

Pour suspendre un thread, il faut appeler la méthode `Suspend()`. Le thread est alors suspendu jusqu'à l'appel à la méthode `Resume()`. Cependant ces deux méthodes sont obsolètes, car un thread suspendu détient toujours les ressources qu'il avait acquis avant sa suspension. Ce problème risque de provoquer le blocage d'autres threads tant que celui-là est suspendu.

Arrêter un thread

L'arrêt d'un thread peut être demandé en appelant la méthode `Interrupt()`. Cette méthode provoque le lancement d'une exception lorsque le thread appelera une méthode d'entrée-sortie. Donc l'arrêt du thread n'est pas instantané.

Cependant l'appel à cette méthode peut interrompre le thread à n'importe quel moment de son exécution. Il faut donc prévoir que ce genre d'exception soit lancé pour pouvoir libérer les ressources dans un bloc

`try..finally`, voire utiliser `using`).

Une solution alternative est de tester une condition de terminaison du thread. Ceci permet de spécifier où le thread peut se terminer, et libérer les ressources correctement.

Exemple :

```
public void UnThread()
{
    // ... initialisation
    while ( continuer_thread )
    {
        // ... tâches du thread
    }
    // ... libération des ressources
}
```

Cependant, l'utilisation d'un bloc `try..finally` (ou du mot clé `using`) est tout de même nécessaire au cas où le système interromperait le thread (fin brutale de l'application par exemple).

Propriétés d'un thread

Voici une liste des principales propriétés de la classe `Thread` :

`bool IsAlive`
: (lecture seule) Indique si le thread est toujours actif.

`bool IsBackground`
: Indique si le thread est un thread d'arrière plan. Un thread d'arrière plan ne permet pas de maintenir l'exécution d'une application. C'est à dire qu'une application est terminée dès qu'il n'y a plus aucun thread de premier plan en cours d'exécution.

`bool IsThreadPoolThread`
: (lecture seule) Indique si le thread appartient au pool de threads standard.

`int ManagedThreadId`
: (lecture seule) Retourne l'identifiant attribué au thread par le framework .Net.

`string Name`
: Nom attribué au thread.

`ThreadPriority Priority`
: Priorité du thread :
- `Lowest` : priorité la plus basse,
- `BelowNormal` : priorité inférieure à normale,
- `Normal` : priorité normale,
- `AboveNormal` : priorité supérieure à normale,
- `Highest` : priorité la plus haute.

`ThreadState ThreadState`
: (lecture seule) Etat actuel du thread.

  Cette propriété est une combinaison de valeurs de l'énumération `ThreadState` :
- `Running = 0x00000000` : en cours d'exécution,
- `StopRequested = 0x00000001` : arrêt demandé,
- `SuspendRequested = 0x00000002` : suspension demandée,
- `Background = 0x00000004` : thread d'arrière plan,
- `Unstarted = 0x00000008` : thread non démarré,

- `Stopped = 0x00000010` : thread arrêté,
- `WaitSleepJoin = 0x00000020` : thread en attente (wait, sleep, ou join),
- `Suspended = 0x00000040` : thread suspendu,
- `AbortRequested = 0x00000080` : abandon demandé,
- `Aborted = 0x00000100` : thread abandonné.

Le pool de threads

Un pool de threads est associé à chaque processus. Il est composé de plusieurs threads réutilisables effectuant les tâches qu'on lui assigne dans une file d'attente. Une tâche est placée dans la file d'attente, puis on lui affecte un thread inocupé qui l'effectuera, puis le thread se remet en attente d'une autre tâche. Une partie de ces threads est consacrée aux opérations d'entrées-sorties asynchrones.

Le pool de threads est géré par les méthodes statiques de la classe `ThreadPool`. Par défaut, le pool contient 25 threads par processeur.

La méthode QueueUserWorkItem permet d'ajouter une nouvelle tâche :

```
bool QueueUserWorkItem ( WaitCallback callback, object state )
```

Le premier paramètre est un délégué dont la signature est la suivante :

```
delegate void WaitCallback(object state)
```

Le second paramètre est optionnel et contient l'argument passé au délégué.

Le délégué passé en paramètre effectuera la tâche, dans le thread qui lui sera attribué.

Synchronisation entre les threads

Le fait que tous les threads d'un même processus partagent les mêmes données signifie que les threads peuvent accéder simultanément à un même objet. Si un thread modifie un objet (écriture, en utilisant une méthode de l'objet par exemple) pendant qu'un autre thread récupère des informations sur cet objet (lecture), ce dernier peut obtenir des données incohérentes résultant d'un état intermédiaire temporaire de l'objet accédé.

Pour résoudre ce genre de problème, des outils de synchronisation permettent de suspendre les threads essayant d'accéder à un objet en cours de modification par un autre thread.

Moniteur

Un moniteur (*monitor* en anglais) ne permet l'accès qu'à un seul thread à la fois. C'est à dire que si plusieurs threads essaye d'accéder au même moniteur, un seul obtiendra l'accès, les autres étant suspendus jusqu'à ce qu'ils puissent à leur tour détenir l'accès exclusif.

La classe Monitor gère ce type d'objet de synchronisation. Toutes les méthodes de cette classe sont statiques. Les principales méthodes sont :

`void Enter(object obj)`
: Cette méthode suspend le thread appelant si un autre thread possède déjà l'accès exclusif, ou retourne

immédiatement sinon.

`void Exit(object obj)`
: Cette méthode met fin à l'accès exclusif par le thread appelant, et permet à un thread suspendu d'obtenir l'accès exclusif à son tour.

`bool TryEnter(object obj)`
: Cette méthode permet de tenter d'obtenir l'accès exclusif. Elle retourne true si l'accès exclusif est obtenu, false sinon.

`bool TryEnter(object obj,int milliseconds)`
: Cette méthode permet de tenter d'obtenir l'accès exclusif, dans le temps imparti spécifié en millisecondes. Elle retourne true si l'accès exclusif est obtenu, false sinon.

L'objet passé en paramètre identifie le moniteur accédé. C'est à dire que tout objet peut être utilisé comme moniteur. En C#, tout est objet, même les chaînes de caractères, et les valeurs numériques et booléennes. Cependant, il n'est pas recommandé d'utiliser de telles valeurs ou des références publiques, car ce sont des références globales. Il est préférable d'utiliser des membres privés, voire des variables locales.

Exemple :

```
using System;
using System.Threading;
public class TestMonitor
{
    private object synchro = new object();
    public void MethodeThread()
    {
        int id = Thread.CurrentThread.ManagedThreadId;
        Console.WriteLine("Début du thread " + id );

        Monitor.Enter( synchro );

        Console.WriteLine("Le thread " + id + " entre exclusivement ..." );
        Thread.Sleep(1000); // attend 1 seconde
        Console.WriteLine("Le thread " + id + " sort ..." );

        Monitor.Exit( synchro );

        Console.WriteLine("Fin du thread " + id );
    }

    public void Test()
    {
        // 2 threads
        Thread
            thread1 = new Thread( new ThreadStart( MethodeThread ) ),
            thread2 = new Thread( new ThreadStart( MethodeThread ) );
        thread1.Start();
        thread2.Start();
    }

    public static void Main()
    {
         new TestMonitor().Test();
    }
}
```

Ce programme affiche :

```
Début du thread 3
```

```
Le thread 3 entre exclusivement ...
Début du thread 4
Le thread 3 sort ...
Le thread 4 entre exclusivement ...
Fin du thread 3
Le thread 4 sort ...
Fin du thread 4
```

Ce programme ne tient pas compte des exceptions. Il faut cependant les prévoir pour s'assurer de libérer le moniteur, en utilisant un bloc `try..finally` :

```
...
    public void MethodeThread()
    {
        int id = Thread.CurrentThread.ManagedThreadId;
        Console.WriteLine("Début du thread " + id );

        Monitor.Enter( synchro );
        try
        {
            Console.WriteLine("Le thread " + id + " entre exclusivement ..." );
            Thread.Sleep(1000); // attend 1 seconde
            Console.WriteLine("Le thread " + id + " sort ..." );
        }
        finally
        {
            Monitor.Exit( synchro );
        }

        Console.WriteLine("Fin du thread " + id );
    }
...
```

Le mot clé `lock`

Le mot clé `lock` est utilisable pour produire un code équivalent au précédent :

```
...
    public void MethodeThread()
    {
        int id = Thread.CurrentThread.ManagedThreadId;
        Console.WriteLine("Début du thread " + id );

        lock( synchro ) // <- enter ( synchro )
        {
            Console.WriteLine("Le thread " + id + " entre exclusivement ..." );
            Thread.Sleep(1000); // attend 1 seconde
            Console.WriteLine("Le thread " + id + " sort ..." );
        } // <- exit ( synchro )

        Console.WriteLine("Fin du thread " + id );
    }
...
```

Le mot clé `lock` est suivi de la référence de l'objet dont l'accès doit être exclusif durant le bloc de code qui suit.

Autres outils de synchronisation

Les autres outils de synchronisation dérivent de la classe abstraite `WaitHandle`. Les objets `WaitHandle` possèdent deux états :

- l'**état signalé** (*set*) dans lequel l'objet permet aux threads d'accéder à la ressource,
- l'**état non signalé** (*reset*) ne permet pas de nouvel accès. Toute tentative d'accès suspendra le thread, jusqu'au retour de l'état signalé.

Attendre l'état signalé d'un objet `WaitHandle`

Les méthodes suivantes de la classe `WaitHandle` permettent d'attendre l'état signalé d'un objet `WaitHandle` :

```
// Attendre que l'objet WaitHandle soit dans l'état signalé :
public virtual bool WaitOne();
public virtual bool WaitOne(int millisecondsTimeout, bool exitContext);
public virtual bool WaitOne(TimeSpan timeout, bool exitContext);
```

Il est possible d'appeler ces méthodes en spécifiant un temps limite (`int`: nombre de millisecondes ou `TimeSpan`) et un indicateur booléen `exitContext`. Ce paramètre vaut `true` si le contexte de synchronisation (l'ensemble des verrous) doit être libéré avant l'attente puis récupéré ensuite. Il vaut `false` sinon, c'est à dire que les verrous seront toujours détenus par le thread durant son attente.

Ces méthodes retournent `true` si l'état est signalé, et `false` sinon (temps limite expiré).

Attendre l'état signalé de plusieurs objets `WaitHandle`

Les méthodes statiques suivantes de la classe `WaitHandle` permettent d'attendre l'état signalé pour un tableau d'objets `WaitHandle` :

```
// Attendre que tous les objets WaitHandle soit dans l'état signalé :
public static bool WaitAll(WaitHandle[] waitHandles);
public static bool WaitAll(WaitHandle[] waitHandles,
                           int millisecondsTimeout, bool exitContext);
public static bool WaitAll(WaitHandle[] waitHandles,
                           TimeSpan timeout, bool exitContext);

// Attendre qu'au moins l'un des objets WaitHandle soit dans l'état signalé :
public static int WaitAny(WaitHandle[] waitHandles);
public static int WaitAny(WaitHandle[] waitHandles,
                          int millisecondsTimeout, bool exitContext);
public static int WaitAny(WaitHandle[] waitHandles,
                          TimeSpan timeout, bool exitContext);
```

Le tableau d'objets `WaitHandle` ne doit pas comporter de doublon, sinon une exception `DuplicateWaitObjectException` est levée.

Il est possible d'appeler ces méthodes en spécifiant un temps limite (`int`: nombre de millisecondes ou `TimeSpan`) et un indicateur booléen `exitContext` (voir section précédente).

Les méthodes `WaitAll` retournent `true` si l'état de tous les objets est signalé, ou `false` sinon (temps limite expiré).

Les méthodes `WaitAny` retournent l'indice dans le tableau de l'objet dont l'état est signalé, ou la constante `WaitHandle.WaitTimeout` sinon (temps limite expiré).

Signaler et attendre

Les méthodes statiques suivantes permettent de mettre l'état signalé sur un objet `WaitHandle` et d'attendre un autre objet `WaitHandle` :

```
// Mettre l'état signalé sur l'objet toSignal
// Ensuite, attendre que l'objet toWaitOn soit dans l'état signalé :
public static bool SignalAndWait(WaitHandle toSignal, WaitHandle toWaitOn);
public static bool SignalAndWait(WaitHandle toSignal, WaitHandle toWaitOn,
                                 int millisecondsTimeout, bool exitContext);
public static bool SignalAndWait(WaitHandle toSignal, WaitHandle toWaitOn,
                                 TimeSpan timeout, bool exitContext);
```

Évènements

Un évènement est une instance de la classe `EventWaitHandle` (sous-classe de la classe `WaitHandle` vue précédemment). Il permet de modifier son état signalé / non-signalé grâce aux deux méthodes suivantes :

```
public bool Reset(); // -> état non-signalé
public bool Set();   // -> état signalé
```

Cette classe possède le constructeur suivant :

```
public EventWaitHandle(bool initialState, EventResetMode mode);
```

Les paramètres sont les suivants :

initialState
: État initial : signalé (`true`) ou non-signalé (`false`).

mode
: Mode pour le retour à l'état non-signalé : AutoReset ou ManualReset :
- ManualReset : le retour à l'état non-signalé (*reset*) se fait explicitement en appelant la méthode `Reset()`.
- AutoReset : le retour à l'état non-signalé est automatiquement effectué quand un thread est activé, c'est à dire quand il a terminé d'attendre l'état signalé avec une méthode wait (WaitOne, WaitAny ou WaitAll).

Il y a une sous-classe pour chacun des deux modes (voir ci-dessous pour une description).

Synchronisation inter-processus

Cette classe possède également le constructeur suivant :

```
public EventWaitHandle(bool initialState, EventResetMode mode,
                       string name, ref Boolean createdNew,
                       System.Security.AccessControl.EventWaitHandleSecurity eventSecurit
```

Les trois paramètres supplémentaires sont tous optionnels et sont utilisés pour le partage au niveau système et donc pour la synchronisation entre processus :

name

Nom unique identifiant cette instance de la classe `EventWaitHandle`.

createdNew

Référence à une variable booléenne que la fonction va utiliser pour indiquer si un nouvel objet `EventWaitHandle` a été créé (`true`) ou s'il existe déjà (`false`).

eventSecurity

Définit les conditions d'accès à l'objet partagé.

Elle possède également une méthode statique permettant de retrouver une instance de la classe `EventWaitHandle` existante partagée au niveau système :

```
public static EventWaitHandle OpenExisting(string name, System.Security.AccessControl.Eve
```

Le paramètre `rights` est optionnel et permet d'accéder à l'objet partagé.

Sous-classes

Cette classe a deux sous-classes :

- `AutoResetEvent` : Le retour à l'état non-signalé est automatique. Une fois que l'état signalé est obtenu par un thread (fin de l'attente par une méthode wait), l'objet `AutoResetEvent` revient à l'état non-signalé, empêchant un autre thread d'obtenir l'état signalé.
- `ManualResetEvent` : Le retour à l'état non-signalé se fait explicitement.

Ces deux sous-classes ont un constructeur qui accepte comme paramètre un booléen `initialState` indiquant son état initial : `true` pour l'état signalé, `false` pour l'état non-signalé.

Exemple :

```
using System;
using System.Threading;

class Exemple
{
    static AutoResetEvent evenementTermine;

    static void AutreThread()
    {
        Console.WriteLine("   Autre thread :   0% accompli, attente...");

        evenementTermine.WaitOne(); // Attend état signalé + Reset (auto)
        // l'état a été signalé, mais retour à l'état non-signalé
        // maintenant que l'appel à WaitOne est terminé.

        Console.WriteLine("   Autre thread :  50% accompli, attente...");

        evenementTermine.WaitOne(); // Attend état signalé + Reset (auto)
        // l'état a été signalé, mais retour à l'état non-signalé
        // maintenant que l'appel à WaitOne est terminé.

        Console.WriteLine("   Autre thread : 100% accompli, terminé.");
    }

    static void Main()
    {
        evenementTermine = new AutoResetEvent(false);
```

```
        Console.WriteLine("Main: démarrage de l'autre thread...");
        Thread t = new Thread(AutreThread);
        t.Start();

        Console.WriteLine("Main: tâche 1/2 : 1 seconde ...");
        Thread.Sleep(1000);
        evenementTermine.Set(); // -> état signalé

        Console.WriteLine("Main: tâche 2/2 : 2 secondes ...");
        Thread.Sleep(2000);
        evenementTermine.Set(); // -> état signalé

        Console.WriteLine("Main: fin des tâches.");
    }
}
```

Verrou d'exclusion mutuelle

Un verrou d'exclusion mutuelle (mutex en abrégé) permet de donner un accès exclusif pour une ressource à un seul thread à la fois.

Cette classe dérive de la classe `WaitHandle` décrite avant.

Pour obtenir l'accès, il faut appeler une méthode wait (WaitOne par exemple), c'est à dire attendre l'état signalé. La méthode suivante permet de libérer le verrou en positionnant l'état signalé :

```
public void ReleaseMutex(); // -> État signalé
```

Comme pour une instance de la classe `AutoResetEvent`, le retour à l'état non-signalé est automatique, quand un thread sort de la fonction d'attente.

Cette classe possède le constructeur suivant :

```
public Mutex(bool initiallyOwned,
             string name, ref Boolean createdNew,
             System.Security.AccessControl.MutexSecurity mutexSecurity);
```

Tous les paramètres sont optionnels :

- `initiallyOwned` : `true` si le mutex est dans l'état non-signalé et appartient au thread appelant, `false` sinon (état signalé).
- Les autres paramètres servent à la synchronisation inter-processus et sont décrits dans la section "Synchronisation inter-processus" précédente.

Cet objet est dédié à la synchronisation inter-processus. Pour de l'exclusion mutuelle entre threads du même processus, du point de vue des performances, il est préférable d'utiliser un moniteur, ou l'instruction lock.

Processus

Pour chaque application lancée, le système d'exploitation crée un nouveau processus gérant l'état de l'application : mémoire (variables, données), code en cours d'exécution (threads), variables d'environnement,

ressources allouées (fichiers ouverts, sockets connectées, ...).

Chaque processus possède trois flux de communication :

- le flux d'entrée (généralement associé à l'entrée standard « stdin ») permet au processus de recevoir des données de l'utilisateur, ou du processus appelant,
- le flux de sortie (généralement associé à la sortie standard « stdout ») permet au processus d'afficher sous forme textuelle ses résultats à l'utilisateur, ou de les transmettre au processus appelant,
- le flux d'erreur (généralement associé à l'erreur standard « stderr ») permet au processus de notifier les messages erreurs à l'utilisateur, ou au processus appelant.

Entrées-sorties

Les fonctions d'entrées-sorties utilisent l'espace de nom `System.IO`.

La classe `Stream`

La classe abstraite `Stream` possède des méthodes permettant à la fois de lire et d'écrire. Cependant, l'écriture ou la lecture peut ne pas être autorisée (fichier en lecture seule, ...).

Propriétés

La classe `Stream` possède les propriétés suivantes :

`bool CanRead`
: (lecture seule) Cette propriété vaut `true` quand la lecture est possible.

`bool CanWrite`
: (lecture seule) Cette propriété vaut `true` quand l'écriture est possible.

`bool CanSeek`
: (lecture seule) Cette propriété vaut `true` quand le positionnement dans le flux est possible (méthode `Seek` ou propriété `Position`).

`bool CanTimeout`
: (lecture seule) Cette propriété vaut `true` quand le flux peut expirer (fin de connexion pour une socket, ...).

`long Length`
: (lecture seule) Longueur du flux en nombre d'octets.

`long Position`
: Position courante dans le flux en nombre d'octets depuis le début.

`int ReadTimeout`
: Temps imparti pour la méthode `Read`, en millisecondes. L'accès à cette propriété peut déclencher le lancement d'une exception de type `InvalidOperationException` si la fonctionnalité n'est pas supportée.

`int WriteTimeout`
: Temps imparti pour la méthode `Write`, en millisecondes. L'accès à cette propriété peut déclencher le lancement d'une exception de type `InvalidOperationException` si la fonctionnalité n'est pas supportée.

Méthodes

Les méthodes de la classe `Stream` sont les suivantes :

`void Close()`
: Ferme le flux.

`void Dispose()`
: Libère les ressources occupées par le flux.

`void Flush()`
: Cette méthode vide les buffers d'écriture vers le support associé (fichier, socket, ...).

`int ReadByte()`
: Cette méthode lit un octet et retourne sa valeur, ou -1 en cas d'erreur.

`void WriteByte(byte value)`
: Cette méthode écrit un octet.

`int Read(byte[] buffer, int offset, int count)`
: Cette méthode lit `count` octets dans le buffer spécifié, à partir de l'offset `offset` dans le tableau `buffer`. Elle retourne le nombre d'octets effectivement lus.

`void Write(byte[] buffer, int offset, int count)`
: Cette méthode écrit `count` octets du tableau `buffer` dont le premier octet est situé à l'offset `offset`.

`void Seek(long offset, SeekOrigin origin)`
: Déplace le pointeur d'écriture/lecture de offset octets depuis l'origine indiquée:
- `SeekOrigin.Begin` : depuis le début du flux,
- `SeekOrigin.End` : à partir de la fin du flux,
- `SeekOrigin.Current` : à partir de la position courante dans le flux.

`void SetLength(long value)`
: Cette méthode modifie la longueur totale du flux (troncature ou remplissage avec des octets nuls).

Méthodes asynchrones

Les méthodes `Read` et `Write` utilisant un tableau d'octet existent également en version asynchrone :

```
IAsyncResult BeginRead(byte[] buffer, int offset, int count, AsyncCallback, object).
int EndRead(IAsyncResult result).
IAsyncResult BeginWrite(byte[] buffer, int offset, int count, AsyncCallback, object).
void EndWrite(IAsyncResult result).
```

Membres statiques

La classe `Stream` possède également deux membres statiques :

- La constante `Null` est un flux dont la lecture ou l'écriture ne produit aucun effet.
- La méthode statique `Synchronized` retourne une version synchronisée du flux passé en paramètre.

Fonctions asynchrones

Les fonctions dont le retour peut prendre du temps existent également en version asynchrone. La méthode appelée est alors celle dont le nom commence par `Begin`. Elle demande au pool de threads standard d'exécuter l'opération. Une fois l'opération terminée, le delegate passé à la fonction `Begin` doit appeler la fonction `End` correspondante pour récupérer le résultat.

Exemple

Cet exemple utilise la version asynchrone de la méthode `Read` de la classe `Stream`.

```
class LectureAsync
{
    private byte[] buffer = new byte[4000];

    public void commencerLecture(Stream s)
    {
        // Commencer la lecture
        IAsyncResult iar = s.BeginRead(buffer, 0, buffer.length, finLecture, s);
        // et retour immmédiat
        // La méthode finLecture transmise en 4ème paramètre sera appelée
        // quand la lecture du tableau d'octets sera terminée.
    }

    public void finLecture(IAsyncResult result) //AsyncCallback
    {
        // Stream : le dernier argument transmis à BeginRead
        Stream s=(Stream)result.AsyncState;
        // récupérer le nombre d'octets lus
        int nb_octets = s.EndRead(result);
    }
}
```

Flux de fichier

La classe `FileStream` dérive de la classe `Stream`. Elle possède donc les mêmes méthodes.

La classe `FileStream` possède les constructeurs suivants :

```
public FileStream(string path, System.IO.FileMode mode,
  [ System.IO.FileAccess access,
  [ System.IO.FileShare share,
  [ int bufferSize,
  [ System.IO.FileOptions options ] ] ] ] );

public FileStream(string path, System.IO.FileMode mode,
  [ System.IO.FileAccess access,
  [ System.IO.FileShare share,
  [ int bufferSize,
  [ bool useAsync ] ] ] ] );

public FileStream(string path, System.IO.FileMode mode,
  System.Security.AccessControl.FileSystemRights rights,
  System.IO.FileShare share,
  int bufferSize, System.IO.FileOptions options,
  [ System.Security.AccessControl.FileSecurity fileSecurity ] );
```

Les crochets indiquent les paramètres optionnels.

La sérialisation

La sérialisation est un procédé d'entrée-sortie permettant de sauvegarder et recharger l'état d'un objet. Cette fonctionnalité permet de faire abstraction du format de fichier utilisé. Celui-ci dépend de l'outil de sérialisation utilisé.

L'état d'un objet correspond à l'ensemble des valeurs de ses champs. Les propriétés sont calculées en fonction des champs, et le code des méthodes ne change pas au cours de l'exécution.

Attributs

L'attribut `System.SerializableAttribute` marque les classes dont les instances peuvent être sérialisées. Si l'attribut est absent pour une classe, la sérialisation de ses instances provoquera une exception.

Exemple :

```
[Serializable]
class Facture
{
    public string Client;
    public double TotalHT;
    public double TotalTTC;
}
```

L'attribut `System.NonSerializedAttribute` marque les champs qu'il ne faut pas enregistrer. C'est le cas des champs dont on peut retrouver la valeur par calcul, par exemple.

Exemple :

```
[Serializable]
class Facture
{
    public string Client;
    public double TotalHT;

    [NonSerialized]
    public double TotalTTC; // = TotalHT * (1 + taux_TVA/100)
}
```

Quand la classe évolue, de nouveaux champs sont ajoutés à la classe, d'autres sont retirés. L'attribut `System.Runtime.Serialization.OptionalFieldAttribute` marque les champs optionnels lors de la désérialisation (lecture de l'objet). Il est donc possible de marquer les nouveaux champs comme optionnels, et garder les anciens champs (marqués optionnels également) pour garder la compatibilité avec les anciens fichiers.

Exemple :

```
[Serializable]
class Facture
{
    public string Client;
    public double TotalHT;

    [NonSerialized]
    public double TotalTTC;

    [OptionalField]
```

```
    public string AdresseLivraison; // Nouveau champ
}
```

Dans cet exemple, la classe Facture permettra de lire des fichiers d'objets `Facture` contenant le champ `AdresseLivraison` ou non.

La sérialisation

Le format de sérialisation dépend de la classe utilisée pour sérialiser les objets.

La classe `System.Runtime.Serialization.Formatter`

Cette classe abstraite définit les méthodes suivantes :

```
void Serialize(
    System.IO.Stream serializationStream,
    object graph);
```

Cette méthode enregistre l'objet `graph` dans le flux d'entrée-sortie spécifié.

```
object Deserialize(
    System.IO.Stream serializationStream);
```

Cette méthode retourne l'objet lu depuis le flux d'entrée-sortie spécifié.

Les classes dérivées définissent un format concret de sérialisation :

- La classe System.Runtime.Serialization.Formatters.Binary.BinaryFormatter permet de sérialiser dans un format binaire,
- La classe System.Runtime.Serialization.Formatters.Soap.SoapFormatter permet de sérialiser au format SOAP.

Une classe dérivée de `System.Runtime.Serialization.Formatter` sérialise les attributs et les évènements d'un objet, quel que soit l'accès associé (public, protégé, privé, ou par défaut).

Format XML

La classe `System.Xml.Serialization.XmlSerializer` permet de sérialiser au format XML. Elle ne dérive pas de la classe `System.Runtime.Serialization.Formatter`, et possède les méthodes suivantes :

```
void Serialize(
    System.IO.Stream stream,
    object o);
void Serialize(
    System.IO.TextWriter textWriter,
    object o);
void Serialize(
    System.Xml.XmlWriter xmlWriter,
    object o);
```

Ces méthodes enregistrent l'objet `o` dans le flux d'entrée-sortie spécifié.

```
object Deserialize(
    System.IO.Stream stream);
object Deserialize(
    System.IO.TextReader textReader);
object Deserialize(
    System.Xml.XmlReader xmlReader);
```

Ces méthodes retournent l'objet lu depuis le flux d'entrée-sortie spécifié.

Pour le format XML, les attributs `Serializable` et `NonSerialized` sont ignorés :

- Toute classe est sérialisable en XML,
- L'attribut `System.Xml.Serialization.XmlIgnoreAttribute` marque les champs à ignorer lors de la sérialisation.

La classe `System.Xml.Serialization.XmlSerializer` sérialise les attributs et les propriétés en lecture/écriture publics d'un objet, si la valeur n'est pas nulle. Donc la sérialisation en XML ignore :

- les attributs et propriétés retournant une valeur nulle,
- les attributs et propriétés protégés ou privés,
- les propriétés en lecture seule (impossible de les désérialiser),
- les propriétés en écriture seule (impossible de les sérialiser).

Pour en savoir plus :

- (anglais) XML Serialization in the .NET Framework (http://msdn2.microsoft.com/en-us/library/ms950721.aspx) [archive]

Sérialisation personnalisée

Il est possible de personnaliser la manière de sérialiser un objet en implémentant l'interface `System.Runtime.Serialization.ISerializable`.

Cette interface n'a qu'une méthode, invoquée lors de la sérialisation :

```
void GetObjectData (
    SerializationInfo info,
    StreamingContext context )
```

Mais la classe doit également comporter le constructeur suivant, invoqué lors de la désérialisation :

```
protected NomDeClasse(SerializationInfo info, StreamingContext context)
```

L'objet de type SerializationInfo permet la sérialisation et la désérialisation de l'objet. Chaque valeur sauvegardée est associé à un nom unique. Cet objet possède les méthodes suivantes :

```
public void AddValue(string name, T value) // tout type de valeur
public object GetValue(string name, Type type)
public T GetType(string name) // GetByte, GetChar, GetInt16, GetDecimal, GetDateTime, ...
```

L'implémentation de la méthode `GetObjectData` fait appel à la méthode `AddValue` de l'objet `info` pour

ajouter une valeur à sauvegarder. Le constructeur utilise la méthode `GetValue` ou les méthodes `Get`*`Type`* pour retrouver la valeur sauvegardée.

Exemple :

```
[Serializable]
public class Personne : ISerializable
{
    private string nom;
    private int age;

    public Personne() { }

    protected Personne(SerializationInfo info, StreamingContext context)
    {
        if (info == null)
            throw new System.ArgumentNullException("info");

        nom = (string)info.GetValue("Nom", typeof(string));
        age = (int)info.GetValue("Age", typeof(int));
    }

    public virtual void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        if (info == null)
            throw new System.ArgumentNullException("info");

        info.AddValue("Nom", nom);
        info.AddValue("Age", age);
    }
}
```

Les fichiers

Un fichier permet de stocker des données sur un support physique de stockage (disque dur, disquette, clé USB, ...). Les fichiers sont regroupés dans des répertoires.

L'espace de nom `System.IO` contient les classes permettant de gérer les fichiers et les répertoires.

Les classes `File` et `Directory`

Ces classes possèdent les méthodes de gestion des fichiers et des répertoires (respectivement). Elles sont toutes statiques.

Ces deux classes possèdent plusieurs méthodes communes.

Tester si un fichier ou un répertoire existe

La méthode `Exists` permet de tester si un chemin désigne un fichier ou un répertoire existant :

```
public static bool Exists(string path);
```

Exemple :

```
Console.WriteLine(
   File.Exists( "C:\monfichier.txt" )
       ? "Le fichier existe."
       : "Le fichier n'existe pas"
);
Console.WriteLine(
   Directory.Exists( "C:\WINDOWS" )
       ? "Le répertoire existe."
       : "Le répertoire n'existe pas"
);
```

Si la méthode appelée est celle de la classe File et que le chemin désigne un répertoire plutôt qu'un fichier (ou *vice versa*), cette méthode retournera également `false`.

Déplacer un fichier ou un répertoire

La méthode `Move` permet de déplacer un fichier ou un répertoire dans un autre répertoire, tout en le renommant :

```
public static void Move(string sourceName, string destName);
```

Exemple :

```
File.Move("C:\monfichier.txt", "C:\Documents\essai.txt");
```

Copier un fichier

La méthode `Copy` de la classe `File` permet de copier le contenu d'un fichier dans un autre :

```
public static void Copy(string sourceFileName,
    string destFileName[, bool overwrite]);
```

Le paramètre `overwrite` de type booléen indique si le fichier destination doit être écrasé s'il existe déjà.

Suppression

La méthode `Delete` permet de supprimer un fichier ou un répertoire :

```
public static void Delete(string path);
```

Informations

Les méthodes suivantes donnent ou modifient les informations sur le fichier ou le répertoire donné en paramètre.

Dates

```
public static DateTime GetCreationTime(string path);
```

```
public static DateTime GetCreationTimeUtc(string path);
```

Ces méthodes retournent respectivement :

- la date et l'heure de création (heure locale)
- la date et l'heure de création (UTC)

```
public static DateTime GetLastAccessTime(string path);
public static DateTime GetLastAccessTimeUtc(string path);
```

Ces méthodes retournent respectivement :

- la date et l'heure du dernier accès (heure locale)
- la date et l'heure du dernier accès (UTC)

```
public static DateTime GetLastWriteTime(string path);
public static DateTime GetLastWriteTimeUtc(string path);
```

Ces méthodes retournent respectivement :

- la date et l'heure de modification (heure locale)
- la date et l'heure de modification (UTC)

```
public static void SetCreationTime(string path, DateTime creationTime);
public static void SetCreationTimeUtc(string path, DateTime creationTimeUtc);
```

Ces méthodes modifient respectivement :

- la date et l'heure de création (heure locale)
- la date et l'heure de création (UTC)

```
public static void SetLastAccessTime(string path, DateTime lastAccessTime);
public static void SetLastAccessTimeUtc(string path, DateTime lastAccessTimeUtc);
```

Ces méthodes modifient respectivement :

- la date et l'heure du dernier accès (heure locale)
- la date et l'heure du dernier accès (UTC)

```
public static void SetLastWriteTime(string path, DateTime lastWriteTime);
public static void SetLastWriteTimeUtc(string path, DateTime lastWriteTimeUtc);
```

Ces méthodes modifient respectivement :

- la date et l'heure de modification (heure locale)
- la date et l'heure de modification (UTC)

Contrôle d'accès

```
// classe File
public static FileSecurity GetAccessControl(string path,
    [ AccessControlSections includeSections ] );
```

Cette méthode retourne les informations de sécurité d'un fichier.

```
// classe Directory
public static DirectorySecurity GetAccessControl(string path,
    [ AccessControlSections includeSections ] );
```

Cette méthode retourne les informations de sécurité d'un répertoire.

```
// classe File
public static void SetAccessControl(string path,
    FileSecurity fileSecurity);
```

Cette méthode modifie les informations de sécurité d'un fichier.

```
// classe Directory
public static void SetAccessControl(string path,
    DirectorySecurity directorySecurity);
```

Cette méthode modifie les informations de sécurité d'un répertoire.

Les classes `FileSecurity`, `DirectorySecurity` et `AccessControlSections` sont définies dans l'espace de nom `System.Security.AccessControl`.

Attributs d'un fichier

La classe `File` possède deux méthodes pour la gestion des attributs de fichiers :

```
public static FileAttributes GetAttributes(string path);
```

Cette méthode retourne les attributs du fichier.

```
public static void SetAttributes(string path, FileAttributes fileAttributes);
```

Cette méthode modifie les attributs du fichier.

L'énumération `FileAttributes` contient les éléments suivants :

- `Directory` : le chemin désigne un répertoire.
- `Normal` : le fichier est normal.
- `Archive` : le fichier peut être archivé.
- `Hidden` : le fichier est caché.
- `ReadOnly` : le fichier est en lecture seule.
- `System` : le fichier est un fichier système.
- `Compressed` : le fichier est compressé (NTFS).
- `Encrypted` : le fichier est crypté (NTFS).

- `Device`
- `NotContentIndexed`
- `Offline`
- `ReparsePoint`
- `SparseFile`
- `Temporary` : le fichier est temporaire.

Répertoire courant

Les deux méthodes suivants sont définies dans la classe `Directory`. Le répertoire courant désigne le répertoire à partir duquel sont définis les fichiers et répertoires désignés par un chemin relatif.

```
public static string GetCurrentDirectory();
```

Cette méthode retourne le chemin du répertoire courant de l'application.

```
public static void SetCurrentDirectory(string path);
```

Cette méthode défini le chemin du nouveau répertoire courant de l'application.

Les classes `FileInfo` et `DirectoryInfo`

Une instance de la classe `FileInfo` (resp. `DirectoryInfo`) représente un fichier (resp. un répertoire), c'est à dire correspond à un chemin donné.

Ces classes permettent les mêmes fonctionnalités que celles vues précédemment, en utilisant des méthodes d'instances. Il n'y a qu'un seul constructeur qui prend comme paramètre le chemin du fichier ou répertoire.

Exemple :

```
FileInfo file = new FileInfo( @"C:\Documents\Projets\Images\Photo252.jpg" );
// Rappel : une chaîne verbatim ( @"..." ) évite de doubler les anti-slashs

DirectoryInfo dir = file.Directory; // -> C:\Documents\Projets\Images\
dir = dir.Parent;                   // -> C:\Documents\Projets\
dir = dir.Root;                     // -> C:\
```

Propriétés

Les deux classes ont les propriétés communes suivantes :

```
// Nom et extension :
public string         Name { virtual get; }
public string         Extension { get; }
public string         FullName { virtual get; }

// Test d'existence, et attributs
public bool           Exists { virtual get; }
public FileAttributes Attributes { get; set; }

// Dates et heures (création, accès et écriture), locales et UTC :
```

```
public DateTime       CreationTime { get; set; }
public DateTime       CreationTimeUtc { get; set; }
public DateTime       LastAccessTime { get; set; }
public DateTime       LastAccessTimeUtc { get; set; }
public DateTime       LastWriteTime { get; set; }
public DateTime       LastWriteTimeUtc { get; set; }
```

La classe `FileInfo` a les propriétés spécifiques suivantes :

```
// Test lecture seule et taille du fichier (octets) :
public bool          IsReadOnly { get; set; }
public long          Length { get; }

// Répertoire parent (objet, chemin) :
public DirectoryInfo Directory { get; }
public string        DirectoryName { get; }
```

La classe `DirectoryInfo` a les propriétés spécifiques suivantes :

```
// Répertoires parent et racine :
public DirectoryInfo Parent { get; }
public DirectoryInfo Root { get; }
```

Méthodes

Programmation réseau

L'utilisation d'un réseau informatique dans une application permet de communiquer avec d'autres applications exécutées sur d'autres machines, voire la même machine.

L'espace de noms System.Net regroupe les classes concernant la programmation réseau en général (adresse IP, ...).

L'espace de noms System.Net.Sockets regroupe les classes concernant l'utilisation de sockets.

Utilisation de sockets

L'espace de noms System.Net.Sockets possède une classe Socket qui est une abstraction de bas niveau d'une socket en général.

Les classes TcpClient et TcpListener gèrent une socket utilisant le protocole TCP.

La classe UdpClient gère une socket utilisant le protocole UDP.

Exécution distante

Une application .NET peut être un service distant et peut s'exécuter comme :

Serveur RPC

Serveur Remoting

À l'aide de la librairie `System.Runtime.Remoting` à placer dans l'application servante et cliente, celles-ci peuvent partager une classe par Marshaling suivant un canal Http, Ipc ou Tcp.

Le remoting utilise les échanges RPC.

Dans cette solution :

1. INTERFACE : IRemoteMath est une librairie de classe
2. SERVER : RemoteServer est une application console
3. CLIENT : RemoteClient est une application console

L'interface partagée

Une interface IOperations comprenant un prototype d'addition.

```
namespace IRemoteMath
{
    public interface IOperations
    {
        int Addition(int a, int b);
    }
}
```

Le serveur

Le serveur est l'application qui distribue ses services. Pour qu'elle déserve à distance, il lui faut :

- Dans ses références :
    1. System.Runtime.Remoting
    2. Le namespace d'IRemoteMath
- Implémenter IOperations

Implémenter IOperations

La classe à servir doit faire partie du service distributeur ici RemoteServer

```
namespace RemoteServer
{
    public class RemoteOperations : MarshalByRefObject, IRemoteMath.IOperations
    {
        // l'objet aura une durée de vie illimitée
        public override object InitializeLifetimeService()
        {
            return null;
        }

        // cette méthode sera servie
```

```
        public int Addition(int a, int b)
        {
            Console.WriteLine(String.Format("> Addition() : a={0}, b={1}", a, b));
            return a + b;
        }
    }
}
```

MarshalByRefObject signifie que RemoteOperations fera l'objet d'un marshaling.

Configurer RemoteMain

Pour écouter les appels, le serveur doit créer un canal d'écoute sur un port et enregistrer le service à distribuer.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace RemoteServer
{
    class RemoteMain
    {
        [STAThread]
        static void Main(string[] args)
        {
            try
            {
                // Création du canal sur le port 1050
                TcpChannel channel = new TcpChannel(1050);
                // Enregistrement du canal
                ChannelServices.RegisterChannel(channel);
                // Distribution de l'objet en mode singleton
                RemotingConfiguration.RegisterWellKnownServiceType(
                    typeof(RemoteOperations),
                    "RemoteOperations",
                    WellKnownObjectMode.Singleton);

                Console.WriteLine("Serveur démarré");
                Console.ReadLine();
            }
            catch
            {
                Console.WriteLine("Erreur au démarrage");
                Console.ReadLine();
            }
        }
    }
}
```

1. [STAThread] au point d'entrée instruit le main comme "appartement" pour cloisonner les traitements au partage de ses ressources.
2. RemotingConfiguration.RegisterWellKnownServiceType enregistre RemoteOperations dans ses services
3. Le canal est ici en TCP

Le client

Le client appelle les services. Pour qu'elle soit servie, il lui faut :

- Dans ses références :
    1. System.Runtime.Remoting
    2. Le namespace d'IRemoteMath

```
using System;
using System.Text;

// Remoting
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace RemoteClient
{
    class RemoteTestClient
    {
        // préparation de l'objet distribué
        private IRemoteMath.IOperations remoteOperation;

        public RemoteTestClient()
        {
            try
            {
                // init channel - facultatif
                // TcpChannel channel = new TcpChannel();
                // ChannelServices.RegisterChannel(channel);
                // init IOperations < RemoteServer.RemoteOperations
                this.remoteOperation = (IRemoteMath.IOperations)Activator.GetObject(
                    typeof(IRemoteMath.IOperations),
                    "tcp://localhost:1050/RemoteOperations");
            }
            catch {
                Console.WriteLine("Erreur de connexion");
            }
        }

        public void RemoteAddition(int a, int b)
        {
            try
            {
                if (this.remoteOperation != null)
                {
                    Console.WriteLine("Résultat : " + this.remoteOperation.Addition(a, b)
                }
            }
            catch
            {
                Console.WriteLine("Erreur à l'appel");
            }
        }

        [STAThread]
        static void Main()
        {
            RemoteTestClient Client = new RemoteTestClient();
            Client.RemoteAddition(15, 20);

            System.Threading.Thread.Sleep(5000);
        }
    }
```

```
}
```

1. [STAThread] indique qu'on est dans le même type de cloisonnement des threads que pour server.
2. remoteOperation dépile la classe reçu par TCP permettant l'utilisation de ses propriétés.

Application

1. Pour effectuer le test, il faut compiler les trois projets IRemoteMath,RemoteServer,RemoteClient
2. Executer RemoteServer.exe avant RemoteClient.exe

Le résultat du client devrait être :

```
Résultat : 35
```

Le message du serveur devrait être :

```
> Addition() : a=15, b=20
```

Serveur de pipes

Serveur de queue

Bibliographie et liens

Liens internes

- Programmation .Net

Liens externes

- (français) La documentation MSDN de Visual C# (http://msdn2.microsoft.com/fr-fr/library/kx37x362.aspx) [archive]
- (français) Centre de Développement C# - Site MSDN (http://msdn.microsoft.com/fr-fr/vcsharp/default.aspx) [archive]
- (français) Guide de programmation C# (http://msdn2.microsoft.com/fr-fr/library/67ef8sbd(VS.80).aspx) [archive]
- (français) Un blog sur .Net (http://www.labo-dotnet.com/) [archive]
- (anglais) C# Language Specification 4th edition (http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf) [archive]
- (anglais) Encyclopédie collaborative sur C# et .Net (http://en.csharp-online.net/) [archive]
- (anglais) Page du standard ISO (http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=42926) [archive]
- (anglais) Page du standard ECMA (http://www.ecma-international.org/publications/standards/Ecma-334.htm) [archive]
- (anglais) C# Help : aide aux développeurs (http://www.csharphelp.com) [archive]
- (anglais) Vidéos d'apprentissage de C#, Visual Studio express, et .Net (http://msdn.microsoft.com/vstudio

/express/visualcsharp/learning/) [archive]
- (anglais) Vidéos d'apprentissage de C#, Visual Studio, et .Net (http://www.learnvisualstudio.net) [archive]

Livres

- français *Pierre-Yves Saumont, Antoine Mirecourt* - **Introduction à C#** - Éditions Eyrolles - 2001 - *292 pages* - ISBN 2-7464-0301-3
- français *Gérard Leblanc* - **C# et .NET** - Éditions Eyrolles - 2002 - *710 pages* - ISBN 2-212-11066-9
- français *Gérard Leblanc* - **C# et .NET Version 2** - Éditions Eyrolles - 2006 - ISBN 2-212-11778-7
- français *Donis Marshall, Christine Eberhardt, Philippe Beaudran, Dorothée Sittler* - **Visual C# 2005** - Collection *Microsoft Press*, Éditions Dunod - 2006 - ISBN 2-10-049942-4
- français *Mickey Williams* - **Manuel de référence Microsoft Visual C#** - Collection *Langages et Programmation*, Éditions Dunod - 2002 - *784 pages* - ISBN 2-10-006659-5
- français *Tom Archer* - **Formation à C#** - Collection *Formation à...*, Éditions Dunod - 2001 - *400 pages* - ISBN 2-84082-864-2
- anglais *Adrian Kingsley-Hughes, Kathie Kingsley-Hughes* - **C# 2005 Programmer's Reference** - Éditions Wrox - 2006 - ISBN 0-470-04641-4
- anglais *Simon Robinson, Christian Nagel, Karli Watson, Jay Glynn, Morgan Skinner, Bill Evjen* - **Professional C#** *(3ème édition)* - Éditions Wrox - 2004 - ISBN 0-7645-5759-9
- anglais *Ben Albahari, Peter Drayton, Brad Merrill* - **C# Essentials** *(2ème édition)* - Éditions O'Reilly Media - 2002 - ISBN 0-5960-0315-3
- anglais *Andrew Krowczyk, Zach Greenvoss, Christian Nagel, Ashish Banerjee, Thiru Thangarathinam, Aravind Corera, Chris Peiris, Brad Maiani* - **Professional C# Web Services: Building .NET Web Services with ASP.NET and .NET Remoting** - Éditions Wrox Press - 2001 - *550 pages* - ISBN 1-8610-0439-7
- anglais *Matthew MacDonald* - **Pro .NET 2.0 Windows Forms and Custom Controls in C#** - Éditions Apress - 2005 - *1080 pages* - ISBN 1-59059-439-8

Index des mots-clés du langage

Un mot-clé est un nom ayant une signification spéciale, et qui ne peut pas être utilisé comme identificateur de classe, de méthode ou de variable.

Voici la liste complète des mots clés reconnus par le langage C#.

A

- abstract
- add
- alias
- as

B

- base
- bool

F

- false
- finally
- fixed
- float
- for
- foreach

G

O

- object
- operator
- out
- override

P

- params
- partial

T

- this
- throw
- true
- try
- typeof

U

- uint

- break
- byte

C

- case
- catch
- char
- checked
- class
- const
- continue

D

- decimal
- default
- delegate
- do
- double
- dynamic

E

- else
- enum
- event
- explicit
- extern

- get
- global
- goto

I

- if
- implicit
- in
- int
- interface
- internal
- is

L

- lock
- long

N

- namespace
- new
- null

- private
- protected
- public

R

- readonly
- ref
- remove
- return

S

- sbyte
- sealed
- set
- short
- sizeof
- stackalloc
- static
- string
- struct
- switch

- ulong
- unchecked
- unsafe
- ushort
- using

V

- value
- var
- virtual
- void

W

- where
- while

Y

- yield

abstract

Ce mot-clé permet de déclarer une classe ou une méthode abstraite (voir classe abstraite).

add

Accesseur d'ajout d'une méthode à un `event` (voir fonctionnement interne d'un event).

alias

Alias d'assemblage pour créer de nouvelles racines d'espace de nom (voir Alias d'assemblages).

as

Opérateur de conversion (voir l'opérateur as).

base

Référence à la classe de base (voir héritage de classes).

bool

Le type booléen (voir Les types de bases).

break

Interruption de l'exécution d'une boucle ou d'un test de cas (voir les structures de contrôle).

byte

Le type octet (8 bits) non signé (voir Les types de bases).

case

Déclaration d'un cas à tester (voir tests de plusieurs cas, default, switch).

catch

Début du bloc exécuté quand une exception est attrapée (voir attraper une exception).

char

Le type caractère unicode (16 bits) (voir Les types de bases).

checked

Évaluation de l'expression donnée dans un contexte vérifiant le débordement, et lançant une exception de type `System.OverflowException` dans ce cas (voir Vérification du débordement et unchecked).

class

Déclaration d'une nouvelle classe d'objets (voir les classes).

const

Déclaration d'une constante (voir Les constantes).

continue

Poursuite immédiate de l'exécution d'une boucle sans exécuter la fin du bloc d'instruction (voir continuer une boucle).

decimal

Le type nombre décimal à virgule flottante, grande précision (voir Les types de bases).

default

- Déclaration du cas par défaut (voir tests de plusieurs cas, case, switch),
- Obtenir la valeur par défaut d'un type (voir Types de base : Valeur par défaut, Types génériques : Opérateur default).

delegate

Déclaration d'une référence à une méthode de signature spécifique (voir Les délégués).

do

Cette instruction permet d'exécuter plusieurs fois une instruction (voir les boucles, while).

double

Le type nombre à virgule flottante, double précision (voir Les types de bases).

else

Cette instruction précède l'instruction à exécuter quand la condition est fausse (voir condition).

enum

Déclaration d'une nouvelle énumération (voir les énumérations).

event

Déclaration d'une référence de méthode pour la gestion d'évènements (voir Les évènements).

explicit

Déclarer un opérateur de conversion explicite (voir Explicite/Implicite, implicit).

extern

Déclarer une fonction définie dans une DLL (voir Appel de plateforme).

false

Faux. L'une des deux valeurs possibles pour le type booléen (voir Les types de bases, true).

finally

Début du bloc de code exécuté à la fin d'un bloc `try`, quel que soit les exceptions éventuellement lancées (voir attraper une exception).

fixed

Éviter le déplacement en mémoire d'un tableau ou d'un objet par le ramasse-miettes (voir Éviter le déplacement par le ramasse-miettes).

float

Le type nombre à virgule flottante, simple précision (voir Les types de bases).

for

Cette instruction permet d'exécuter plusieurs fois une instruction (voir les boucles).

foreach

Cette instruction permet d'exécuter une instruction pour chacun des éléments d'un ensemble : tableau, liste, ... (voir les boucles, in).

get

Accesseur de lecture d'une propriété ou d'un indexeur (voir propriétés et indexeurs).

global

Racine par défaut des espaces de nom (voir Conflit de nom).

goto

Cette instruction poursuit l'exécution au cas indiqué (voir tests de plusieurs cas, switch).

if

Cette instruction permet d'exécuter une instruction si une condition est vraie (voir condition).

implicit

Déclarer un opérateur de conversion implicite (voir Explicite/Implicite, explicit).

in

Cette instruction permet de spécifier l'ensemble d'éléments (tableau, liste, ...) pour lequel une instruction sera exécutée pour chacun des éléments (voir les boucles, foreach).

int

Le type entier signé sur 32 bits (voir Les types de bases).

interface

Déclaration d'une nouvelle interface (voir les interfaces).

internal

Niveau de protection d'un membre de classe : accès possible au sein du même assemblage seulement (voir Niveaux de protection).

is

Test du type d'un objet (voir L'opérateur is).

lock

Outil de synchronisation entre threads permettant un accès exclusif à un objet pendant l'exécution du bloc de code associé (voir Synchonisation avec le mot clé `lock`).

long

Le type entier signé sur 64 bits (voir Les types de bases).

namespace

Déclaration d'un espace de nom (voir les espaces de noms).

new

- Création d'une nouvelle instance de classe (voir Instance d'une classe).
- Surcharge d'une méthode sans polymorphisme (voir Surcharge sans polymorphisme).

null

Référence nulle (voir La référence nulle).

object

Type d'objet à la base de tous les autres types (voir Les objets).

operator

Mot clé précédant l'opérateur surchargé dans une déclaration de méthode (voir Surcharge des opérateurs).

out

Mode de passage de paramètre à une méthode de type écriture seule. Un tel paramètre doit obligatoirement être modifié par la méthode appelée, et n'a pas besoin d'être initialisé avant l'appel à la méthode (voir

Paramètre out, ref).

override

Surcharge d'une méthode avec polymorphisme (voir Surcharge avec polymorphisme, virtual).

params

Précède la déclaration du dernier paramètre d'une méthode (de type tableau) pour que celui-ci recueille tous les paramètres supplémentaires sous la forme d'un tableau (voir Nombre variable de paramètres).

partial

Ce mot clé indique que le fichier source ne contient qu'une partie de la classe, la structure ou l'interface déclarée. (Voit type partiel).

private

Niveau de protection d'un membre de classe le plus restrictif : accès possible au sein de la classe seulement (voir Niveaux de protection).

protected

Niveau de protection d'un membre de classe : accès possible au sein de la classe ou de ses sous-classes, quelquesoit le niveau d'héritage (voir Niveaux de protection et Héritage).

public

Niveau de protection d'un membre de classe le moins restrictif : accès possible depuis n'importe quelle classe (voir Niveaux de protection).

readonly

Déclaration d'une variable en lecture seule, c'est à dire dont l'affectation ne peut s'effectuer qu'une seule fois (voir Variable en lecture seule).

ref

Mode de passage de paramètre à une méthode de type lecture et écriture (référence). Un tel paramètre peut être modifié par la méthode appelée, et doit être initialisé avant l'appel à la méthode (voir Paramètre out, out).

remove

Accesseur de retrait de méthode d'un event (voir fonctionnement interne d'un event).

return

Cette instruction spécifie ce que la fonction retourne (voir les fonctions).

sbyte

Le type octet (8 bits) signé (voir Les types de bases).

sealed

Une classe déclarée "Sealed" ne peut plus être héritée (voir Classe sans héritière).

set

Accesseur de modification d'une propriété ou d'un indexeur (voir propriétés et indexeurs).

short

Le type entier signé sur 16 bits (voir Les types de bases).

sizeof

Obtenir la taille du type ou de la variable spécifiée entre parenthèses. Le code utilisant cet opérateur doit être déclaré comme non vérifié (voir Taille d'une variable ou d'un type, unsafe).

stackalloc

Allocation sur la pile au lieu du tas. Ce mot-clé doit être utilisé dans un contexte de code non vérifié (voir Allocation sur la pile, unsafe).

static

Déclaration d'un membre statique d'une classe, ou d'une classe statique ne contenant que des membres statiques (voir Membres statiques de classe et membres d'instance).

string

Le type chaîne de caractères (voir Les types de bases).

struct

Déclaration d'une nouvelle structure de données (voir les structures).

switch

Cette instruction permet de tester la valeur d'une expression avec plusieurs cas (voir tests de plusieurs cas, case, default).

this

Référence à l'objet lui-même (voir héritage de classes).

throw

Cette instruction lance un exception (voir lancer une exception).

true

Vrai. L'une des deux valeurs possibles pour le type booléen (voir Les types de bases, false).

try

Début du bloc de code pour lequel les exceptions sont attrapées (voir attraper une exception).

typeof

Récupérer le type (`System.Type`) de la classe dont le nom est spécifié entre parenthèses (voir Types de base : Obtenir le type).

uint

Le type entier non signé sur 32 bits (voir Les types de bases).

ulong

Le type entier non signé sur 64 bits (voir Les types de bases).

unchecked

Évaluation de l'expression donnée dans un contexte ne vérifiant pas le débordement, et copiant le résultat même si celui-ci ne loge pas dans le type requis (voir Non vérification du débordement, checked).

unsafe

Déclarer du code non vérifié (voir Code non vérifié).

ushort

Le type entier non signé sur 16 bits (voir Les types de bases).

using

- Déclaration des espaces de nom utilisés (voir Utiliser les membres d'un espace de nom),
- Création d'un alias d'espace de nom (voir Alias d'espace de nom), ou de classe générique (voir Alias de type générique),
- Utilisation d'un objet `IDisposable` (voir Libérer des ressources).

value

Nom réservé au paramètre des accesseurs add, remove et set.

virtual

Déclaration d'une méthode dont la surcharge est avec polymorphisme (voir Surcharge avec polymorphisme, override).

void

Le type vide pour indiquer qu'une fonction ne retourne rien (voir Les fonctions).

where

Contraintes sur un type générique (voir Contraintes sur les types génériques).

while

Cette instruction permet d'exécuter plusieurs fois une instruction tant qu'une condition est vraie (voir les boucles, do).

yield

Cette instruction permet de créer une énumération (voir l'interface IEnumerable et le mot clé yield).

Glossaire

Ce chapitre donne une brève définition des termes employés en programmation orientée objet et leur application en C# en donnant des liens vers les chapitres concernés de ce livre.

A

Abstrait(e)

Une méthode abstraite possède une déclaration (signature) mais pas d'implémentation. La classe qui la déclare est donc obligatoirement abstraite également : aucune instance de cette classe ne peut être créée. La méthode devra être redéfinie dans une sous-classe qui fournira l'implémentation de la méthode.
Une classe abstraite peut ne pas avoir de méthode abstraite. Aucune instance de cette classe ne peut être créée : il faut instancier une de ses sous-classes concrète.

- Voir le chapitre Classe abstraite

Attribut

Un attribut est l'équivalent d'une variable déclarée dans une classe. Il possède donc un nom et un type. Il a également un niveau d'accès (public, protégé, privé) et peut également avoir une valeur par défaut.

- Voir Membres d'une classe du chapitre *Les classes*.

C

Classe

Une classe est un type de données permettant d'encapsuler des informations sur une entité et d'y associer des méthodes de traitement de ces informations.

- Voir le chapitre Les classes.

Commentaire

Un commentaire sert à documenter le fonctionnement global et détaillé d'une classe.

- Voir le chapitre Les commentaires.
- Voir le chapitre Documentation XML des classes.

Concret, Concrète

Opposé à abstrait(e).

Constante

Une constante est déclarée avec une valeur qui ne change pas.

- Voir Les constantes du chapitre *Les variables et les constantes*.

E

Énumération

Une énumération est un type qui est défini par la liste des valeurs possibles.

- Voir Énumération du chapitre *Structures et énumérations*.

Espace de noms

Un espace de noms permet de regrouper des classes dans un espace commun. Chaque classe utilisée doit alors être nommée en spécifiant l'espace de noms où elle est déclarée, à moins que la référence se fasse depuis le même espace de nom ou en utilisant le mot-clé `using`.

- Voir le chapitre Les espaces de noms.

Exception

Une exception est un objet particulier signalant une erreur. Une exception est lancée au moment où l'erreur se produit. Elle se propage en remontant la pile d'appel jusqu'à ce qu'un traitement approprié soit rencontré.

- Voir le chapitre Les exceptions.

F

Fonction

Une fonction possède un nom et une série de paramètre et retourne une valeur. Lorsqu'elle est membre d'une classe, elle est plutôt appelée méthode.

- Voir le chapitre Les fonctions.

H

Héritage

Une classe peut hériter des membres (méthodes, attributs, propriétés) d'une ou plusieurs classes de base. En C#, une classe ne peut hériter que d'une seule classe de base (par défaut de System.Object).

- Voir le chapitre Héritage de classes.

I

Indexeur

Un indexeur dans une classe est une propriété spéciale permettant la surcharge de l'opérateur [] permettant d'adresser un élément d'un ensemble. Il est donc en général surchargé dans les classes servant de conteneur d'éléments (tableaux, listes, ...).

- Voir Les indexeurs du chapitre *Propriétés et indexeurs*.

Instance

L'instance d'une classe est un objet alloué en mémoire. L'espace mémoire alloué est suffisant pour stocker chacun des attributs déclarés dans la classe et les classes dont elle hérite. La référence pointe vers cet espace alloué. Les méthodes de la classe, appelée pour cette instance, connaissent l'adresse de cet espace mémoire par l'utilisation explicite ou implicite de la référence this.

- Voir le chapitre Les objets.
- Voir Instance d'une classe du chapitre *Les classes*.

Interface

Une interface ne fait que décrire une liste de méthodes, sans implémentation. Le code de ces méthodes est fourni par les classes qui implémentent l'interface.

- Voir le chapitre Interfaces.

M

Membre

Il s'agit d'un attribut ou une méthode appartenant à la classe ou il est déclaré.

- Voir Membres d'une classe du chapitre *Les classes*.

Méthode

Une méthode est membre d'une classe. Il s'agit en général d'une fonction ou procédure. En C#, toutes les méthodes sont des fonctions, c'est-à-dire qu'elles retournent une valeur du type déclaré.

- Voir le chapitre Les fonctions.

O

Objet

Un objet est une instance d'une classe. En comparaison avec la programmation impérative, la classe est un type, tandis que l'objet est une variable de ce type.

- Voir le chapitre Les objets.
- Voir Instance d'une classe du chapitre *Les classes*.

Opérateur

Les expressions emploient des opérateurs arithmétiques, binaires, logiques de comparaisons, ...

- Voir le chapitre Les opérateurs.

Les classes peuvent surcharger les opérateurs afin que leur instances puissent être directement utilisées dans des expressions (par exemple, pour une classe de nombre complexe).

- Voir le chapitre Surcharge des opérateurs.

P

Polymorphisme

Le polymorphisme est lié à l'héritage entre classes. Il permet d'appeler la méthode redéfinie d'une sous-classe alors que l'on possède une référence à une instance de la classe de base.

- Voir le chapitre Héritage de classes.

Propriété

Une propriété est un couple de méthode (get et set) permettant respectivement de lire et d'écrire une valeur particulière de la classe. Cette propriété est utilisé comme un attribut classique d'une classe. Cela permet de contrôler par exemple la validité des modifications d'une valeur.

- Voir Les propriétés du chapitre *Propriétés et indexeurs*.

R

Redéfinition

Une méthode dans une classe peut être redéfinie dans une sous-classe (même nom, même signature) pour avoir un comportement différent. Dans le cas des méthodes abstraites d'une classe de base, la redéfinition permet de fournir une implémentation dans une sous-classe.

- Voir Redéfinition de méthodes et de propriétés du chapitre *Héritage de classes*.

S

Signature

La signature d'un membre d'une classe identifie ce membre. Deux membres d'une même classe ne peuvent avoir la même signature. En C#, la signature d'un attribut est composé de son nom uniquement. Celle d'une méthode est composée de son nom et de la liste des types de ses paramètres. Une sous-classe peut redéfinir une méthode de la classe de base en utilisant la même signature et type de retour que la méthode redéfinie.

Structure

Une structure contient plusieurs champs de différent types. Ce type de données permet de rassembler plusieurs informations à propos d'une même entité. En C#, les structures peuvent également avoir des méthodes comme les classes.

- Voir Structure du chapitre *Structures et énumérations*.

Surcharge

Une même classe peut comporter plusieurs méthodes ayant le même nom si celles-ci se distinguent par leur signature (nombre et types de leur paramètres).

- Voir Surcharge de méthode du chapitre *Les fonctions*.
- Voir le chapitre Surcharge des opérateurs.

T

Tableau

Un tableau contient plusieurs valeurs du même type.

- Voir le chapitre Les tableaux.

Programmation C sharp

Cette page est le livre d'or du livre Programmation C#.

N'hésitez pas à faire vos remarques ici, indiquez des fautes, proposez des améliorations, ...

Utilisez le modèle suivant pour noter le livre :

```
{{RemarqueNotée|note=de 0 à 10|titre=titre de la critique|votre remarque ici}}
```

Livre d'or

Ne couvre pas tout le C#

Note : ★☆☆☆☆	**Ne couvre pas tout le C#**
Le livre ne couvre que le C# 2.0	

bonne lecture

Note : ★★★★☆	**bonne lecture**
Je viens de télécharger cet eBook, c'est de la qualité - Zulul 29 août 2009 à 13:41 (CEST)	

REMERCIMMENT

MERCI BEAUCOUP A TOUS CEUX QUI NOUS AIDE A AVANCER QUE LES BÉNÉDICTIONS DU DIEU VIVANTS SOIT

Ce n'est pas un livre, c'est une bible !

Note : ★★★★☆	**Ce n'est pas un livre, c'est une bible !**
Le plus utile des WikiBooks à mon sens. Très complet, couvre tout le C# du débutant à la programmation de tous les jours. Proposition d'évolution : Les design patterns pour les nuls ! encore bravo !	

Merci et bravo!

Merci et bravo!

Comparaison avec d'autres langages

Beau travail :)

Il est intéressant de comparer ce langage avec Java, C++ mais aussi avec Objective-C. Delegate n'est pas une nouveauté c'est un emprunt comme d'autres concepts aussi.

Remerciements

Nous vous remercions infiniment d'avoir mis mis cette œuvre à la disposition des internautes.

cool

cool

Note : ★★★★☆	**cool**
Agréable à lire, ces concepts appliqué a des tp auraient été le bien-venus.	

Titre du commentaire

Remplacez ceci par le titre de votre commentaire

Note : Erreur d'expression : mot « remplacez » non reconnu.Erreur d'expression : mot « remplacez » non reconnu.Erreur d'expression : mot « remplacez » non reconnu.Erreur d'expression : mot « remplacez » non reconnu.Erreur d'expression : mot « remplacez » non reconnu.	**Remplacez ceci par le titre de votre commentaire**
Remplacez ceci par votre commentaire	

pas mal

pas mal

Note : ★★★★☆	**pas mal**
pas mal, mais nécessite plus d'exemples et de démonstrations pour permettre d'apprendre le c# de 0.	