Formation à C#

Support de cours

Résumé

Support d'accompagnement d'une formation à la programmation en C# faite en présentiel

Cyril Seguenot

2018

SOMMAIRE

1 Introduction

Le présent document sert de support pour une formation en présentiel. Il explique de façon synthétique les notions de base de la programmation en C#, et n'a pas vocation à remplacer un ouvrage de référence. En particulier, Winform, WPF, WCF, ADO.Net, et d'autres sujets vastes ne sont pas du tout abordés dans ce support. Par défaut, toutes les notions abordées sont valables depuis la version 3.0 de C#. Si certaines sont spécifiques à des versions plus récentes, ces dernières seront indiquées.

Les notions sont illustrées par des exemples de code, dont certains font appel à des notions qui ne sont abordées que plus loin dans le support. Il est donc tout à fait normal que vous ne compreniez pas immédiatement toutes les lignes de code. Ceci ne doit pas vous empêcher de continuer, car toutes les notions seront expliquées progressivement.

N'hésitez pas à annoter vous-même ce support tout au long de la formation, afin d'en faire le document le plus utile pour vous-même.

2 La plateforme .Net

Le langage C# (C Sharp) est un langage objet créé spécialement pour le framework Microsoft .NET. L'équipe qui a créé ce langage a été dirigée par *Anders Hejlsberg*, un informaticien danois qui avait également été à l'origine de la conception du langage *Delphi* pour la société Borland (évolution objet du langage Pascal).

2.1 .Net Framework

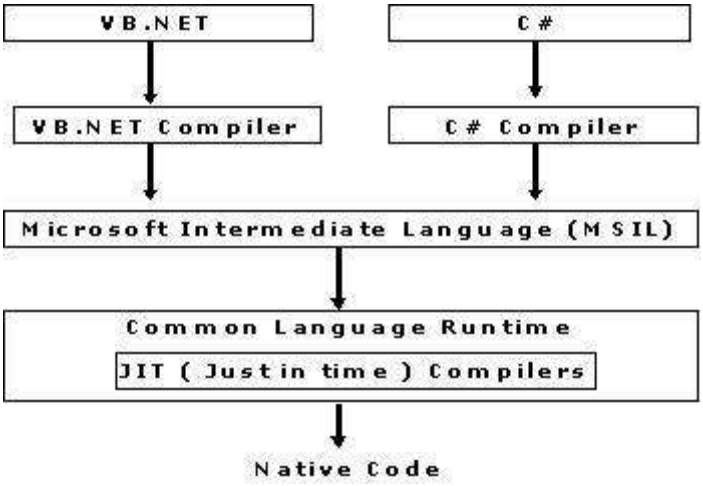
Le **.NET Framework** est constitué d'un environnement d'exécution (CLR Common Language Runtime) et d'une bibliothèque de classes, interfaces et types valeur (plusieurs milliers répartis dans 53 espaces de noms principaux dans .net 4.6). L'environnement d'exécution (CLR) de .NET est comparable à la machine virtuelle de Java (JVM), bien que plus évolué.

Les classes .Net peuvent être utilisées par tous les langages prenant en charge l'architecture .NET. Les langages .NET doivent satisfaire certaines spécifications : utiliser les mêmes types CTS (Common Type System), les compilateurs doivent générer un même code intermédiaire appelé MSIL (Microsoft Intermediate Language).

Le code source est compilé en MSIL dans un fichier .exe, et pris en charge par le runtime. Ce n'est qu'au moment de l'exécution, qu'il est compilé en code natif par le compilateur JIT (Just In Time) intégré au runtime.

Définir un langage .NET revient à fournir un compilateur qui peut générer du langage MSIL. Les spécifications .NET sont publiques (Common Language Specifications), et n'importe quel éditeur de logiciels peut donc concevoir un langage et un compilateur .NET. Plusieurs compilateurs sont actuellement disponibles : C++.NET (version Managée de C++), VB.NET, C#, J#, Delphi, IronPython...etc.

Remarque : la version .NET de C++ a la particularité de permettre à l'utilisateur de générer soit du code natif soit du code managé, c'est-à-dire contrôlé par le runtime.



Le runtime est un environnement d'exécution qui fournit des services aux programmes qui s'exécutent sous son contrôle (on parle de code « managé ») :

- Chargement/exécution/isolation des programmes
- Vérification des types
- Conversion du code intermédiaire (IL) en code natif
- Accès aux métadonnées (informations sur le code contenu dans les assemblages .NET)
- Vérification des accès mémoire (évite les accès en dehors de la zone allouée au programme)
- Gestion de la mémoire (Garbage Collector)
- Gestion des exceptions
- Adaptation aux caractéristiques nationales (langue, représentation des nombres)
- Compatibilité avec les DLL et modules COM qui sont en code natif (code non managé)

"En .NET, les langages ne sont guère plus que des interfaces syntaxiques vers les bibliothèques de classes." Formation à C#, Tom Archer, Microsoft Press.

On ne peut rien faire sans utiliser des types/classes fournis par le framework .NET puisque le code MSIL s'appuie également sur les types CTS. L'avantage de cette architecture est que le framework .NET facilite *l'interopérabilité* (projets constitués de sources en différents langages). Avant cela, faire cohabiter différents langages pour un même exécutable imposait des contraintes de choix de types "communs" aux langages, ainsi que de respecter des conventions d'appel de fonctions pour chacun des langages utilisés.

Evolution du .Net Framework (cf. page de doc Microsoft) **:**

Version	Date sortie	Visual Studio	C#	Installé dans Windows	CLR	Nouveautés majeures
1.0	13/02/02	2002	1		1.0	1ère version
1.1	24/03/03	2003	1	XP Server 2003	1.1	MAJ ASP.Net et ADO.Net
2.0	07/11/05	2005	2	Server 2003 R2	2.0	Génériques Ajouts ASP.Net
3.0	06/11/06		2	Vista Server 2008	2.0	WPF, WCF, WF

3.5	19/11/07	2008	3	7 Server 2008 R2	2.0	Sites web AJAX LINQ Données dynamiques
4.0	12/04/10	2010	4	Server 2008 R2 SP1	4	Bibliothèque de classes étendue, classes portables
4.5	12/09/12	2012	5	8 Server 2012	4	MAJ WPF, WCF, WF, ASP.Net Prise en charge applis Windows Store
4.5.1	17/10/13	2013	5	8.1 Server 2012 R2	4	Prise en charge applis Windows Phone Store Améliorations perf et debug
4.6	20/07/15	2015	6	10 Server 2016	4	ASP.Net Core 5 Compilation .Net native
4.7	02/05/17	2017	7	10 (1703) Server 2016	4	Prise en charge applis High DPI Support Touch dans WPF pour Windows 10
4.8	18/04/19	2019	7.3	10 (1905) Server 2019	4	Amélioration du support du High DPI WCF ServiceHealthBehavior

Les principaux avantages du .net framework sont :

- Tous les avantages apportés par le runtime : environnement d'exécution sécurisé, gestion de la mémoire, des exceptions, développement simplifié, intégration avec l'OS…
- Beaucoup de langages de programmation compatibles, et pas seulement ceux de Microsoft (ex : IronPython, IronRuby…)
- Interopérabilité avec du code non managé
- Une bibliothèque de classes extrêmement riche, permettant de faire tous types d'applications
- Compatibilité avec Linux et MacOS grâce à Mono, qui est une mise en œuvre open source de.net gérée par la société Xamarin (rachetée en 2016 par Microsoft)

Les inconvénients :

- Ce framework est assez lourd (plusieurs centaines de Mo) et non modulaire (on installe tout ou rien)
- La cohabitation d'applications utilisant des versions différentes du framework peut parfois poser problème.
- Ce framework n'est pas nativement multiplateformes, contrairement à Java
- Les performances du code managé sont généralement moins bonnes que celles du code natif
- L'environnement de développement Visual Studio nécessite une machine puissante.

2.2 .Net Core

Le .Net Framework est désormais en fin de vie. Depuis 2016, Microsoft concentre ses efforts sur un nouveau framework nommé **.Net Core**.

Celui-ci résout une bonne partie des inconvénients énumérés ci-dessus. En effet, .Net Core est

- Modulaire
- Déployable avec l'application, ce qui permet de faire cohabiter des applications utilisant des versions de .Net Core différentes

- Multiplateformes (Windows, Linux, MacOS)

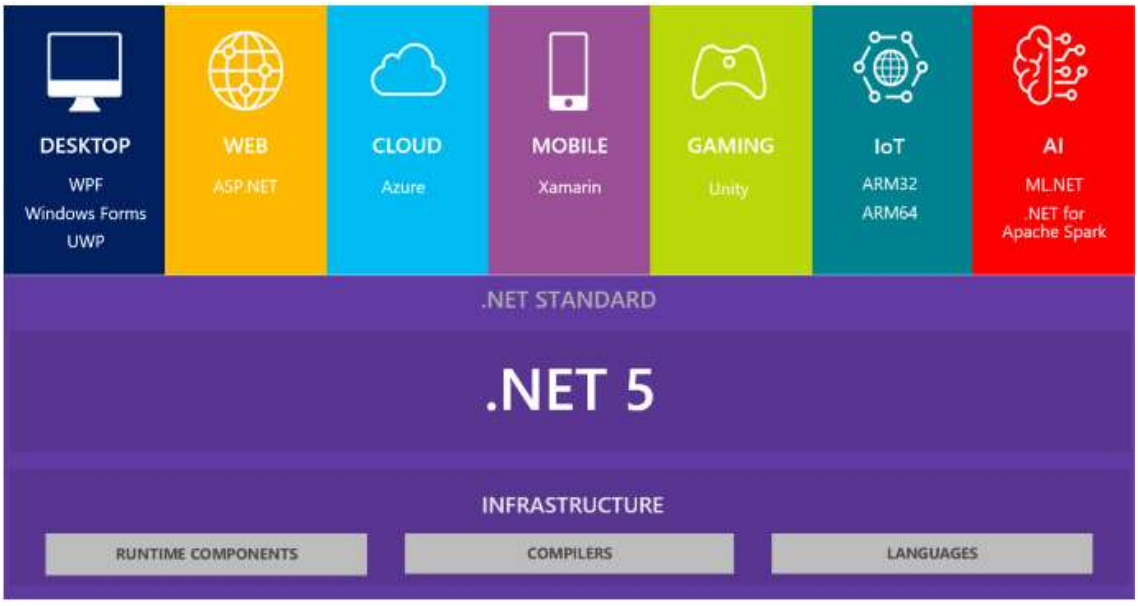
Sa bibliothèque de classes n'est pas encore aussi riche que celle du .Net Framework, mais elle s'enrichit rapidement. Voici son historique des versions majeures (cf. cette page et les suivantes) :

Version	Date sortie	Visual Studio	C#
1.0	27/06/16	2015 - 3	7.3
1.1	16/11/16	2017 15.0	7.3
2.0	14/08/17	2017 15.3	7.3
3.0	23/09/19	2019 16.3	8.0

2.3 .Net

.Net Core 3.1, sorti en décembre 2019, sera la dernière version de .Net Core.
Les différents frameworks .Net (standard, Core, Mono) laisseront place à un seul framework, nommé simplement « .Net ». Celui-ci sortira en novembre 2020, avec le N° de version 5 pour s'inscrire dans la continuité des versions actuelles. Microsoft prévoit ensuite de sortir une nouvelle version majeure de .Net tous les ans au mois de Novembre.

.Net se destine à un ensemble très vaste de types d'applications et de plateformes :



3 Création d'une application dans Visual Studio

3.1 Solution, projet, assembly

La solution est le conteneur de plus haut niveau d'une application. Elle peut contenir un ou plusieurs projets. Elle est décrite par un fichier .sln, au format texte.

Un projet est un ensemble de fichiers sources destiné à être compilé pour produire un fichier binaire dll ou exe, que l'on nomme **assembly**. Tous les fichiers sources d'un même projet doivent être écrits dans le même langage. Un projet est décrit par un fichier .csproj au format xml.

Les propriétés d'un projet permettent de définir entre autres, le nom de l'assembly généré, la version du .net framework ciblé, l'emplacement du fichier binaire à générer, et le chemin du répertoire où aller chercher les assemblies référencées.

3.2 Références

Un projet peut faire référence à :

- Des assemblies externes
- D'autres projets de la même solution

Le code source du projet peut alors utiliser les objets décrits dans ces assemblies ou projets.

Les assemblies référencées peuvent être celles fournies par Microsoft avec Visual Studio, ou des assemblies créées par soi-même ou par d'autres éditeurs.

Pour les grosses solutions contenant de nombreux projets, une bonne pratique est de spécifier le chemin du répertoire contenant les assemblies externes dans un fichier .targets centralisé (placé à côté du fichier .sln par exemple), et de le référencer dans les fichiers csproj. De cette façon, on peut très facilement modifier le chemin de recherche des assemblies pour l'ensemble des projets de la solution.

Exemple de fichier targets :

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <ReferencePath>$(SolutionDir)\References\$(Configuration)\;</ReferencePath>
  </PropertyGroup>
</Project>
```

On utilise ici la variable $(SolutionDir) pour désigner le chemin du répertoire de la solution, ce qui est plus souple que de définir un chemin en dur si on est amené à déplacer la solution.

$(Configuration) peut prendre la valeur « debug » ou « release ». On peut ainsi selon le mode d'exécution de l'application, référencer des assemblies compilées en mode debug ou release.

Exemple de fichier csproj :

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="12.0" DefaultTargets="Build"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Import Project="$(SolutionDir)PR.Targets" />
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
    <DebugSymbols>true</DebugSymbols>
    <DebugType>full</DebugType>
    <Optimize>false</Optimize>
```

```
    <OutputPath>bin\Debug\</OutputPath>
    <DefineConstants>DEBUG;TRACE</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
    <Prefer32Bit>false</Prefer32Bit>
    <UseVSHostingProcess>true</UseVSHostingProcess>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Release|AnyCPU' ">
    <DebugType>pdbonly</DebugType>
    <Optimize>true</Optimize>
    <OutputPath>bin\Release\</OutputPath>
    <DefineConstants>TRACE</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
    <Prefer32Bit>false</Prefer32Bit>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="System" />
    <Reference Include="System.Core"/>
    <Reference Include="System.Runtime.Serialization" />
    <Reference Include="System.Xml.Linq"/>
    <Reference Include="System.Xml" />
    <Reference Include="WindowsBase" />
  </ItemGroup>
  <ItemGroup>
    <Compile Include="Program.cs" />
    <Compile Include="Animal.cs" />
  </ItemGroup>
  <ItemGroup>
    <ProjectReference Include="..\Tools\Tools.csproj">
      <Project>{BBCFE307-45C6-4A3A-830F-7BD22C218275}</Project>
      <Name>Tools</Name>
    </ProjectReference>
  </ItemGroup>
  <ItemGroup />
  <Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
</Project>
```

On voit entre autres :

- La référence au fichier targets
- Les références aux assemblies Microsoft (System, Sytem.Core…)
- Une référence à un autre projet de la solution (Tools.csproj)

3.3 Compilation, exécution, débogage

3.3.1 Compilation

La compilation d'un projet consiste à générer le fichier binaire (assembly) dll ou exe.

La compilation peut se faire dans différents modes : debug, release ou autre mode personnalisé.

Lorsqu'on compile un projet P1 qui référence un autre projet P2 de la solution, P2 est compilé en premier si ce n'est pas déjà fait.

Visual Studio est assez intelligent pour compiler les projets dans le bon ordre et uniquement ceux qui ont été modifiés depuis la dernière compilation.

3.3.1.1 Exécution et débogage

La solution contient toujours au moins un projet maître, qui est celui dont l'assembly (fichier exe) est exécuté en premier. Cet exe ne peut néanmoins pas fonctionner sans les assemblies qu'il référence.

L'exécution en mode debug lance l'exe du projet maître, et arrête l'exécution au premier point d'arrêt rencontré, s'il y en a un. Le mode debug permet d'exécuter le code pas à pas et d'examiner en détail les valeurs des variables.

L'exécution en mode release lance l'exe, et ne redonne pas la main au développeur ; elle ne s'arrête pas aux points d'arrêt. L'application peut être lancée soit depuis Visual Studio, soit en double-cliquant sur le fichier exe situé dans le répertoire de génération de l'appli.

Il est possible de lancer manuellement l'application compilée en debug, en double-cliquant sur l'exe, puis d'attacher ensuite Visual Studio au processus de l'application pour pouvoir la déboguer. Ceci peut faire gagner du temps lorsqu'il s'agit de déboguer un scénario un peu long à reproduire.

4 Héritages du langage C

Le C# reprend beaucoup d'éléments de syntaxe du C :

- Structures d'instructions similaires (terminées par ;) : déclaration de variables, affectation, appels de fonctions, passage des paramètres, opérations arithmétiques
- Blocs délimités par {}
- Commentaires // ou /* */
- Structures de contrôles identiques : if/else, while, do/while, for
- Portée des variables : limitée au bloc de la déclaration

4.1 Premier exemple console

L'exemple ci-dessous illustre :

- Les similitudes avec la syntaxe du C
- La fonction principale **`Main(...)`** qui constitue le point d 'entrée d'une application
- La déclaration de variables (n'importe où avant leur utilisation, et pas obligatoirement au début de la fonction)
- Les entrées/sorties (saisies clavier + affichage) en mode console, grâce aux méthodes de la classe System.**`Console`**

```
using System;
namespace ConsoleApplication
{
   class Program
   {
         static void Main(string[] args) // Point d'entrée
         {
            int x = 2;
            for (int i = 0; i < 5; i++)
            {
               x += i;
               Console.WriteLine("i={0} x={1}", i, x);
            }
            float f = 2F;
            while (f < 1000F)
            {
```

```
            f = f * f;
            Console.WriteLine("f={0}", f);
        }

        // Lecture de la ligne saisie par l'utilisateur, puis affichage
        Console.WriteLine("Saisissez du texte et appuyez sur Entrée");
        string str = Console.ReadLine();
        Console.WriteLine("Ligne lue = {0}", str);

        // Attend l'appui d'une touche afin d'empêcher la console de se fermer
        Console.ReadKey(true);
    }
  }
}
```

```
Sortie :
i=0 x=2
i=1 x=3
i=2 x=5
i=3 x=8
i=4 x=12
f=4
f=16
f=256
f=65536
Saisissez du texte et appuyez sur Entrée
Ligne lue = ...
Appuyez sur une touche pour continuer...
```

Les arguments de la ligne de commande

Lorsqu'on exécute un programme console depuis l'invite de commandes, on peut passer des paramètres qui sont reçus en arguments de la fonction Main(). Ex :

`C:\Essais\ConsoleApplication\bin\release > ConsoleApplication Toto 23 Texte`
L'application est appelée avec les paramètres suivants :

- Argument n° 0 [Toto]
- Argument n° 1 [23]
- Argument n° 2 [Texte]

```
static void Main(string[] args)
{
   Console.WriteLine("Programme appelé avec les paramètres suivants:");
   for (int i = 0; i < args.Length; i++)
   {
      Console.WriteLine("argument n° {1} [{0}]", args[i], i);
   }
}
```

4.2 Types primitifs du C# (alias de types .NET)

Le framework .NET fournit les types utilisables, et ce quel que soit le langage utilisé (VB.NET, C#, C++.NET…). Dans C#, les types élémentaires sont nommés comme suit :

`bool` : booléen (valeur vrai ou faux) sur 1 bit

`char` : caractère Unicode 16bits

`sbyte`/`byte` : entier 8 bits signé/non signé

`short`/`ushort` : entier 16 bits signé/non signé

`int`/`uint` : entier 32 bits signé/non signé

`long`/`ulong` : entier 64 bits signé/non signé

`float` : réel 32 bits

`double` : réel 64 bits

`decimal` (128 bits) : entier multiplié ou divisé par une puissance de 10

`string` : chaîne de caractères. Type référence qui s'utilise comme un type valeur.

4.3 Littéraux / format des constantes

En C#, une lettre placée après un littéral numérique permet de spécifier son type, qui doit être en accord avec le type de la variable qui stocke le nombre.

Constantes entières

```
int x = 10; // entier
long l = 10000L; // entier long
```

Constantes réelles

Par défaut, les littéraux numériques à virgule sont considérés comme des double.

Pour qu'ils soient considérés comme des float, il faut ajouter le suffixe F.

Pour qu'ils soient considérés comme des décimaux, il faut ajouter le suffixe m.

```
double d1 = 10.2;  //OK
float f1 = 10.2;  //Error
float f2 = 10.2F;  // OK
double d2 = 1.2E-2;  // d2 = 0,012
float f3 = 3.1E2F; // f3=310
decimal d4 = 2.5m;
```

Constantes chaînes de caractères

```
String s1 = “abc”;
```

Pour représenter certains caractères spéciaux, il faut utiliser une séquence d'échappement, constituée par le caractère « \ » suivi d'un caractère de contrôle.

Séquences d'échappement les plus courantes (liste complète ici) :

\n (newline) : nouvelle ligne

\t (horizontal tab) : tabulation horizontale

\b (backspace) : retour arrière

\r (carriage return) : retour charriot

\\ (backslash) : antislash

\" (double quote) : guillemet double

```
string s2 = "Bonjour\nJe m'appelle \"Zorro\"";
Console.WriteLine(s2) ;
```

L'affichage de s2 ressemblera à ceci :

```
Bonjour
Je m'appelle "Zorro"
```

Chaîne Verbatim

Pour ne pas avoir à spécifier les séquences d'échappement, on peut simplement préfixer la chaîne par @. C'est ce que l'on appelle une chaîne Verbatim. Pour afficher des guillemets doubles dans une chaîne Verbatim, il faut les doubler :

```
string s3 = @" Bonjour
Je m'appelle ""Zorro""";
Console.WriteLine(s3) ;
```

L'affichage de s3 sera ici strictement identique à celui de s2

4.4 Fonctions et paramètres

4.4.1 Définition des fonctions

En C#, les fonctions sont nécessairement définies au sein de classes ou structures, c'est pourquoi on les appelle aussi **méthodes** (terme utilisé en programmation objet).

Une fonction comporte un en-tête et un corps :

```
static int CompterMots(string phrase) // en-tête de la fonction
{
   // Corps de la fonction
}
```

L'en-tête définit :

- **Le type du résultat** renvoyé par la fonction (ici : int). Si la fonction ne renvoie pas de résultat, il faut tout de même spécifier le type **void**.
- **Le nom** de la fonction, ici : CompterMots
- **Les paramètres** de la fonction, placés entre parenthèses. Chaque paramètre est décrit par son type et son nom. S'il y a plusieurs paramètres, on les sépare par des virgules. Exemple :

```
static int Additionner(int a, int b)
```

4.4.2 Paramètres passés par valeur

Lorsqu'on appelle une fonction, il faut renseigner ses paramètres. Dans l'exemple ci-dessous, on appelle une fonction MultiplierPar2 en lui fournissant en paramètre la variable n. La valeur de cette variable est assignée au paramètre nombre, c'est à dire copiée dans la variable nombre.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 3;
        int res = MultiplierPar2(n);

        Console.WriteLine("Résultat de la fonction : " + res);
        Console.WriteLine("Valeur de n après appel de la fonction : " + n);
        Console.ReadKey();
    }

    private static int MultiplierPar2(int nombre)
    {
        nombre *= 2;
        return nombre; // return renvoie le résultat de la fonction
    }
}
```

Sortie console :

```
Résultat de la fonction : 6
Valeur de n après appel de la fonction :  3
```

La fonction n'a pas modifié la valeur initiale de la variable n, car elle a utilisé une copie de n. Les éventuelles modifications du paramètre dans la fonction n'ont aucune incidence sur la variable passée en paramètre lors de l'appel de la fonction. C'est ce qu'on appelle le passage par valeur.

4.4.3 Paramètres passés par référence

Il est possible de faire en sorte qu'une fonction puisse modifier les variables qu'on lui passe en paramètres. On utilise pour cela le mot clé ref placé devant la déclaration du paramètre, et devant la variable passée en paramètre.

Le mot clé **ref** indique que c'est la **référence** de la variable qui sera passée au paramètre, plutôt qu'une copie de sa valeur. C'est-à-dire que le paramètre pointera sur la même case mémoire que la variable.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 3;
        MultiplierPar2(ref n); // Passage de la référence de n en paramètre

        Console.WriteLine("Valeur de n après appel de la fonction : " + n);
        Console.ReadKey();
    }

    private static void MultipliePar2(ref int nombre)
    // Nombre pointe sur la même case mémoire que la variable passée en paramètre
    {
        nombre *= 2; // Modifie réellement n
```

```
        // Pas besoin de valeur de retour pour la fonction
    }
}
```

La sortie console est : `Valeur de n après appel de la fonction : 6`

Pour faire un passage de paramètre par référence, il faut que :

- La variable passée en paramètre (n) soit initialisée **avant** l'appel de la fonction.
- Le mot clé ref soit placé devant la déclaration du paramètre dans l'en-tête de la fonction, et devant le nom de la variable dans l'appel de la fonction

Notons que comme la variable passée en paramètre est réellement modifié par la fonction, la fonction n'a pas besoin de renvoyer de résultat (retour de type void).

4.4.4 Paramètres de sortie

Les paramètres que nous avons vus précédemment sont des paramètres d'entrée de la fonction. La fonction reçoit les valeurs ou références de variables extérieures et les utilise dans son code interne.

Mais on peut également définir des paramètres de sortie. Dans ce cas, c'est la fonction qui fournit les valeurs de ces paramètres et c'est le code externe qui exploite ces valeurs.

Pour déclarer un paramètre de sortie, on utilise le mot-clé **out** placé juste avant.

Voici un exemple de mise en œuvre avec une fonction qui génère un nombre aléatoire :

```
class Program
{
    static void Main(string[] args)
    {
        int n; // variable non initialisée
        GenererNombreAleatoire(out n); // Passage de la référence de n en paramètre

        Console.WriteLine("Valeur de n : " + n);
        Console.ReadKey(true);
    }

    static void GenererNombreAleatoire(out int nombre)
    {
        Random rd = new Random(); // Classe pour générer des nombres aléatoires
        nombre = rd.Next();
        // La valeur de nombre (et de n) est initialisée par la fonction elle-même
    }
}
```

Remarque : les paramètres de sortie sont passés par référence. Le code qui appelle la fonction récupère donc une référence vers la variable passée en paramètre de sortie.

Les paramètres de sortie sont surtout utilisés lorsque la fonction doit fournir plusieurs résultats. Dans L'exemple ci-dessous, la fonction renvoie l'espèce et la famille de l'animal reçu :

```
static void DécrireAnimal (string animal, out string espèce, out string famille)
{
}
```

Evolutions apportées par C#7 :

- La variable passée en référence peut être déclarée en même temps, ce qui simplifie la syntaxe :

```
GenererNombreAleatoire(out int n);
```

- La notion de **tuple** apporte une alternative intéressante aux paramètres de sortie, car elle permet à une fonction de renvoyer plusieurs résultats, comme le montre l'exemple suivant :

```
public static (string espèce, string famille) DécrireAnimal(string animal)
{
    esp = ...;
    fam = ...;
    return (esp, fam);
}
```

4.5 Enumérations

4.5.1 Enumération simples

Un type énuméré représente une liste de valeurs prédéfinies et nommées. On le déclare avec le mot clé **enum**, suivi d'un nom et de la liste des valeurs possibles entre accolades.

```
// Déclaration d'un type énuméré
public enum Feux { Vert, Orange, Rouge }
```

NB/ Les types énumérés sont généralement déclarés en dehors des classes de façon à être utilisables dans plusieurs classes. Ils sont généralement représentés par de noms au pluriel.

L'exemple ci-dessous montre comment créer et utiliser une variable du type énuméré défini précédemment :

```
class Program
{
    static void Main(string[] args)
    {
        // Déclaration et initialisation d'une variable énumérée
        Feux feu = Feux.Vert;

        // Test de la valeur
        if (feu == Feux.Vert)
            Console.WriteLine("Passez");
```

Les valeurs d'un type énuméré sont stockées en mémoire sous forme d'entiers, c'est pourquoi il est possible de transtyper une variable de type énuméré en int :

```
        // On peut transtyper une variable énumérée en int
        if ((int)feu == 2)
            Console.WriteLine("Arrêtez-vous");
```

Par défaut les valeurs énumérées correspondent à la suite de nombres 0,1,2,3… Mais on peut leur affecter explicitement des valeurs différentes si on le souhaite :

```
public enum Feux { Vert=50, Orange=60, Rouge=70 }
```

Pourquoi déclarer un type énuméré et ne pas utiliser directement des entiers ? Un type énuméré permet de :

- Restreindre la liste des valeurs que pourront prendre les variables de ce type
- Associer des noms explicites aux valeurs, ce qui facilite le codage
- Modifier facilement la valeur entière associée à une valeur de la liste si besoin

Dans notre exemple, la notion de feu tricolore pourrait être représentée par trois entiers 0, 1 et 2. Mais il ne serait pas évident pour le développeur de souvenir à quelle valeur correspond chaque couleur. Et rien ne l'empêcherait de faire des comparaisons avec des valeurs sans signification (> 2).

4.5.2 Enumérations à indicateurs binaires

On a quelquefois besoin de manipuler des **combinaisons** de valeurs énumérées.

Par exemple, pour modéliser les droits d'accès d'un utilisateur sur des fichiers ou répertoires, on pourrait utiliser le type énuméré suivant :

```
public enum Droits { Aucun, Lecture, Création, Exécution, Modification }
```

…mais on ne pourrait pas déclarer une variable de ce type pour représenter la combinaison des droits de lecture **et** modification par exemple.

Dans ce cas, il faut déclarer le type énuméré de la façon suivante :

```
[Flags]
public enum Droits
{
    Aucun = 0,
    Lecture = 1,
    Création = 2,
    Exécution = 4,
    Modification = 8
}
```

On ajoute l'attribut [Flags] juste au-dessus de la déclaration, et on affecte explicitement les valeurs énumérées avec des puissances de deux ($2^0 = 1$, $2^1 = 2$, $2^2 = 4$…). La valeur 0 signifie « aucune valeur ».

Une variable de ce type peut stocker n'importe quelle **combinaison** des valeurs définies dans la liste.

Le fonctionnement de ce type d'énumération est basé sur les représentations binaires des nombres et sur les opérateurs binaires ~, &, |, ^ dont voici les tables de vérité :

A	**B**	**~A** (Négation)	**A & B** (ET)	**A | B** (OU)	**A ^ B** (OU exclusif
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Ainsi, pour représenter les droits de lecture et d'exécution sur un fichier, on pourra utiliser la variable suivante :

```
Droits droitsUtilsateur = Droits.Exécution | Droits.Lecture;
```

/!\ On utilise bien l'opérateur binaire **OU** pour représenter les droits de lecture **et** d'écriture.

L'opérateur | peut être vu comme une somme et l'opérateur & comme une multiplication.

L'exemple ci-dessous résume les opérations les plus courantes réalisées avec les énumérations à indicateurs binaires :

```
// Création d'un dictionnaire d'utilisateurs avec des droits associés
var droitsUtilisateurs = new Dictionary<string, Droits>();
droitsUtilisateurs.Add("Yves", Droits.Aucun);
droitsUtilisateurs.Add("Aline", Droits.Lecture);
droitsUtilisateurs.Add("Marie", Droits.Exécution | Droits.Modification);
droitsUtilisateurs.Add("Eric", Droits.Lecture | Droits.Exécution | Droits.Modificati
on);
droitsUtilisateurs.Add("Denis", Droits.Lecture | Droits.Création | Droits.Exécution
|  Droits.Modification);

Console.WriteLine("Utilisateurs ayant au moins les droits de lecture et d'exécution
: ");

Droits LectureExec = Droits.Lecture | Droits.Exécution;
foreach (var d in droitsUtilisateurs)
{
    if ((d.Value & LectureExec) == LectureExec)
        Console.WriteLine(d.Key); // Affiche Eric et Denis
}

Console.WriteLine("Utilisateurs ayant au moins le droit de lecture ou d'exécution :
");
foreach (var d in droitsUtilisateurs)
{
    if ((d.Value & LectureExec) != 0)
        Console.WriteLine(d.Key); // Affiche Aline, Marie, Eric et Denis
}

// Ajout du droit de modification à Aline
droitsUtilisateurs["Aline"] |= Droits.Modification;

// Retrait du droit de modification à Eric
droitsUtilisateurs["Eric"] &= ~Droits.Modification;
```

Pour comprendre le test réalisé dans la première instruction if, on peut examiner sa représentation binaire. Prenons l'exemple de l'utilisateur Éric :

d.Value = 1101

LectureExec = 0001 | 0100 = 0101

d.Value & LectureExec = 1101 & 0101 = 0101 = LectureExec

5 Gestion des erreurs

Au cours de l'exécution d'une application, de nombreuses erreurs peuvent survenir, pour de multiples raisons. Les erreurs peuvent être liées à des défauts de conception/réalisation de l'application, à son interaction avec son environnement extérieur (système d'exploitation, réseau, autres applications…), ou bien aux informations saisies ou importées dans l'application.

Exemples :

- L'utilisateur réalise un enchaînement d'actions qui n'a pas été prévu à la conception
- Une coupure réseau intervient durant un accès à des ressources réseau
- Une autre application monopolise une trop grande partie de la mémoire ou du processeur, ce qui provoque des problèmes dans notre application
- L'utilisateur saisit des données dont les valeurs ou les formats ne sont pas conformes à ce qu'attend l'application
- …etc.

Gérer correctement les erreurs consiste à :

- Identifier les erreurs que l'on peut et veut gérer
- Faire en sorte qu'elles ne fassent pas planter l'application lorsqu'elles surviennent, tout en informant l'utilisateur de façon approprié (affichage de messages, log dans un fichier…)

5.1 Lever une exception

Une **exception** est une classe .net qui modélise une erreur. Lorsqu'un bloc de code identifie une situation qui l'empêche de fonctionner correctement, il peut **lever** (= émettre) une exception décrite par un type (une classe) d'exception.

Ex : une méthode est chargée d'écrire dans un fichier existant, mais celui-ci est en lecture seule. La méthode peut dans ce cas lever une exception afin de signaler au code appelant qu'elle n'est pas en mesure de s'exécuter correctement, et lui fournir des informations sur le type d'erreur rencontré.

C'est un mécanisme souple et puissant, qui permet au code appelant de gérer proprement les erreurs. Il est abondamment utilisé dans les classes du .net framework, et on peut bien entendu l'utiliser dans notre propre code.

Les types d'exception

Le .net framework fournit beaucoup de types d'exceptions pour décrire toutes sortes d'erreurs. Ces types forment une hiérarchie d'héritage (une arborescence), dont on pourra avoir un aperçu sur cette page. L'ancêtre de plus haut niveau est la classe Exception.

On peut utiliser directement ces classes, ou bien créer une nouvelle classe dérivée d'Exception, afin de personnaliser davantage l'exception.

Lever une exception avec throw

Pour lever une exception, on utilise le mot clé **throw**, suivi d'une instance d'exception. Exemple :

```
public static string NomMois(int mois)
{
   switch (mois)
   {
      case 1 :
          return "Janvier";
      case 2 :
          return "Février";
      ...
      case 12 :
          return "Décembre";
      default :
          throw new ArgumentOutOfRangeException("Ce n'est pas un numéro de mois");
   }
```

```
}
```

Cette méthode renvoie le nom d'un mois à partir de son numéro. Si le nombre passé en paramètre est négatif par exemple, on ne peut pas renvoyer un nom de mois. Dans ce cas, on lève une exception de type ArgumentOutOfRangeException (type fourni par le .net framework, et bien adapté à ce cas).

5.2 Intercepter une exception

On s'intéresse maintenant à la façon dont une méthode peut gérer une exception levée par une autre méthode.

Par exemple, comment le code qui appelle la méthode NomMois ci-dessus peut-il gérer l'exception ArgumentOutOfRangeException lorsqu'elle survient ?

Blocs try…catch

Pour gérer proprement les exceptions, C# propose l'instruction try…catch, qui permet une bonne séparation entre le code de gestion d'erreur et le code qui implémente la logique applicative.

Voici sa syntaxe :

```
try
{
   // code susceptible de produire une erreur
}
catch (Exception e)
{
   // code de gestion de l'erreur
}
// suite du code
```

A l'intérieur du bloc try, si une instruction provoque une erreur, l'instruction suivante n'est pas jouée, et c'est le bloc catch qui est exécuté. Après le bloc catch, la suite du code est exécutée.

Si aucune erreur n'est rencontrée, toutes les instructions du bloc try sont exécutées, et le bloc catch n'est pas exécuté.

L'argument `e` de l'instruction cacth est un objet de type **`Exception`** ou dérivé, tels que `IndexOutOfRangeException`, `FormatException`, `SystemException`, etc…

En écrivant catch (`Exception` e), on indique qu'on veut gérer tous les types d'exceptions. Si on veut gérer une ou plusieurs exceptions particulières, on utilisera plutôt des types dérivés d'Exception dans un ou plusieurs blocs catch :

```
try
{
   //code susceptible de générer des exceptions
}
catch (IndexOutOfRangeException e1)
{
   //traiter les exceptions de type indice hors limite
}
catch (FormatException e2)
{
   //traiter les exceptions de type format incorrect
}
```

```
//instruction suivante
```

La propagation des exceptions

Si aucun des blocs catch fournis ne permet de gérer l'exception, la méthode qui contient le code se termine immédiatement, et rend la main à sa méthode appelante. Si cette dernière possède un bloc catch permettant de gérer l'exception, elle le fait, sinon, la méthode se termine, et rend la main à sa méthode appelante, et ainsi de suite... C'est ce qu'on appelle la propagation des exceptions.

Finalement, si l'erreur est remontée jusqu'au plus haut niveau, sans être interceptée par un bloc catch, l'application plante. L'erreur est dite non gérée.

```
static void Main(string[] args)
{
    try
    {
        Methode1();
    }
    catch (NotImplementedException e)
    {
        Console.WriteLine("interception de l'erreur");
    }
}


public static void Methode1()
{
    Methode2();
}

public static void Methode2()
{
    Methode3();
}

public static void Methode3()
{
    throw new NotImplementedException(); // lève une exception
}
```

Dans cet exemple, une exception est levée dans Methode3. Comme elle n'est pas interceptée par Methode2 (qui ne contient pas de try catch), elle remonte à Methode1. Mais elle n'est pas non plus interceptée par cette méthode, elle remonte donc jusqu'à Main, qui elle, l'intercepte. Si Main n'interceptait pas l'erreur, l'application planterait.

Importance de l'ordre des gestionnaires d'exceptions

Si plusieurs gestionnaires d'exceptions (i.e. plusieurs blocs catchs) sont aptes à intercepter une erreur, c'est le premier de la liste qui est exécuté. C'est pourquoi l'ordre des gestionnaires est important ; il faut les ordonner du plus spécialisé au plus général.

Attention à ne pas masquer les erreurs !

Le code à l'intérieur du ou des bloc(s) catch doit être pertinent. Si on ne fait rien dans le bloc catch, l'utilisateur n'a aucun moyen de savoir qu'une erreur s'est produite, et croit que tout s'est déroulé

normalement, alors que ce n'est pas le cas ! On dit que l'erreur est masquée. Il n'est pas toujours pertinent d'afficher un message d'erreur, mais dans ce cas, il faut au minimum loguer l'erreur dans un fichier journal.

5.3 Finally et using

5.3.1 Le bloc Finally

Un bloc finally permet de garantir que le code qu'il contient sera exécuté, même si une erreur survient dans le bloc try associé, pour peu que celle-ci ne fasse pas planter l'application. Cela permet par exemple de libérer des ressources allouées dans le bloc try.

Exemple :

```
StreamWriter outputFile = null;
try
{
   outputFile = new StreamWriter(@"C:\Temp\essai.txt", true);
   outputFile.WriteLine("Coucou !");
}
finally
{
   if (outputFile != null) outputFile.Close();
}
```

Ce code écrit du texte dans un fichier texte au moyen d'un objet StreamWriter. Cet objet utilise une ressource allouée par le système d'exploitation (un handle de fichier), qu'il faut libérer lorsqu'on ne s'en sert plus, en appelant la méthode Close.

Cependant, l'utilisation de l'objet StreamWriter peut provoquer différents types d'erreurs (ex : tentative d'écriture dans un fichier en lecture seule…). Si on ne gère pas ces erreurs dans des blocs catch, il faut au minimum faire en sorte que la méthode Close soit appelée. Pour cela, on place l'appel de cette méthode dans un bloc finally.

On peut tout à fait combiner les clauses try, catch et finally, comme le montre l'exemple suivant :

```
class Program
{
   static void Main(string[] args)
   {
      try
      {
          EcritFichier();
      }
      catch (Exception e)
      {
          // permet de gérer notamment l'erreur System.UnauthorizedAccessException
          Console.WriteLine(e.Message);
      }

      Console.ReadKey(true);
   }

   static void EcritFichier()
   {
      StreamWriter outputFile = null;
      try
      {
```

```
            outputFile = new StreamWriter(@"C:\Temp\essai.txt", true);
            outputFile.WriteLine("Coucou !");
        }
        catch(DirectoryNotFoundException)
        {
            Console.WriteLine("Le répertoire spécifié n'existe pas");
        }
        finally
        {
            Console.WriteLine("Libération de la ressource");
            if (outputFile != null) outputFile.Close();
        }
    }
}
```

Dans cet exemple, la méthode EcritFichier tente d'ajouter du texte dans un fichier déjà existant. Elle gère uniquement l'erreur DirectoryNotFoundException qui se produit si le répertoire spécifié n'existe pas. Mais d'autres erreurs peuvent se produite, comme par exemple UnauthorizedAccessException si le fichier est en lecture seule. C'est pourquoi l'appel de la méthode Close est placée dans un bloc finally, afin qu'on soit sûr qu'il est exécuté, même si une erreur autre que DirectoryNotFoundException se produit.

Si le chemin du fichier existe, mais que le fichier est en lecture seule, on passera d'abord dans le bloc finally, puis dans le bloc catch de la méthode Main, qui gère toutes les exceptions (type Exception).

/!\ Attention ! Un bloc finally n'est pas exécuté si l'erreur fait planter l'application.

Ainsi, dans l'exemple précédent s'il n'y avait pas le try…catch dans la fonction Main, le bloc finally de la fonction EcritFichier ne serait jamais exécuté.

En résumé, dans l'exemple de code précédent :

- Si aucune erreur ne se produit dans le bloc try, le bloc finally est exécuté après le bloc try
- Si une erreur de type DirectoryNotFoundException se produit dans le bloc try, le bloc catch correspondant est exécuté, puis le bloc finally est exécuté
- Si une erreur d'un type différent de DirectoryNotFoundException se produit dans le bloc try, comme il n'y a pas de bloc catch correspondant dans EcritFichier, c'est le bloc catch de Main qui intercepte l'erreur (car le type Exception gère toutes les erreurs). Juste avant l'exécution de ce bloc catch, le bloc finally est exécuté. S'il n'y avait pas de bloc catch dans Main, le bloc finally ne serait pas exécuté.

5.3.2 L'Instruction using

Nous avons vu précédemment comment utiliser un bloc finally pour être sûr que le code de libération de certaines ressources soit exécuté. Mais cette solution n'est pas idéale car :

- Elle devient vite complexe si on a plusieurs ressources à libérer, car il faut dans ce cas imbriquer les blocs try et finally.
- La gestion de la valeur null est sensible : il ne faut pas oublier de tester que outputFile n'est pas null, et rien n'empêche d'utiliser accidentellement cette variable après le bloc finally

L'instruction using résout ces problèmes. Elle permet de créer un objet utilisant une ressource, et de le détruire automatiquement à la fin du bloc de l'instruction (à la fermeture de l'accolade). L'exemple de code précédent peut ainsi s'écrire de façon beaucoup plus simple, comme ceci :

```
using(StreamWriter outputFile = new StreamWriter(@"C:\ Temp\essai.txt", true))
{
   outputFile.WriteLine("Coucou !");
}
```

Cette syntaxe appelle automatiquement la méthode **Dispose** du StreamWriter à la fin du bloc. La méthode Dispose appelle elle-même la méthode Close.

La seule contrainte pour utiliser using est que l'objet instancié dans l'instruction (ici le StreamWriter) doit implémenter l'interface **IDisposable**.

Remarques :

- /!\ ne pas confondre l'instruction using avec la directive using utilisée pour les espaces de noms (cf. plus bas).
- L'instruction using est beaucoup utilisée pour gérer les accès à la base de données (connexion et exécution de commandes).
- On peut tout à fait imbriquer les instructions using, mettre des using dans des blocs try…etc.

6 Classes

6.1 Le modèle objet en bref

La programmation orientée objet (POO) consiste à décrire un système sous forme de classes et d'interfaces.

Une classe est un modèle (=type) d'objet, qui décrit complètement toutes les caractéristiques permettant de créer des exemplaires.

Un objet est un exemplaire d'une classe, sa représentation en mémoire. Un terme synonyme est **instance de classe**. Instancier une classe signifie créer un objet de cette classe.

Exemple :

- Renault Clio est un modèle de voiture. Il est représenté par une classe
- La Renault Clio de Paul et celle de Marie sont deux exemplaires de ce modèle. Ils sont représentés par des objets (= instances) de cette classe.

Une classe contient :

- **Des champs**, qui modélisent les données
- **Des méthodes,** qui modélisent des traitements/interactions sur ces données

Un objet sert donc à stocker des données dans des champs, et à les gérer au travers de méthodes.

Une interface représente un contrat, sous forme d'une liste de méthodes abstraites. Une classe implémente l'interface (= remplit le contrat) si elle fournit une implémentation de toutes les méthodes.

La POO met en œuvre trois notions fondamentales, que nous allons étudier en détail :

- L'encapsulation
- L'héritage (= dérivation)
- Le polymorphisme

La POO présente de nombreux avantages :

- **Découpage de la complexité** : la POO permet de mieux isoler les responsabilités, et ainsi de découper un système complexe en briques plus ou moins autonomes ayant des responsabilités bien définies.
- **Sécurisation** : via le principe d'encapsulation et via l'isolation des responsabilités, on ne peut pas faire n'importe quoi
- **Modularité, réutilisation** : grâce à ces principes, on peut réutiliser des classes dans différents contextes, sans avoir à recoder plusieurs fois les mêmes choses
- **Extensibilité, évolutivité** : un système bien modélisé peut facilement être étendu, enrichi au fil du temps, sans remettre en cause tout l'architecture du code

6.2 Portées et espaces de noms

La portée d'une variable ou méthode est la région du programme dans laquelle elle est utilisable. Elle est fonction de l'emplacement de sa déclaration.

Portée locale
Les accolades ouvrante et fermante {} qui forment le corps d'une méthode, définissent une portée. Les variables déclarées à l'intérieur du corps d'une méthode sont appelées variables locales, car elles sont locales à la méthode dans laquelle elles sont déclarées ; elles ne sont pas utilisables dans une autre méthode (i.e., pas dans sa portée).

Une méthode peut contenir des sous-blocs, par exemples pour les instructions if, for, switch… etc. Les variables déclarées dans ces sous-blocs ne sont utilisables qu'à l'intérieur de ceux-ci.

Portée de classe
Les accolades ouvrante et fermante qui forment le corps d'une classe créent aussi une portée. Toutes les variables déclarées dans le corps d'une classe, mais pas dans une méthode sont dans la portée de cette classe. Le nom exact en C# pour une variable de classe est **champ**. Les champs sont utilisables dans toutes les méthodes de la classe et permettent donc de partager des informations entre elles.

Un espace de noms désigne une portée de niveau supérieur à la portée de classe. Les espaces de noms permettent simplement de regrouper les types (classes, structures, énumérations) de façon cohérente, et d'éviter des conflits de noms. On peut ainsi avoir un même nom de type dans différents espaces de noms, comme le montre l'exemple ci-dessous :

```
namespace Espace1  // espace de noms contenant deux types A et B
{
   class A { }
   class B { }
}

namespace Espace2  // autre espace de noms contenant des types de mêmes noms
{
   class A { }
   class B { }
}
```

```
// espaces de noms imbriqués
namespace Espace2
{
   namespace ClassesImportantes
   {
      class A { }
      class B { }
   }
}

// Autre façon équivalente d'imbriquer les espaces de noms
namespace Espace2.ClassesImportantes
{
   class A { }
   class B { }
}


// Exemple d'utilisation des types des différents espaces de noms
namespace ExempleNamespace
{
   class Program
   {
      static void Main(string[] args)
      {
         Espace1.A obj1 = new Espace1.A();
         Espace2.A obj2 = new Espace2.A();
         Espace2.ClassesImportantes.A obj3 = new Espace2.ClassesImportantes.A();
      }
   }
}
```

Le nom complet d'un type est donc composé de ses espaces de noms et du nom du type, séparés par des points.

Si on ne spécifie pas d'espace de noms, le compilateur en ajoute un par défaut, sans nom, qu'on appelle espace de noms global.

/!\ Le découpage d'un ensemble de classes en assemblies (dll) est indépendant du découpage des espaces de noms. Ne pas confondre les deux notions.

Directive using

Dans un espace de noms donné, pour utiliser un type contenu dans un autre espace de noms, il faut utiliser son nom complet (c'est-à-dire préfixé de son espace de nom), ce qui peut être fastidieux et rendre le code très verbeux. L'utilisation de la directive using facilite les choses.

Cette directive permet d'utiliser les éléments d'un espace de noms, sans avoir à les qualifier par leur nom complet (espace de noms + nom).

Ex : le nom complet de la classe Console est System.Console (System est son espace de nom). Pour l'utiliser, on devrait donc écrire :

```
System.Console.WriteLine("Hello!);
```

Grâce à la directive **`using`**, on peut simplifier cette écriture :

```
using System;

namespace ConsoleApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Hello!);
         // le type Console est automatiquement trouvé dans le namespace System
       }
   }
}
```

On peut bien entendu appliquer ce principe à tous les espaces de noms dont on a besoin dans le code :

```
using System;
using System.Globalization;

namespace ConsoleApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Hello!);
         // le type Console est automatiquement trouvé dans le namespace System
          var s = String.Format(CultureInfo.InvariantCulture, "La date : {0}",
DateTime.Today);
          // le type CultureInfo est automatiquement trouvé dans le namespace
System.Globalization
          Console.WriteLine(s);
       }
   }
}
```

Remarque : si on utilise plusieurs directives using, et que les espaces de noms correspondants comportent des noms de types identiques, dans ce cas, pour utiliser ces types il faudra tout de même spécifier leurs noms complets.

Pour utiliser une classe du .net framework, il faut connaître son espace de noms. Visual Studio sait reconnaître les types du .net framework utilisés dans le code, et propose spontanément de préfixer le nom du type par celui de son namespace, ou d'ajouter la directive using manquante.

On peut aussi se référer à cette page de doc Microsoft qui décrit tous les espaces de noms, et qui permet donc de rechercher les classes dont on a besoin sans connaître leur nom à l'avance.

6.3 Accessibilité des membres d'une classe

Rappel : la portée d'une variable ou méthode est la région du programme dans laquelle elle est utilisable.

Les membres (champs, propriétés, méthodes) définis au sein d'une classe sont par défaut dans la portée de cette classe, c'est-à-dire qu'ils ne sont accessibles que dans le corps de la classe. Cependant, on peut leur appliquer individuellement des **modificateurs d'accès**, afin qu'ils soient visibles en dehors de la classe. Les modificateurs sont les suivants, du plus restrictif au plus permissif :

- private (valeur par défaut) : ne modifie rien ; le membre reste accessible uniquement à l'intérieur de la classe
- protected : le membre est accessible aussi dans les classes dérivées
- Internal : le membre est accessible dans tout l'assembly (=projet) courant
- public : le membre est accessible dans tout l'assembly courant, et dans les assemblies qui y font référence.

Remarques :

- Il n'est pas obligatoire de spécifier le modificateur private, puisque c'est la valeur par défaut, cependant, il est tout de même conseillé de le faire pour éviter toute ambigüité.
- Les modificateurs d'accès s'appliquent aussi aux membres des structures, sauf protected, puisqu'une structure ne peut pas être dérivée.

Les modificateurs d'accès sont à la base du principe d'encapsulation. On ne doit rendre public que le strict nécessaire pour que les objets soient manipulables.

Exemple :

```
public class Animal
{
   // Champs privés (accessibles uniquement à l'intérieur de la classe)
   private float _dureeVie;
   private float _poids;

   // Propriété publique (accessible à l'extérieur de la classe)
   public float Poids
   {
      get { return _poids; }
   }

   // Méthode protégée (accessible dans les classes dérivées de Animal)
   protected void Grossir(float qteNourriture)
   {
      _poids += (float)0.1 * qteNourriture;
   }

   // Méthodes publiques (accessibles à l'extérieur de la classe)
   public void Manger(float qte)
   {
      Grossir(qte);
   }

   public override string ToString()
   {
      return String.Format("Je suis un animal qui pèse {0} kg", _poids);
   }
}
```

Les classes dérivées d'Animal ont accès à la propriété Poids et aux méthodes Grossir, Manger et ToString. Le reste du code (en dehors de la classe et de ses dérivées) n'a pas accès à la méthode Grossir.

6.4 Les propriétés

6.4.1 Présentation

En C#, on peut définir des *propriétés*. Il s'agit de membres qui se manipulent comme des champs, mais dont l'accès est contrôlé par des fonctions en lecture et en écriture (get et set), appelés **accesseurs**.

Dans les propriétés, le mot clé **`value`** dans le bloc set désigne la valeur reçue en écriture.

```
public class Animal
{
   private float _dureeVie;
   private float _poids;

   // Propriété DureeVie qui encapsule le champ privé _dureeVie
   public float DureeVie
   {
      // Bloc get pour accéder au champ en lecture
      get { return _dureeVie;}

      // Bloc set pour modifier le champ de façon contrôlée
      set
      {
         if (value > 0 && value <= 150)
            _dureeVie = value;
         else
         {
            // lève une exception pour indiquer une valeur incorrecte
            throw new ArgumentOutOfRangeException();
         }
      }
   }

   public float Poids
   {
      get { return _poids; }
      set
      {
         if (value > 0)
            _poids = value;
         else
         {
            // lève une exception pour indiquer une valeur incorrecte
            throw new ArgumentOutOfRangeException();
         }
      }
   }

   public override string ToString()
   {
      return String.Format("Je suis un animal qui pèse {0} kg", _poids);
   }
}

public class Program
{
   static void Main(string[] args)
   {
      Animal a = new Animal();
```

```
        a.Poids = 50; // une propriété s'utilise comme un champ
        a.DureeVie = 160;  // produit une exception car la valeur est
                           // en dehors de la plage définie dans la partie set
    }
}
```

La propriété DureeVie évite d'avoir à créer deux fonctions du style GetDureeVie et SetDureeVie pour la lecture et l'écriture du champ _dureeVie. La syntaxe est plus propre, et l'utilisation plus intuitive.

Caractéristiques :

- Les deux accesseurs **get** et **set** d'une propriété ne sont pas obligatoires. Si on ne met pas d'accesseur set, la propriété n'est pas modifiable, sauf par le constructeur (de façon à pouvoir l'initialiser)
- On peut mettre un modificateur d'accès devant un des deux accesseurs. Typiquement, on peut rendre l'accesseur set protégé, de façon que la propriété ne soit modifiable que dans la classe et ses dérivées, tout en étant accessible en lecture depuis les autres classes.

```
private float _dureeVie;
public float DureeVie
{
    get { return _dureeVie; }
    protected set { _dureeVie = value; }
}
```

- Le niveau d'accessibilité sur l'accesseur doit être plus restrictif que celui sur la propriété, et le modificateur est autorisé uniquement sur un des deux accesseurs.

/!\ Pour respecter le principe d'encapsulation, attention à ne rendre les propriétés modifiables que lorsque c'est nécessaire. La plupart des classes du .net framework fournissent des propriétés non modifiables.

6.4.2 Implémentation automatique

Lorsqu'une propriété est destinée uniquement à stocker une donnée, sans fournir de code spécifique pour sa lecture ou son écriture, on peut utiliser une syntaxe très simplifiée, très commode et couramment utilisée, présentée dans l'exemple suivant :

```
public class Animal
{
    // Propriétés
    public float DureeVie { get; private set; }
    public float Poids{ get; set; }
    ...
}
```

Il s'agit d'une implémentation automatique : le compilateur génère lui-même une variable privée en interne pour manipuler la donnée.

Depuis C# 6, il est possible de ne pas mettre l'accesseur set d'une propriété implémentée automatiquement, de sorte qu'elle ne soit pas modifiable. Ex :

```
public float DureeVie { get; }
```

6.4.3 Initialisation

La valeur d'une propriété est par défaut initialisée avec la valeur par défaut de son type :

- 0 pour un nombre ou un énuméré
- false pour un booléen
- null pour un type référence

Pour une propriété implémentée automatiquement, on n'a pas accès à la variable privée interne qui stocke la donnée. Dans ce cas, il y a deux façons d'initialiser la propriété :

- Grâce au constructeur de la classe, qui est une méthode spécifique pour initialiser les champs et propriétés d'un objet. Nous aborderons les constructeurs dans un prochain chapitre.
- Au moment de sa déclaration (valable depuis C# 6) : `public float DureeVie { get; } = 20F;`

Il est important de noter que ceci fonctionne même si la propriété est non modifiable (sans bloc set).

6.5 Le principe d'encapsulation

L'encapsulation désigne le fait que :

- Un objet contient à la fois des données (champs) et les propriétés et méthodes pour les gérer
- L'utilisateur de l'objet (développeur) ne peut pas accéder directement aux champs, ni aux méthodes destinées à la gestion interne de l'objet.
  Pour utiliser l'objet, il n'a accès qu'à un ensemble restreint de méthodes et propriétés. Tous les rouages internes lui sont invisibles.

L'image ci-contre représente des boitiers permettant de commander une machine complexe (ex : une machine industrielle). La complexité interne de la machine est inaccessible et n'intéresse de toute façon pas l'opérateur. Ce dernier interagit avec la machine uniquement au travers des manettes et boutons des boîtiers de commandes, et souhaite que cette interface de commande soit simple.

Concrètement, l'encapsulation est mise en œuvre par le respect des principes suivants :

- Laisser tous les champs privés
- Mettre publiques uniquement les propriétés et méthodes qui permettent de manipuler l'objet d'un point de vue extérieur
- Rendre les propriétés non modifiables (enlever le bloc set), si leur modification par l'extérieur de la classe n'est pas nécessaire

Les propriétés et méthodes publiques forment ce que l'on appelle l'**interface publique** de l'objet (à ne pas confondre avec le mot clé Interface que nous verrons plus loin). Il faut s'efforcer de la rendre la plus simple possible pour faciliter la compréhension de l'utilisation de l'objet.

Cas particuliers :

Un champ constant (ou constante) est un champ dont la valeur ne peut jamais changer. Il doit pour cela être déclaré avec le mot clé **const**, et sa valeur doit être définie dès sa déclaration. Ce type de champ est généralement public.

```
class Math
{
   public const double PI = 3.14159265358979323846;
}
```

Un champ en lecture seule est semblable à une constante dont l'initialisation de la valeur peut être faite par le constructeur de la classe (cf. plus bas), et donc retardée jusqu'au moment de l'exécution. On le déclare avec le mot clé **readonly** et on le met généralement public.

```
class MaClasse
{
   public readonly int R;
   public MaClasse(int val)
   {
      R = val;
   }
}
```

6.6 Signature et surcharge des méthodes

La signature d'une méthode se compose du nom de la méthode, ainsi que du type et du genre (valeur, ref ou out) de chacun de ses paramètres, considérés de gauche à droite.

Le type de retour et les noms des paramètres d'une fonction ne font pas partie de sa signature. Ainsi, plusieurs méthodes avec des en-têtes différents peuvent avoir la même signature.

La surcharge des méthodes permet à une classe, une structure ou une interface de déclarer plusieurs méthodes avec le même nom, sous réserve que leurs signatures soient différentes.

Voici un exemple de surcharge d'une méthode :

```
class Essai
{
    public static double Carré(double d) {
        return d * d;
    }

    public static void Carré(ref double d) {
        d *= d;
    }

    // Pas bon car le type de retour ne permet pas de différentier 2 surcharges
    public static double Carré(ref double d) {
        d *= d;
        return d;
    }

    public static void Carré(double d, out double res) {
        res = d * d;
    }
}
```

Autre exemple : la méthode ToString possède également plusieurs surcharges.

6.7 Constructeurs et initialiseurs

Un *constructeur* est une méthode qui initialise l'état d'un objet au moment de son instanciation avec new. L'exemple ci-dessous montre une classe avec deux constructeurs :

```
public class CompteBancaire
{
    #region Champs privés
    private long _numéro;
    private DateTime _dateCréation;
    private decimal _découvertAutorisé;
    #endregion

    /// <summary>
    /// Initialisation d'un compte avec son numéro
    /// </summary>
    public CompteBancaire(long numéro)
    {
        _numéro = numéro;
    }

    /// <summary>
    /// Initialisation d'un compte avec son numéro,
    /// sa date de création et son découvert autorisé
    /// </summary>
    public CompteBancaire(long numéro, DateTime dateCréa, decimal découvertAutorisé)
    {
        _numéro = numéro;
        _dateCréation = dateCréa;
        _découvertAutorisé = découvertAutorisé;
    }
...
}

class Program
{
    static void Main(string[] args)
    {
        CompteBancaire cpt1 = new CompteBancaire(6546589);
        CompteBancaire cpt2 = new CompteBancaire(8712564, DateTime.Today, 800m);
        Console.ReadKey();
    }
}
```

Cet exemple illustre les caractéristiques des constructeurs :

- Ils portent le même nom que la classe
- Ils peuvent avoir des paramètres, dont les valeurs sont fournies au moment de l'instanciation de l'objet avec new
- Ils n'ont pas de type de retour, même pas void
- Ils peuvent être surchargés

Une classe doit obligatoirement avoir au moins un constructeur. Si on n'en définit aucun, le compilateur en génère automatiquement un par défaut, sans paramètre et avec un corps vide. Si on définit au moins un constructeur, le compilateur ne génère pas de constructeur par défaut.

6.7.1 Appel d'un constructeur par un autre

Un constructeur peut en appeler un autre. Cela permet de bénéficier d'un traitement d'initialisation déjà fourni par cet autre constructeur. Pour réaliser cet appel, on utilise le mot clé this. Ainsi, dans l'exemple précédent, on aurait pu écrire :

```
public CompteBancaire(long numéro)
{
    _numéro = numéro;
}

public CompteBancaire(long numéro, DateTime dateCréa, decimal découvertAutorisé) :
 this(numéro)
{
    _dateCréation = dateCréa;
    _découvertAutorisé = découvertAutorisé;
}
```

Le second constructeur appelle le premier avec this, en lui transmettant son paramètre numéro. Il n'a ainsi pas besoin d'initialiser lui-même le champ _numéro, ce qui évite de la répétition de code.

Remarque : Le mot clé this est aussi utilisé pour représenter l'instance courante de la classe, ce qui peut être utile pour éviter les ambiguïtés de noms, comme illustré dans l'exemple suivant :

```
class Personne
{
   private string nom;

   // Constructeur
   public Personne(string nom)
   {
      this.nom = nom;
   }
...
```

Dans cet exemple, this désigne l'instance courante de personne et this.nom fait référence à son champ nom. Comme le paramètre du constructeur a le même nom que le champ privé, l'utilisation du mot clé this est le seul moyen de distinguer les deux.

Il est cependant préférable de respecter les conventions de nommage des champs privés (préfixés par _) afin d'éviter de se retrouver dans cette situation.

6.7.2 Initialiseurs

Dans l'exemple précédent, on initialise les champs des objets grâce aux différents constructeurs :

```
CompteBancaire cpt1 = new CompteBancaire(6546589);
CompteBancaire cpt2 = new CompteBancaire(8712564, DateTime.Today, 8000m);
```

…mais si les données sont accessibles via des propriétés modifiables, C# propose également une autre technique pour initialiser des objets : les initialiseurs :

```
public class CompteBancaire
{
    // Constructeur
    public CompteBancaire(long numéro)
    {
        Numéro = numéro;
```

```
    }

    // Propriété non modifiable
    public long Numéro { get; }

    // Propriétés modifiables
    public DateTime DateCréation { get; set; }
    public decimal DécouvertAutorisé { get; set; }
...
```

```
class Program
{
    static void Main(string[] args)
    {
        // Initialisation d'un compte avec un constructeur + un initialiseur
        CompteBancaire cpt = new CompteBancaire(8712564)
        {
            DateCréation = DateTime.Now,
            DécouvertAutorisé = 800m
        };
    }
}
```

L'initialiseur est la partie entre accolades. Il permet d'affecter les valeurs des propriétés modifiables de l'instance en les nommant, et en séparant les affectations par des virgules.

Un initialiseur n'est donc qu'une syntaxe condensée regroupant l'initialisation de propriétés avec l'appel d'un constructeur.

NB/ Lorsque le constructeur appelé est celui sans paramètre, on peut omettre les parenthèses du constructeur :

```
CompteBancaire cpt = new CompteBancaire
{
   DateCréation = DateTime.Now,
   DécouvertAutorisé = 800m
};
```

On peut aussi utiliser un initialiseur pour initialiser les éléments d'un tableau ou d'une collection :

```
int[] tab = new int[] { 12, 7, 25, 9 };
```

6.8 Membres et classes statiques

Un champ statique (appelé aussi *variable de classe*) est un champ dont la valeur et partagée par toutes les instances de cette classe (si cette classe n'est pas elle-même statique).

NB/ Les constantes sont des champs statiques.

Une méthode ou propriété statique (appelée aussi *méthode de classe*) est une méthode ou propriété qui peut être appelée sans instance de classe, simplement en mettant le nom de la classe devant (ex : Console.WriteLine()). Elle n'a accès qu'aux champs statiques de la classe.

L'espace mémoire associé à un membre statique est alloué de façon statique, c'est-à-dire au lancement de l'application, contrairement à l'allocation dynamique qui est faite au fur et à mesure des besoins.

Dans l'exemple ci-dessous, Cos et PI sont respectivement une méthode statique et un champ statique de la classe Math. On voit qu'on peut les utiliser directement sans instancier la classe Math.

```
Console.WriteLine(Math.Cos(Math.PI / 3));
```

Une classe statique est une classe qui ne peut pas être instanciée. Elle ne peut contenir **que** des membres statiques.

NB/ Une classe non statique peut contenir des membres statiques et non statiques.

Pour déclarer une classe ou un membre statique, on utilise le mot clé **static** devant son nom

L'exemple ci-dessous montre comment un champ statique peut être utilisé pour compter le nombre d'instances de classes créées.

```
using System;
namespace SyntaxeCSharp
{
   class MaClasse
   {
      private static int _nbObjetsCrees = 0;
      public static int NbObjets
      {
         get { return _nbObjetsCrees; }
      }

      public MaClasse()
      {
         _nbObjetsCrees++;
      }
   }

   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine(MaClasse.NbObjets); // Affiche 0
         MaClasse m = new MaClasse();
         Console.WriteLine(MaClasse.NbObjets); // Affiche 1
      }
   }
}
```

Un constructeur statique est appelé une seule fois avant tout autre constructeur, et permet d'initialiser des champs statiques, ou de faire des actions qui ne doivent être faites qu'une seule fois. Il est appelé automatiquement avant la création de la première instance, ou avant l'accès à un membre statique.

Il ne faut pas préciser de modificateur de visibilité (public/protected/private) pour ce constructeur.

```
public class Couleur
{
   // Constructeur statique (appelé automtiquement en premier)
   static Couleur()
   {
      Console.WriteLine("Appel au constructeur statique");
   }

   // Constructeur ordinaire
```

```
    public Couleur(string c)
    {
        Console.WriteLine(c);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Couleur c = new Couleur("Vert");
    }
}
```

Sortie console :

```
Appel au constructeur statique
Vert
```

7 Composition et agrégation

La programmation orientée objet permet de décrire les applications comme des collaborations d'objets. Par exemple, certaines méthodes d'une classe utilisent les services d'une autre classe.

La composition décrit la situation où un objet est lui-même constitué d'objets d'autres classes. Par exemple un livre est constitué de différents chapitres, une voiture est constituée d'un moteur, d'une carrosserie, de roues…etc.

Dans le code, la relation de composition se traduit par le fait qu'une classe possède des instances d'autres classe, et qu'elle les instancie elle-même, généralement dans un de ses constructeurs.

L'agrégation est une relation moins forte dans laquelle un objet contient d'autre objets, mais peut exister sans eux. Par exemple, un garage peut contenir des voitures.

Dans le code, la relation d'agrégation se traduit par le fait qu'une classe agrège des instances d'une autre classe, mais que ces dernières sont créées en dehors de la classe contenante.

Exemple :

Composition	Agrégation

`public class Moteur {  }`
`public class Voiture`
`{`
`  private Moteur _moteur;`
`  public Voiture()`
`  {`
`  _moteur = new Moteur();`
`  }`
`}`
	`public class Garage`
	`{`
	`  private List<Voiture> _voitures;`
	
	`  public void RentrerVoiture(Voiture v)`
	`  {`
	`  _voitures.Add(v);`
	`  }`
	`}`

Le moteur fait partie de la voiture et est instancié par elle	Les voitures sont instanciées en dehors de la classe et ajoutées à la liste interne du garage

NB/ En pratique, même si la relation entre deux classes est une agrégation du point de vue du sens, il est courant de l'implémenter comme une composition, sans que cela pose un problème.

8 Héritage (ou dérivation)

La dérivation est une seconde relation essentielle de la programmation orientée objet.

La dérivation est un mécanisme qui permet de spécialiser une classe existante en modifiant le comportement de ses méthodes, et en lui ajoutant éventuellement de nouveaux membres. La classe dérivée possède tous les champs, propriétés et méthodes publics ou protégés de la classe de base. On peut donc considérer un objet de la classe dérivée comme un objet de la classe de base (objet dérivé **est un** objet de base).

Syntaxe :

```
public class Derivee : Ancetre
{
}

class Program
{
   static void Main(string[] args)
   {
      Derivee d = new Derivee();
      Ancetre a = new Derivee(); // autorisé
   }
}
```

8.1 Appels des constructeurs

Il est recommandé qu'un constructeur d'une classe dérivée appelle le constructeur de sa classe de base dans le code d'initialisation. Ceci peut être réalisé au moyen du mot clé **base**.

Si on ne le fait pas explicitement, le compilateur tente d'insérer automatiquement un appel au constructeur par défaut de la classe de base avant l'exécution du code du constructeur de la classe dérivée.

```
using System;

namespace EssaiHeritage
{
   // Classe ancêtre
   public class Animal
   {
      private float _poids;
      private float _taille;

      // constructeur sans paramètre
      public Animal()
      {
      }

      // constructeur avec paramètres
      public Animal(float poids, float taille)
      {
         _poids = poids;
         _taille = taille;
```

```
        }
    }

    // Classe dérivée
    public class Mammifere : Animal
    {
        // base permet d'appeler le constructeur de la classe ancêtre
        public Mammifere(float poids, float taille) : base(poids, taille)
        {

        }
    }
}
```

Dans cet exemple, si le constructeur de la classe Mammifere n'appelait pas explicitement un constructeur de son ancêtre avec le mot clé base, le constructeur sans paramètres de la classe de base serait automatiquement appelé.

Remarque : attention à ne pas confondre les mots clés this et base. This permet d'appeler un autre constructeur de la même classe, alors que base permet d'appeler un constructeur de la classe ancêtre.

8.2 Méthodes virtuelles et redéfinies

Une méthode virtuelle d'une classe ancêtre est une méthode destinée à être **redéfinie** dans ses classes dérivées.

Dans la classe ancêtre, le caractère virtuel d'une méthode est précisé à l'aide du mot clé **virtual** placé avant son type de retour.

Dans la classe dérivée, pour indiquer qu'une méthode redéfinit une méthode virtuelle, on utilise le mot clé **override**.

```
using System;

namespace EssaiHeritage
{
    public class Animal
    {
        private float _poids;
        private float _taille;

        // Constructeur
        public Animal(float poids, float taille)
        {
            _poids = poids;
            _taille = taille;
        }

        // Propriété
        public float Poids
        {
            get { return _poids; }
            set { _poids = value; }
        }

        // Méthode virtuelle (peut être redéfinie dans les dérivées)
        protected virtual void Grossir(float qteNourriture)
```

```
        {
            _poids += (float)0.1 * qteNourriture;
        }

        // Méthode non virtuelle
        public void Manger(float qte)
        {
            Grossir(qte);
        }
    }

    public class Mammifere : Animal
    {
        // Appel du constructeur de base
        public Mammifere(float poids, float taille) : base(poids, taille) { }

        // Méthode redéfinie
        protected override void Grossir(float qteNourriture)
        {
            Poids += (float)0.2 * qteNourriture;
        }
    }
}
```

Dans l'exemple ci-dessus, la méthode virtuelle Grossir est redéfinie dans la classe dérivée.

Remarque : attention à ne pas confondre redéfinition et surcharge. La surcharge consiste à créer plusieurs méthodes de même nom, mais avec des signatures différentes au sein d'une même classe. Une méthode redéfinie dans une classe dérivée a la même signature que la méthode de son ancêtre.

Une méthode redéfinie est aussi virtuelle

Une méthode redéfinie est par défaut également virtuelle, et il n'est pas nécessaire de remettre le mot clé virtual devant. Ceci permet de gérer facilement plusieurs étages de dérivation. Dans l'exemple ci-dessus, on pourrait créer une classe Chien dérivée de Mammifere et redéfinir sa méthode Grossir.

8.2.1 Appel de la méthode de base dans la méthode redéfinie

Nous avons vu plus haut comment appeler un constructeur d'une classe ancêtre dans un constructeur de la classe dérivée à l'aide du mot clé base. Cette technique s'applique également à n'importe quelle méthode virtuelle redéfinie, comme le montre l'exemple suivant :

```
using System;

namespace EssaiHeritage
{
    public class Animal
    {
        public override string ToString()
        {
            return "Je suis un animal";
        }
    }

    public class Mammifere : Animal
    {
        public override string ToString()
        {
```

```
            string s = base.ToString(); // Appelle la méthode de la classe ancêtre
            return s + ", et je suis un mammifère";
        }
    }
}
```

Remarque : dans la classe Animal, on remarque que la méthode ToString est déjà redéfinie (mot clé override). En effet, en .net, toutes les classes héritent implicitement de la classe Object, qui contient une méthode virtuelle ToString().

8.2.2 Masquage

Que se passe-t-il si on utilise le mot clé virtual, mais pas override ? il y a un avertissement à la compilation. Le code s'exécute quand-même, mais le comportement n'est pas celui attendu d'une relation de dérivation classique ; il n'est pas polymorphique (cf. plus bas). On dit que la méthode de la classe de base est **masquée** par la méthode de la classe dérivée.

Le tableau ci-dessous résume ce qui se passe dans les différents cas

Classe de base → Classe dérivée ↓	**Méthode déclarée avec virtual**	**Méthode déclarée sans virtual**
Méthode déclarée avec override	Redéfinition	Erreur de compilation
Méthode déclarée sans override	Masquage avec avertissement à la compilation	Masquage avec avertissement à la compilation

Remarque : le masquage est très rarement souhaité. Donc attention à ne pas oublier override !

8.3 Propriétés virtuelles et redéfinies

Comme les méthodes, les propriétés peuvent également être virtuelles et redéfinies.

```
using System;

namespace EssaiHeritage
{
    public class Animal
    {
        private float _dureeVie;

        // Propriété virtuelle
        public virtual float DureeVie {
            get { return _dureeVie; }
            set
            {
                if (value > 0 && value <= 150)
                    _dureeVie = value;
                else
                {
                    // lève une exception pour indiquer une valeur incorrecte
                    throw new ArgumentOutOfRangeException();
                }
            }
        }
    }
```

```
    public class Mammifere : Animal
    {
        // Propriété redéfinie
        public override float DureeVie
        {
            get { return base.DureeVie; }
            set
            {
                if (value > 0 && value <= 99)
                    base.DureeVie = value;
                else
                {
                    // lève une exception pour indiquer une valeur incorrecte
                    throw new ArgumentOutOfRangeException();
                }
            }
        }
    }
}
```

Dans cet exemple, il est intéressant de noter que dans la classe dérivée Mammifere, grâce à l'utilisation de base, on n'a pas besoin d'accéder directement au champ _dureeVie, qui peut rester privé.

8.4 Polymorphisme d'héritage

Le polymorphisme désigne le fait qu'à l'exécution, un même code peut produire des comportements différents selon le type d'entité qu'il manipule.

Pour obtenir un comportement polymorphique avec la notion d'héritage, il faut utiliser des variables du type ancêtre contenant des objets du type dérivé.

Considérons les classes suivantes :

```
using System;

namespace EssaiHeritage
{
   public class Animal
   {
      private float _poids;
      private float _taille;

      #region Constructeurs
      public Animal()
      {
      }

      public Animal(float poids, float taille)
      {
         _poids = poids;
         _taille = taille;
      }
      #endregion

      public override string ToString()
      {
         return String.Format("Je suis un animal qui pèse {0} kg", _poids);
      }
```

```
    }

    public class Mammifere : Animal
    {
        public Mammifere(float poids, float taille) : base(poids, taille) { }

        public override string ToString()
        {
            string desc = base.ToString();
            return desc + ", et je suis un mammifère";
        }
    }

    public class Poisson : Animal
    {
        public override string ToString()
        {
            return "Je suis un petit poisson";
        }
    }
}
```

Les clases Mammifere et Poisson héritent toutes deux de Animal. Animal fournit une implémentation par défaut de la méthode virtuelle ToString, qui est redéfinie dans les 2 classes dérivées.

On peut faire une utilisation polymorphique de ces classes de la façon suivante :

```
Animal a1 = new Mammifere(10, 80);
Animal a2 = new Poisson();

Console.WriteLine(a1);
Console.WriteLine(a2);
```

```
Sortie console :
Je suis un animal qui pèse 10 kg, et je suis un mammifère
Je suis un petit poisson
```

On constate donc que pour le premier animal, c'est la méthode ToString de la classe Mammifere qui est appelée, et que pour le second, c'est la méthode ToString de la classe Poisson qui est appelée. A partir de 2 variables de type Animal, on a obtenu des comportements différents de la méthode ToString. C'est bien un comportement polymorphique.

8.5 Transtypage et opérateurs is et as

La réussite d'un cast (= transtypage) repose sur la fiabilité du code produit par le développeur. Pour sécuriser cette opération, on pourrait gérer l'exception InvalidCastException, afin que l'application ne plante pas lorsque cette exception se produit. Mais C# propose d'autres alternatives plus adaptées au moyen des opérateurs is et as.

L'opérateur « is » permet de vérifier le type d'un objet avant sa conversion, en renvoyant un booléen. Voici un exemple qui illustre sa syntaxe :

NB/ Dans cet exemple, Poisson est une classe qui dérive d'Animal.

```
Animal a = new ...;
if (a is Poisson)
{
```

```
    Poisson plouf = (Poisson)a; // cast sécurisé
    // suite du code
}
```

Depuis C# 7, cette syntaxe peut être simplifiée de la façon suivante :

```
if (a is Poisson plouf)
{
    // suite du code
}
```

L'opérateur « as » fonctionne de façon un peu différente en renvoyant non pas un booléen, mais une référence. Sa mise en œuvre est illustrée par l'exemple suivant :

```
Animal a = new...;
Poisson plouf = a as Poisson;
if (plouf != null)
{
    // suite du code
}
```

L'opérateur as renvoie donc directement la référence convertie si la conversion est possible, ou null si la conversion est impossible.

Les opérateurs is et as fonctionnent aussi bien avec les types valeurs que les types références, y compris les interfaces :

```
if (j is Enum)
{
    int k = (int)j + 3;
}

Poisson p = new Poisson();
IDomestique dom = p as IDomestique;
if (dom != null)
{
    // suite du code
}
```

Remarques :

L'utilisation fréquente du transtypage peut être révélatrice d'un problème de conception du code. En effet, il est toujours préférable de ne pas avoir à faire de conversion de types, mais plutôt d'utiliser le polymorphisme et les génériques, qui permettent de faire du code plus robuste et plus performant.

9 Classes abstraites et interfaces

9.1 Les classes et méthodes abstraites

9.1.1 Définition et intérêt

Prenons l'exemple d'une classe ancêtre Animal dérivée en plusieurs classes telles que Chat, Chien, Vache, Cerf, Castor… Nous souhaitons pouvoir distinguer les animaux domestiques des autres animaux, car nous voulons leur attribuer des informations supplémentaires telles que le nom, le numéro identifiant, la date de naissance…etc.

Pour répondre à ce besoin, nous pouvons créer une nouvelle classe AnimalDomestique contenant ces nouvelles informations, avec les relations d'héritage suivantes (la flèche signifiant « hérite de ») :

Chat, Chien, Vache → AnimalDomestique → Animal

Les autres classes d'animaux peuvent continuer à hériter directement d'Animal :

Cerf, Castor → Animal

Au fur et à mesure des nouveaux besoins, on peut être amené à créer encore d'autres classes spécialisées, jusqu'à obtenir une arborescence de classes assez complexe.

Cela amène la remarque suivante : rien n'empêche de créer des instances des classes Animal ou AnimalDomestique, alors que cela n'a pas vraiment de sens. En effet, seules les classes en bout de chaîne (Chat, Chien, Cerf…) représentent vraiment des animaux concrets. L'idéal serait d'interdire l'instanciation de toutes les classes ancêtres. C'est précisément à ce besoin que répond la notion de classe abstraite.

Une classe abstraite est une classe qui ne peut pas être instanciée, et qui est destinée uniquement à être dérivée. Pour créer une classe abstraite, il suffit de mettre le mot clé **abstract** devant son nom.

Les méthodes et propriétés abstraites sont des membres de classes abstraites qui ne contiennent pas de corps, et qui sont destinés à être redéfinis dans les classes concrètes dérivées.

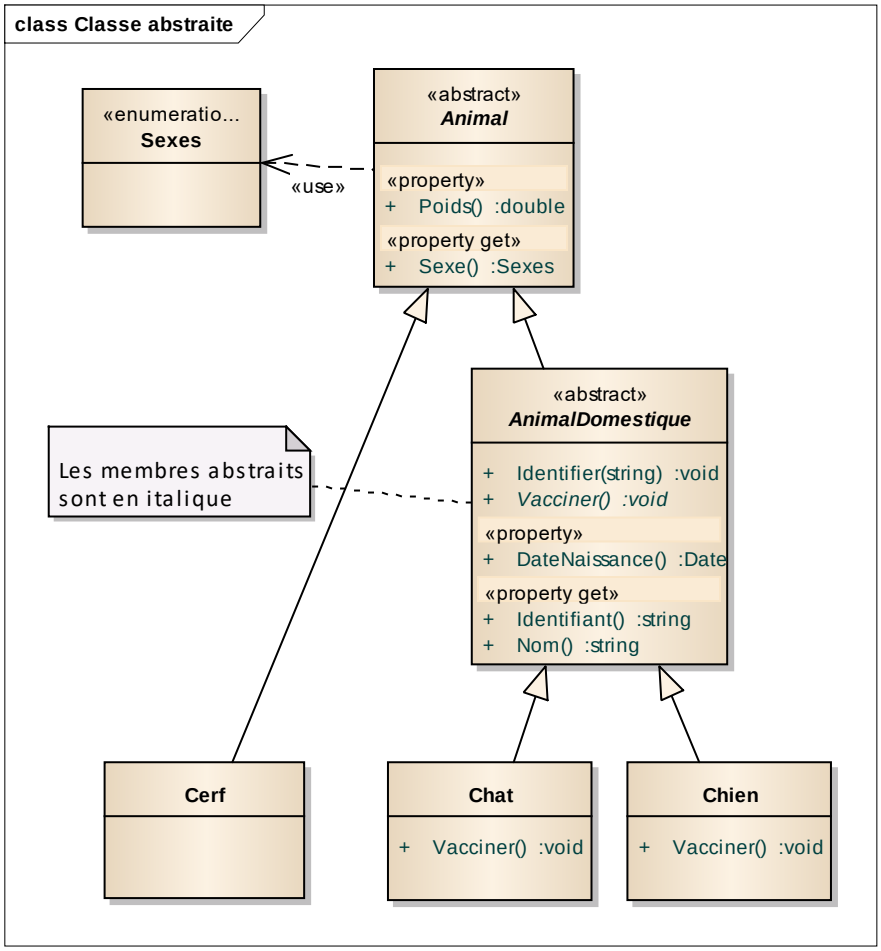
9.1.2 Caractéristiques

- Une classe abstraite peut contenir à la fois des méthodes concrètes, qui fournissent une implémentation par défaut, et des méthodes abstraites, sans corps.
- Une classe concrète ne peut pas contenir de membres abstraits
- Une méthode ou propriété abstraite doit **obligatoirement** être implémentée dans les classes concrètes dérivées.

Dans le framework .net, beaucoup de classes de haut niveau sont abstraites. Comme n'importe quelle classe, les classes abstraites peuvent implémenter des interfaces. C'est d'ailleurs un bon moyen de fournir à des classes concrètes, une implémentation par défaut d'une interface.

9.1.3 Exemple de mise en œuvre

On peut ainsi modéliser la hiérarchie de classes précédente de la façon suivante :



Animal et AnimalDomestique sont abstraites, car créer directement des instances de ces classes n'aurait pas beaucoup de sens. AnimalDomestique.Vacciner() est une méthode abstraite. Elle est donc redéfinie dans les classes concrètes Chat et Chien. Les autres membres de Animal et AnimalDomestique sont concrets, ce qui permet de fournir une implémentation par défaut. Mais rien n'empêche de les redéfinir dans les classes dérivées concrètes si besoin.

Voici le code correspondant à cette architecture :

```
using System;

namespace Exemples
{
    public enum Sexes { Male = 1, Femelle = 2 }

    // Classe ancêtre
    public abstract class Animal
    {
        public double Poids { get; set; }
        public Sexes Sexe { get; }

        public Animal() { }
        public Animal(Sexes sexe, double poids)
        {
            this.Sexe = sexe;
            this.Poids = poids;
        }
```

```
    }

    // Classe abstraite pour les animaux domestiques
    public abstract class AnimalDomestique : Animal
    {
        private string _identifiant;

        #region constructeurs
        public AnimalDomestique() { }
        public AnimalDomestique(Sexes sexe, double poids, string nom, DateTime
dateNais) : base(sexe, poids)
        {
            this.Nom = nom;
            this.DateNaissance = dateNais;
        }
        #endregion

        #region Propriétés
        public string Nom { get; }
        public DateTime DateNaissance { get; set; }
        public string Identifiant
        {
            get { return _identifiant; }
        }
        #endregion

        #region methodes
        public void Identifier(string numero)
        {
            // On vérifie le format du numéro identifiant
            // Il doit contenir l'année de naissance à partir du 8ème caractère
            string anneeNais = String.Empty;
            if (numero.Length >= 12) anneeNais = numero.Substring(7, 4);
            int annee;
            if (int.TryParse(anneeNais, out annee) && annee >= 1900 &&
                annee <= DateTime.Today.Year)
            {
                _identifiant = numero;
            }
            else
            {
                // Si le format est incorrect, on lève une exception
                throw new FormatException("numéro de tatouage incorrect");
            }
        }

        // Méthode abstraite. Ne possède pas de corps
        abstract public void Vacciner();
        #endregion
    }


    // Classes dérivées concrètes
    public class Chat : AnimalDomestique
    {
        public Chat() { }
        public Chat(Sexes sexe, double poids, string nom, DateTime dateNais) :
            base (sexe, poids, nom, dateNais) { }
```

```
        public override void Vacciner()
        {
            // ...
        }
    }

    public class Chien : AnimalDomestique
    {
        public Chien() { }
        public Chien(Sexes sexe, double poids, string nom, DateTime dateNais) :
            base(sexe, poids, nom, dateNais) { }

        public override void Vacciner()
        {
            // ...
        }
    }

    public class Cerf : Animal
    {
        public Cerf() { }
        public Cerf(Sexes sexe, double poids) : base(sexe, poids) { }
    }
}
```

Voici un exemple d'utilisation de ces classes :

```
class Program
{
    static void Main(string[] args)
    {
        Chien wolfy = new Chien(Sexes.Male, 10.8, "Wolfy", new DateTime(2011, 04, 14));
        wolfy.Identifier("FR48HNG201075314");

        Chat ponpon = new Chat();
        ponpon.Identifier("FR52LKS201479847");

        AnimalDomestique x = new AnimalDomestique(); // Incorrect, car
AnimalDomestique est abstraite

        Cerf c = new Cerf();
        c.Poids = 240;  // Correct car Poids est une propriété de Animal
        c.Identifier(); // Incorrect, car Cerf n'hérite pas de AnimalDomestique

        Console.ReadKey(true);
    }
```

9.2 Les interfaces

9.2.1 Définition

Une interface est un ensemble de membres non implémentés qui forment un *contrat*. Pour qu'une classe remplisse ce contrat (i.e. implémente l'interface), elle doit fournir une implémentation de tous les membres prévus par le contrat.

En ce sens, les interfaces ressemblent aux classes abstraites.

Caractéristiques

- Une interface peut contenir uniquement des méthodes, propriétés, événements et indexeurs. Elle ne peut pas contenir de constantes, champs, opérateurs, constructeurs d'instance, destructeurs ou types.
- Une interface ne peut pas être instanciée directement
- Une interface ne contient aucune implémentation. Ses méthodes n'ont pas de corps.
- Les membres d'une interface sont automatiquement publics et ne peuvent pas avoir de modificateur d'accès
- Une classe peut implémenter plusieurs interfaces (tout en dérivant d'une autre classe éventuellement)

L'exemple ci-dessous montre une interface nommée IClassable permettant de classer les êtres vivants.

```
public interface IClassable
{
    string Espece { get; }
    string Famille { get; }
    string Ordre { get;  }
    string Classe { get; }
    string Embranchement { get; }

    string GetClassification();
}
```

Le framework .NET fournit beaucoup d'interfaces. Parmi les plus classiques, on peut citer par exemple l'interface **`System.Collections.IEnumerable`** qui standardise le parcours des classes conteneurs (collection, liste, dictionnaire…). Une classe qui implémente cette interface permet entre autres de parcourir ses éléments au moyen de l'instruction **`foreach`** (cf. Section sur les conteneurs).

Il y a aussi l'interface IComparable, qui permet de comparer des objets afin de pouvoir les trier.

9.2.2 Implémentation

Une classe peut implémenter une interface de façon implicite, explicite ou abstraite. Visual Studio permet de choisir parmi les 3 modes, et de générer automatiquement la structure de code correspondante.

- **Implémentation implicite** : les membres issus de l'interface sont déclarés de façon classique. Si une classe implémente plusieurs interfaces avec des membres de même nom, l'implémentation ne peut pas être implicite, car ceci provoquerait des conflits de nommage.
- **Implémentation explicite** : le nom de chaque membre issu de l'interface est préfixé par le nom de l'interface, et on ne précise pas de modificateur d'accès. Ceci permet de résoudre les conflits de noms éventuels
- **Implémentation abstraite** : les membres issus de l'interface sont déclarés comme abstraits. Ceci n'est possible que si la classe est elle-même abstraite.

On peut choisir le mode d'implémentation pour chaque membre

Voici un exemple d'implémentation de l'interface précédente :

```
public enum Sexes { Male = 1, Femelle = 2 }
```

```
// Classe ancêtre
public abstract class Animal : IClassable
{
   public double Poids { get; set; }
   public Sexes Sexe { get; }

   // Interface IClassable
   // Implémentation abstraite des propriétés
   public abstract string Espece { get; }
   public abstract string Famille { get; }
   public abstract string Ordre { get; }
   public abstract string Classe { get; }
   public abstract string Embranchement { get; }
   // Implémentation explicite de la méthode
   string IClassable.GetClassification()
   {
      return Embranchement + "." + Classe + "." + Ordre + "." + Famille + "." +
Espece;
   }

   // Constructeurs
   public Animal() { }
   public Animal(Sexes sexe, double poids)
   {
      this.Sexe = sexe;
      this.Poids = poids;
   }
}

// Classe abstraite pour les animaux domestiques
public abstract class AnimalDomestique : Animal
{
   //...
}


// Classes dérivées
public class Chat : AnimalDomestique
{
   public Chat() { }
   public Chat(Sexes sexe, double poids, string nom, DateTime dateNais) :
         base(sexe, poids, nom, dateNais) { }

   // Propriétés issues de l'nterface IClassable
   public override string Espece { get { return "Chat"; } }
   public override string Famille { get { return "Félidé"; } }
   public override string Ordre { get { return "Carnivore"; } }
   public override string Classe { get { return "Mammifère"; } }
   public override string Embranchement { get { return "Vertébré"; } }
}

public class Chien : AnimalDomestique
{
   public Chien() { }
   public Chien(Sexes sexe, double poids, string nom, DateTime dateNais) :
         base(sexe, poids, nom, dateNais) { }

   // Propriétés issues de l'nterface IClassable
```

```
    public override string Espece { get { return "Chien"; } }
    public override string Famille { get { return "Canidé"; } }
    public override string Ordre { get { return "Carnivore"; } }
    public override string Classe { get { return "Mammifère"; } }
    public override string Embranchement { get { return "Vertébré"; } }
}

public class Cerf : Animal
{
    public Cerf() { }
    public Cerf(Sexes sexe, double poids) : base(sexe, poids) { }

    // Propriétés issues de l'nterface IClassable
    public override string Espece { get { return "Cerf"; } }
    public override string Famille { get { return "Cervidé"; } }
    public override string Ordre { get { return "Artiodactyle"; } }
    public override string Classe { get { return "Mammifère"; } }
    public override string Embranchement { get { return "Vertébré"; } }
}
```

La classe ancêtre Animal est abstraite et implémente l'interface IClassable. Comme les propriétés sont implémentées de façon abstraite, elles doivent être implémentées réellement dans les classes concrètes dérivées. En revanche, la méthode GetClassification est implémentée explicitement dans Animal.

9.2.3 Utilisation de variables de type interface

Considérons l'exemple de code ci-dessous :

```
IClassable mimine = new Chat();
Console.WriteLine("La famille du chat est : {0}", mimine.Famille);
mimine.Poids = 4.2; // Incorrect car la variable mimine n'est pas de type Animal
mais Iclassable
string c1 = mimine.GetClassification();

Chien capi = new Chien();
Console.WriteLine("La famille du chien est : {0}", capi.Famille);
capi.Poids = 12.8; // correct
string c2 = capi.GetClassification(); // incorrect car GetClassification est
implémentée explicitement
```

Il montre plusieurs choses intéressantes :

La première ligne montre qu'un objet qui implémente une interface peut être référencé au moyen d'une variable du type de l'interface.

Les 2ème et 3ème lignes montrent que cette variable ne permet par contre d'accéder qu'aux membres de l'interface, et pas aux membres de la classe qui l'implémente.

Une variable du type de la classe permet bien entendu d'accéder aux membres de l'interface (capi.Famille), sauf s'ils sont implémentés de façon explicite (capi.GetClassification()).

Pour accéder aux membres implémentés explicitement, il faut obligatoirement passer par une variable du type de l'interface (mimine.GetClassification()). C'est là une limitation de l'implémentation explicite d'interface. Cette limitation n'existe pas avec l'implémentation implicite.

9.2.4 Polymorphisme d'interface

Tous les objets qui implémentent IClassable peuvent être manipulés de la même façon via des variables de type IClassable, quel que soit le type d'objet référencé. Il s'agit d'un comportement polymorphique.

10 Types valeur et types référence

10.1 Allocation mémoire, pile et tas

Les variables nécessitent qu'on leur alloue de la place en mémoire. Il existe deux types d'allocations de mémoire :

- **L'allocation statique** : elle se fait au lancement du programme. Les performances sont optimales, puisqu'on évite les coûts de l'allocation dynamique durant l'exécution ; la mémoire statique est immédiatement utilisable. Elle stocke tous les champs et fonctions statiques (cf. plus bas)
- **L'allocation dynamique** : elle se fait pendant l'exécution du programme.

L'allocation dynamique peut être faite dans :

- **La pile** : c'est une zone mémoire dans laquelle l'allocation et la libération sont gérées automatiquement par le programme. Typiquement, une variable locale à l'intérieur d'une fonction est gérée dans la pile. Elle se voit allouer une zone mémoire au moment de son initialisation, et cette zone est automatiquement libérée à la sortie de la fonction.
- **Le tas** : c'est une zone mémoire dans laquelle l'allocation et la libération sont gérées par le ramasse-miettes

Différences majeures entre la pile et le tas (cf. cette page pour plus de détails) :

- Chaque thread a sa pile, tandis que le tas est partagé (1 par appli).
- La taille de la pile est déterminée quand le thread est créé, tandis que la taille du tas peut augmenter au cours de la vie de l'appli, selon ses besoins.
- La pile est plus rapide du fait de sa gestion LIFO (Last Input, First Output), et son contenu est fréquemment utilisé, ce qui tend en plus à utiliser le cache du processeur, qui est très rapide.

La pile et le tas sont tous deux gérés dans la RAM.

Le CLR .net fournit un **ramasse-miettes (Garbage Collector)**, qui permet au développeur de faire de l'allocation dynamique sur le tas, sans avoir à se soucier de la libération de la mémoire, ou presque, ce qui simplifie beaucoup la tâche, et évite les fuites mémoires.

10.2 Les types

En C, on choisit ce que l'on met dans la pile et ce que l'on met dans le tas.

En C#, c'est différent : les variables de type valeur sont créées dans la pile, tandis que les variables de type référence sont créées dans le tas. Une variable de type référence est une variable pour laquelle l'accès se fait via une référence.

Quels sont les types valeur ? Tous les types primitifs, les structures et les énumérations.

NB/ Les types primitifs (byte, char, int, bool …) sont en fait également des structures en C#

Quels sont les types référence ? Les classes et les délégués.

NB/ string est une classe, bien qu'elle s'utilise comme un type valeur

Quand on instancie une classe (i.e. on crée un objet) avec le mot clé new, on réserve de la mémoire dans le tas, et on récupère une référence sur l'emplacement réservé.

Quand la mémoire est-elle libérée ? Pour les variables de type valeur, à la fin du bloc de déclaration. Pour les objets de type référence, lorsqu'il n'y a plus aucune référence sur l'objet (il n'y a alors plus aucun moyen d'y accéder), le ramasse-miettes peut récupérer son emplacement en mémoire. Il ne le fait pas tout de suite, mais dès que la quantité de mémoire utilisée par l'application dépasse un certain seuil.

10.3Affectation

L'affectation n'a pas le même sens pour les variables de type valeur et celles de type référence.

Affectation pour les types valeur

Une variable de type valeur contient des données, et est allouée dans la pile, même si elle est initialisée avec new.

```
int i = 18;
int j = new int();
// Les variables i et j sont allouées dans la pile. j prend la valeur par défaut 0
```

…et l'affectation concerne la valeur de la variable.

```
int a=12,b=4; // a contient 12 et b contient 4
a = b;
// a contient désormais la valeur 4.
// la valeur de b a été affectée à la variable a
```

Affectation pour les types référence

L'allocation des variables est faite dans le tas, et la libération de l'espace mémoire est gérée par le ramasse-miettes (Garbage Collector).

On peut voir les références (les variables qui désignent les objets) comme des sortes de pointeurs sécurisés. Par exemple, une référence qui ne désigne aucun objet vaut **`null`**. C'est la valeur par défaut d'une référence non initialisée.

Pour les variables de type référence, l'affectation réalise seulement une copie des références (comparable à une copie de pointeurs en C++).

Ex : Considérons le code ci-dessous :

```
class Personne
{
    private string _nom;

    public void SetNom(string nom)
    {
        _nom = nom;
    }

    public string GetNom()
    {
        return _nom;
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Personne p1 = new Personne();
        p1.SetNom("Lenoir");

        Personne p2 = p1;   // p1 et p2 désignent la même personne
        p2.SetNom("Leblanc"); // modifie le nom de la personne

        Console.WriteLine(p1.GetNom()); // Affiche : Leblanc
        Console.WriteLine(p2.GetNom()); // Affiche : Leblanc
        Console.ReadKey();
    }
}
```

Puisque Personne est un type référence, après affectation de p1 à p2, p2 fait référence au même objet que p1. P1 et p2 font toutes deux références à la même zone mémoire. C'est pourquoi on peut modifier le nom de la personne via la variable p2.

10.4Test d'égalité

L'opérateur == pour les types valeur

Par défaut, le test d'égalité avec l'opérateur == sur des types valeur teste l'égalité de valeur des champs. Il y a comparaison de la valeur de tous les champs. L'opérateur renvoie la valeur true seulement si tous les champs ont les mêmes valeurs deux à deux.

L'opérateur == pour les types référence

Par défaut, le test d'égalité avec l'opérateur == teste si deux références désignent le même objet. Cependant, l'opérateur == peut être surchargé pour se comporter de la même façon que pour les types valeur. C'est par exemple ce qui est fait pour le type string : bien que ce soit un type référence, l'opérateur == teste l'égalité des chaînes (même longueur et mêmes caractères), et non celle des références.

10.5Synthèse

Le tableau suivant résume les différences entre les types valeur et les types référence :

Type valeur	Type référence
Stocké sur la pile	Stocké sur le tas
Contient la valeur effective	Contient une référence vers une valeur
Ne peut pas avoir la valeur null (à moins de spécifier un type nullable)	Peut contenir la valeur null
La mémoire est libérée lorsque l'application sort de la portée de la variable	La mémoire est libérée par le ramasse-miettes

10.6Structures et classes

Les classes sont un concept spécifique à la programmation orientée objet. Les structures sont un concept qui existe également dans certains langages non orientés objet (notamment le C).

Points communs :

- Peuvent contenir des champs, propriétés, méthodes et constructeurs
- Peuvent implémenter des interfaces

Différences :

- Les structures sont des types valeur, alors que les classes sont des types référence.
- Les structures ne peuvent pas être dérivées
- On ne peut pas créer de constructeur sans paramètre dans une structure
- Le compilateur n'initialise pas par défaut les champs d'une structure, alors qu'il le fait pour une classe.

Si l'on reprend l'exemple précédent en définissant Personne comme une structure et non comme une classe (il suffit de remplacer le mot class par struct), alors le même code produit un résultat différent :

```
class Program
{
    static void Main(string[] args)
    {
        Personne p1 = new Personne();
        p1.SetNom("Lenoir");

        Personne p2 = p1;   // Crée une personne p2 en copiant les champs de p1
        p2.SetNom("Leblanc"); // modifie le nom de p2

        Console.WriteLine(p1.GetNom()); // Affiche : Lenoir
        Console.WriteLine(p2.GetNom()); // Affiche : Leblanc
        Console.ReadKey();
    }
}
```

Puisque Personne est cette fois un type valeur, l'affectation de p1 à p2 crée une nouvelle personne et copie les valeurs des champs de p1 dans p2. P1 et p2 représentent bien deux personnes distinctes. C'est pourquoi la modification de p2 n'affecte pas p1.

Intérêt des structures

Le principal avantage des structures par rapport aux classes est leur accès mémoire plus rapide, du fait qu'elles sont stockées dans la pile. Cependant, lorsqu'une variable de type structure est passée en paramètre à une fonction, les données qu'elle contient sont dupliquées en mémoire, puisque le passage se fait par valeur. C'est pourquoi les structures ne devraient pas stocker plus de quelques octets de données pour que leur usage soit justifié. Les structures sont ainsi surtout utilisées dans la programmation de jeux vidéo, où la recherche de performance est un point essentiel.

Remarque : les types primitifs (byte, char, int, bool …) sont en fait des structures, qui fournissent des méthodes telles que ToString, Parse, TryParse, Equals…etc. Il est ainsi possible d'écrire : 7.ToString()

10.7Types nullables

La valeur **null** peut être assignée aux variables références. Elle signifie « aucune mémoire allouée ». Elle permet d'initialiser les variables de type référence avant de leur affecter une référence d'instance avec new.

```
Animal a = null;
a = new Animal();
```

Par défaut, on ne peut pas assigner null à une variable de type valeur. L'instruction suivante est par conséquent invalide en C# :

```
int i = null; // invalide
```

Cependant, C# définit le modificateur « ? » pour déclarer qu'une variable de type valeur est nullable. On peut alors lui assigner la valeur null :

```
int? i = null; // valide
if (i == null) // valide
{
  ...
```

On ne peut pas assigner directement la valeur d'une variable nullable à une variable non nullable.

```
int? i = 3;
int j = i; // invalide
```

En fait, un type nullable est une structure (cela reste donc bien un type valeur), qui possède deux propriétés publiques :

- **HasValue**, de type bool. Elle a la valeur true lorsque la variable contient une valeur différente de null.
- **Value** qui contient une valeur significative si HasValue est vraie, ou qui lève une exception InvalidOperationException dans le cas contraire.

Intérêt des types nullables

Pour les types numériques, il est parfois utile de faire la différence entre la valeur 0 et « pas de valeur ». Par exemple un score représenté par un entier peut être null si le joueur n'a pas encore joué, tandis que la valeur 0 signifiera que le joueur à réellement fait un score de 0.

Dans une base de données, les tables peuvent contenir des champs nullables. Si on récupère la valeur d'un champ de ce type dans une propriété C#, on peut définir cette propriété comme nullable de façon à pouvoir représenter la valeur Null issue de la base.

10.8Typage implicite avec var

Lorsqu'on instancie un objet, on peut utiliser la syntaxe suivante :

```
var obj = new MaClasse(20);
```

Avec var, le type de la variable obj n'est pas spécifié explicitement. Il est déduit implicitement par le compilateur, à partir du nom de la classe qui suit le mot clé new.

Avantage : dans le cas de noms de classes longs, éventuellement précédés du nom de leur espace de nom, cela raccourcit la syntaxe, et rend le code plus lisible :

```
Dictionary<string, Animal> dico = new Dictionary<string, Animal>();

// peut s'écrire plus simplement :
var dico = new Dictionary<string, Animal>();
```

var est également utilisable avec les types primitifs, mais l'intérêt est moindre. Cela rend au contraire le code moins facile à lire.

```
var x = 12.5; // x est implicitement défini par le compilateur comme un double
```

var n'est bien entendu utilisable que si l'on fournit une valeur d'initialisation (nécessaire au compilateur pour déduire le type).

/!\ var n'est qu'une facilité d'écriture. Il ne signifie pas que le type de la variable peut changer. Le type d'une variable ne peut en aucun cas changer après sa déclaration.

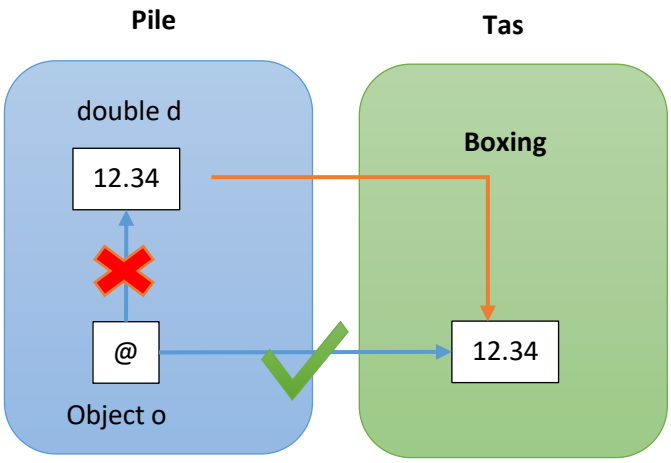
10.9Boxing / unboxing

En .NET, le type **System.Object** est la classe ancêtre de plus haut niveau, de laquelle dérivent implicitement toutes les classes, de façon directe ou indirecte. Comme une variable de classe ancêtre peut faire référence à une instance de classe dérivée, une variable de type Object peut donc faire référence à n'importe quel type référence (i.e. à n'importe quelle classe).

Le boxing étend ce concept en permettant à une variable de type Object de faire référence également à n'importe quel type valeur. Ainsi, il est possible d'écrire ce qui suit :

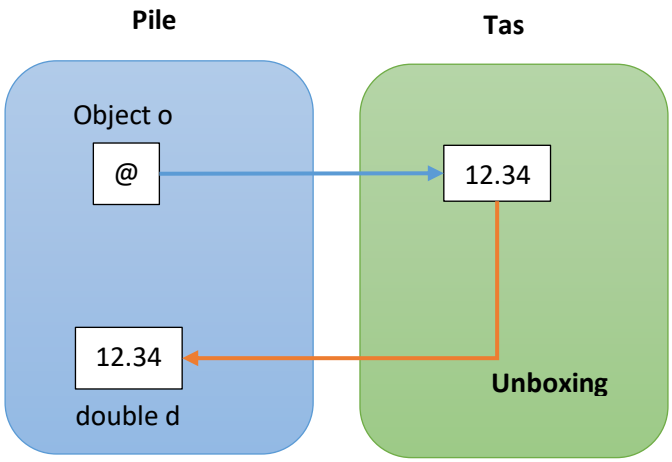
```
double d = 12.34;
Object o = d; // correct grâce au boxing
```

La deuxième instruction nécessite une explication pour comprendre ce qui se passe vraiment :
d est un type valeur qui est donc stocké sur la pile. Si la référence à l'intérieur de l'objet o pointait directement sur d, elle ferait référence à la pile, ce qui créerait un défaut de sécurité potentiel. Par conséquent, le runtime réserve un bloc de mémoire sur le tas, y copie la valeur du réel d, puis référence cette copie dans l'objet o. Cette copie automatique d'un élément de la pile vers le tas est appelée conversion boxing. Elle est illustrée par le schéma suivant :



L'unboxing est l'opération inverse du boxing. Elle consiste à convertir une copie boxed en type valeur. La conversion doit cette fois être explicite au moyen de la syntaxe suivante, qu'on appelle **cast** :

```
Object o = 12.34; // boxing
double d = (double)o; // unboxing
```



Que se passe-t-il si l'objet ne fait pas référence au bon type ?

Une variable Object pouvant contenir n'importe quoi, rien ne garantit que la conversion vers le type simple souhaité réussisse. Si elle échoue, cela provoque une exception InvalidCastException au moment de l'exécution.

Les conversions boxing et unboxing sont des opérations coûteuses en raison du nombre de vérifications nécessaires, et du besoin d'attribuer de la mémoire supplémentaire à partir du tas. La conversion boxing a son utilité, mais une utilisation abusive peut diminuer fortement les performances d'un programme. Les génériques sont un bon moyen d'éviter le boxing, comme nous le verrons plus loin.

11 La classe string

11.1Généralités

La classe **`System.String`** de .NET permet de gérer des chaînes de caractères. Il s'agit d'un type référence, mais dont l'usage ressemble à celui des types valeurs. Pour le type `string`, l'affectation (le signe =) fait en réalité une copie des valeurs, et non des références.
L'autre caractéristique est que les objets de cette classe sont ***immutables***. Cela signifie que les objets gardent la même valeur du début à la fin de leur vie. Toutes les opérations visant à changer la valeur de l'objet retourneront en réalité un nouvel objet.

Voici plusieurs façons de créer des instances de la classe string :

```
class Program
{
   static void Main(string[] args)
   {
      string s1 = "abc";
      string s2 = new string('*', 7);
      var ar = new char[] { 'e', 'f', 'g' }
```

```
        string s3 = new string(ar);

        Console.WriteLine(s1); // abc
        Console.WriteLine(s2); // *******
        Console.WriteLine(s3); // efg
    }
}
```

Les membres principaux de la classe **`string`** :

Propriétés

`[]` : indexeur qui permet d'accéder au caractère à la position spécifiée

`Static readonly string Empty` : Représente la chaîne vide

`int Length` : obtient le nombre de caractères

Méthodes

`bool EndsWith(string value)` : renvoie vrai si la chaîne se termine par value
`bool StartsWith(string value)` : renvoie vrai si la chaîne commence par value

`virtual bool Equals(object obj)` : renvoie vrai si la chaîne est égale à obj

`static string Format(string format, params object[] args)` : remplace un ou plusieurs éléments de mise en forme d'une chaîne par la représentation sous forme de chaîne d'un objet spécifié

`static bool IsNullOrEmpty(string value)` : indique si la chaîne spécifiée est null ou une chaîne String.Empty

`int IndexOf(string value, int startIndex)` : renvoie la première position de value dans la chaîne. La recherche commence à partir du caractère n° startIndex
`int IndexOf(char value, int startIndex)` : idem, mais pour le caractère value
`int LastIndexOf(string value, int startIndex)` : renvoie la dernière position de value dans la chaîne. La recherche commence à partir du caractère n° startIndex.

`string Replace(string oldValue, string newValue)` : renvoie une chaîne dans laquelle la chaîne oldValue a été remplacée par la chaîne newValue

`string[] Split(char[] separator)` : la chaîne est vue comme une suite de champs séparés par les caractères présents dans le tableau separator. Le résultat est le tableau de ces champs

`string Substring(int startIndex, int length)` : sous-chaîne de la chaîne courante commençant à la position startIndex et ayant length caractères

`string ToLower()` : renvoie la chaîne courante en minuscules
`string ToUpper()` : renvoie la chaîne courante en majuscules
`string Trim()` : supprime les espaces blancs en début et en fin de chaîne

```
static void Main(string[] args)
{
    string s1 = "chaîne de caractères";
    Console.WriteLine(s1.Contains("car"));  // True
    Console.WriteLine(s1.StartsWith("ch")); // True
```

```
    Console.WriteLine(s1.IndexOf('c'));     // 0
    Console.WriteLine(s1.LastIndexOf('c')); // 14
    Console.WriteLine(s1.Substring(4, 2));  // ne
}
```

Remarque : Pour vérifier si une chaîne est vide, utiliser la méthode ci-dessus ou la comparaison avec String.Empty (si on est sûr que la valeur n'est pas null). Ne **pas** utiliser la comparaison avec "", qui est moins performante.

11.2 Méthode ToString et formats

Tout objet dérive de la classe Object, et possède donc une méthode ToString(). L'implémentation par défaut de cette méthode se contente de renvoyer le nom de la classe de l'objet (peu intéressant).

Les classes qui redéfinissent la méthode ToString() font généralement en sorte que la méthode renvoie une représentation plus significative de l'objet. Par exemple, une classe Personne redéfinira ToString pour qu'elle renvoie une chaîne composée du prénom et du nom de la personne.

Il peut cependant être nécessaire de spécifier en plus un format pour cette chaîne, notamment si l'objet contient des nombres ou des dates. Pour ce faire, la classe de l'objet doit implémenter l'interface **`IFormattable`**, qui contient une surcharge de la méthode ToString prenant 2 paramètres :

```
interface IFormattable
{
    string ToString(string format, IFormatProvider formatProvider)
}
```

Paramètres :

- **`format`** : chaîne fournissant des instructions de formatage, constituée d'une lettre et d'un nombre optionnel.
- **`formatProvider`** : objet permettant de réaliser les instructions de formatage selon une culture donnée

Le code ci-dessous affiche un prix avec un format, et dans plusieurs cultures :

```
Console.OutputEncoding = Encoding.Unicode;
decimal prix = 3.5m;

string s0 = prix.ToString("C2");
Console.WriteLine(s0);

string s1 = prix.ToString("C2", CultureInfo.GetCultureInfo("en-US"));
Console.WriteLine(s1);

string s2 = prix.ToString("C2", CultureInfo.InvariantCulture);
Console.WriteLine(s2);
```

Sortie console :

```
3.50 €
$3.50
¤3.50
```

Le format « C2 » indique que le nombre doit être formaté en monnaie (Currency), avec 2 chiffres après la virgule.

Dans le 1er cas, comme on ne spécifie aucune culture, c'est la culture par défaut (CurrentCulture) qui est utilisée. Celle-ci est définie par les paramètres régionaux du système d'exploitation.

Dans le 2ème cas, on a spécifié explicitement la culture « en-US ». Le prix est donc affiché en dollars.

Dans le 3ème cas, on a spécifié la culture Invariant, qui est indépendante de toute langue, tout pays, et tout paramétrage du poste.

Chaînes de formats possibles

Il existe un ensemble de chaînes de formats prédéfinies pour chaque type. Exemples :

- Pour les nombres : C = Currency, D = Decimal, G = général, F = fixed point…etc.
- Pour les dates : d = date courte, D = date longue, f = date et heure courtes…etc.
- Pour les types énumérés : G et F affichent la valeur textuelle, D affiche la valeur numérique

En plus de formats prédéfinis, il est possible de définir des formats personnalisés.

Exemples :

```
DateTime date = new DateTime(2016, 07, 14, 13, 58, 35);
Console.WriteLine(date.ToString("d"));
Console.WriteLine(date.ToString("F"));
Console.WriteLine(date.ToString("ddd %d/%M/yy"));
```

Sortie console :

```
14/07/2016
jeudi 14 juillet 2016 13:58:35
jeu. 14/7/16
```

On a appliqué ici successivement trois formats différents à la même date. Les deux premiers sont des formats prédéfinis, et le 3ème est un format personnalisé.

L'ensemble des formats possibles est présenté dans la rubrique « Formatting Types » de la doc Microsoft, dont la page principale est ici.

11.3 Méthode Format

Cette méthode permet de construire des chaînes de caractères formatées contenant des valeurs paramétrables.

```
var s3 = string.Format(" Un réel : {0}\n Une chaîne : {1}\n Une date : {2}",
                       123.1548, "coucou", DateTime.Now);
Console.WriteLine(s3);
```

Sortie console :

```
 Un réel : 123,1548
 Une chaîne : coucou
 Une date : 21/09/2016 19:24:53
```

Format accepte un nombre variable de paramètre de n'importe quel type, grâce au fait qu'elle utilise un tableau d'objets params. C'est un concept que nous verrons dans le chapitre sur les tableaux.

{0], {1}…{N} représentent les paramètres. On doit bien entendu fournir à la fonction autant de valeurs de paramètres qu'il y a de paramètres dans la chaîne.
La chaîne contenant les paramètres est appelée « **chaîne de format composite** ».
La méthode Format appelle la méthode ToString sur chacun des paramètres qu'on lui passe.

Format possède une surcharge qui prend en premier paramètre la culture à utiliser. De plus, on peut préciser finement les formats de chaque comme le montre l'exemple suivant :

```
var s4 = string.Format(CultureInfo.GetCultureInfo("en-US"),
                       " Un réel : {0,12:C2}\n Une chaîne : {1}\n Une date : {2:d}",
                        123.1548, "coucou", DateTime.Now);
Console.WriteLine(s4);
```

Sortie console :

```
 Un réel :      $123.15
 Une chaîne : coucou
 Une date : 2/11/2017
```

Dans cet exemple :

- Le premier paramètre spécifie la culture en-US, c'est-à-dire la langue anglaise et le pays USA.
- Le format {0,12:C2} indique que le paramètre N°1 sera affiché sur 12 caractères sous le format monétaire avec 2 chiffres après la virgule.

Si on spécifie le nombre de caractères, le texte sera para défaut aligné à droite, comme le montre la sortie console. En spécifiant une valeur négative, il sera aligné à gauche.

Remarques :

- La méthode Console.WriteLine se comporte comme String.Format, à ceci près qu'on ne peut pas spécifier la culture. On peut donc écrire par exemple :

```
Console.WriteLine (" Un réel : {0:C1}\n Une chaîne : {1}\n Une date : {2:d}",
                    123.1548, "coucou", DateTime.Now);
```

- C#6 introduit la notion de **chaine interpolée**, grâce à laquelle il est possible de construire une chaîne paramétrée de la façon suivante :

```
var s = String.Format($"Le chien s'appelle {nom}, et il a {age} ans");
```

« nom » et « age » sont directement des noms de variables. Cette notation est plus concise mais ne permet cependant pas de préciser les formats des variables.

- Pour afficher certains caractères spéciaux dans la console, il peut être nécessaire de changer le jeu de caractères à utiliser, comme par exemple UTF8. Voici la syntaxe :

```
Console.OutputEncoding = Encoding.UTF8;
```

12 Les classes conteneurs

Le framework .NET fournit des classes pour stocker des données. Nous décrivons sommairement quelques classes d'usage fréquent.

12.1Les tableaux

En C#, les tableaux sont des objets dérivés de la classe abstraite System.Array.

Un tableau contient des éléments de même type, accessibles par un index entier, de base 0.

On peut créer des tableaux d'éléments de tous types, valeurs ou références.

```
int[] t1, t2; // déclaration de références (aucune allocation mémoire pour les éléments
à ce stade)
t1 = new int[3]; // instanciation. Les éléments sont initialisés par défaut à 0
t2 = new int[] {1, 2, 3}; // instanciation + initialisation

Animal[] anx = new Animal[5]; // déclaration + instanciation. Les éléments sont
initialisés à null
```

Lorsqu'on initialise directement les éléments du tableau, on peut omettre le mot clé new et la dimension du tableau :

```
int[] t3 = {2, 6}; // déclaration + instanciation + initialisation explicite
TimeSpan[] programmation = { new TimeSpan(12, 30, 59), new TimeSpan(5, 10, 0) };
```

La taille du tableau peut être définie dynamiquement :

```
Int size = int.Parse(Console.ReadLine());
Animal[] anx = new Animal[size];
```

…mais cette taille ne peut pas être modifiée après coup (le tableau n'est pas redimensionnable).

Parcours d'un tableau avec une boucle for

```
int[] tab = new int[3];
for (int i = 0; i < tab.Length; i++)
{
   tab[i] = i + 1;
   Console.WriteLine(tab[i]);
}
```

Length est une propriété de la classe System.Array qui donne le nombre d'éléments du tableau

Parcours d'un tableau avec une instruction foreach :

```
int[] tab = new int[3];
foreach(int val in tab)
{
   Console.WriteLine(val);
}
```

L'instruction **foreach** est le moyen le plus simple de parcourir un tableau, mais elle à plusieurs limitations :

- On ne peut pas parcourir seulement une partie du tableau
- On ne peut pas parcourir le tableau dans l'ordre inverse (du dernier élément au premier)
- On ne connaît pas l'index de l'élément courant

- On ne peut pas modifier ou supprimer l'élément courant ; il est en lecture seule

Copie d'un tableau

Un tableau étant un type référence, si on veut le copier avec son contenu, on ne peut pas se contenter de faire :

```
int[] t = {1, 2, 3};
int[] c = t;
```

Il faut plutôt utiliser la méthode CopyTo, fournie par la classe Array :

```
int[] t = {1, 2, 3};
int[] c = new int[t.Length];
t.CopyTo(c,  0); // copie en commençant à l'index 0
```

On est obligé d'instancier le tableau dans lequel on veut copier avant d'appeler CopyTo. On peut choisir de ne copier qu'une partie des éléments.

Tableau à plusieurs dimensions

Juste à titre indicatif, voici un exemple de tableau à plusieurs dimensions :

```
int[, ,] cube = new int[3, 4, 3];
cube[0,0,0] = 1;
Console.WriteLine(cube.Length); // 36
Console.WriteLine(cube.GetLength(0)); // taille de la première dimension
Console.WriteLine(cube.Rank); // nombre de dimensions = 3
```

Tableau params

Une méthode peut avoir un paramètre de type tableau. Dans ce cas, il faut lui passer un tableau en paramètre lorsqu'on l'appelle. Mais en ajoutant le mot clé **params** devant le nom du paramètre, il devient inutile de créer soi-même le tableau. Exemple :

```
class Util
{
   // Méthode qui retourne la valeurminimale d'une liste d'entiers
   public static int Min(params int[] valeurs)
   {
      if (valeurs == null || valeurs.Length == 0)
         throw new ArgumentException("Util.Min : il n'y a pas assez d'arguments");

      int valeurMin = valeurs[0];
      foreach(int i in valeurs)
      {
         if (i < valeurMin)
            valeurMin = i;
      }
      return valeurMin;
   }
}
```

La méthode Min s'utilise de la façon suivante :

```
int m = Util.Min(45, 12, 68, 954, 21, 7, 47);
```

```
Console.WriteLine(m);
```

On voit qu'on peut passer directement la liste des valeurs à comparer en paramètre de la méthode. Il n'est pas nécessaire de créer un tableau. Ceci est rendu possible grâce au mot clé params. Il permet à la méthode d'accepter un nombre quelconque de paramètres.

Quand de plus le tableau params est de type Object, la méthode est capable d'accepter n'importe quel nombre et n'importe type de paramètres. C'est cette technique qui est utilisée dans les méthodes String.Format et Console.Writeline, que nous avons déjà utilisées plusieurs fois.

12.2 Les collections non génériques

Les collections sont un autre moyen de stocker des données et de les parcourir. A la différence des tableaux, les éléments d'une collection sont toujours stockés sous forme d'**objets**, même s'il s'agit à la base de types valeur. Le mécanisme de boxing intervient donc lorsqu'on ajoute des éléments à la collection, et le mécanisme d'unboxing intervient lorsqu'on veut récupérer la valeur d'un élément de type simple.

Il existe plusieurs sortes de collections définies dans l'espace de noms System.Collections, qui diffèrent par la façon d'ajouter des éléments, d'y accéder, et de les ranger en mémoire. Nous les présentons ici de façon succincte :

ArrayList : se comporte comme un tableau, avec des possibilités supplémentaires : on peut ajouter ou supprimer un élément au milieu de la collection, avec les méthodes Insert et RemoveAt.

Queue : implémente une file d'attente, selon le mécanisme de premier entré, premier sorti (FIFO). Les éléments sont ajoutés avec la méthode Enqueue et enlevés avec la méthode Dequeue

Stack : implémente une pile, selon le mécanisme du dernier entré, premier sorti (LIFO). Les éléments sont ajoutés avec la méthode Push et enlevés avec la méthode Pop.

Hashtable : semblable à un tableau dans lequel les éléments ne sont pas indexés par un entier, mais par un objet de n'importe quel type (string, double DateTime…), qu'on appelle clé. Chaque clé doit être unique. Cette collection gère en interne un tableau pour les clés, et un autre pour les valeurs.

SortedList : similaire à une Hastable dans laquelle les clés sont triées. Lors de l'ajout d'un élément, celui-ci est placé au bon endroit pour maintenir la liste triée.

On pourra consulter plus en détails les méthodes de chaque collection sur cette page MSDN.

Voici juste un exemple de mise en œuvre d'une ArrayList :

```
using System;
using System.Collections;

namespace Exemples
{
    class Collections
    {
        public void TestArrayList()
        {
            ArrayList ar = new ArrayList();
            for (int i=0; i<10; i++)
            {
                // utilisation comme un tableau ordinaire
```

```
                ar[i] = "Elément " + i.ToString();
            }

            ar.Add("Elément x"); // ajout d'un élément à la fin du tableau
            ar.Insert(8, "Elément y"); // Ajout d'un élément au milieu du tableau
            ar.RemoveAt(5); // Suppression d'un élément au milieu du tableau
        }
    }
}
```

Différences entre tableau et collection

Les différences fondamentales entre un tableau et une collection sont :

- Un tableau déclare le type des éléments qu'il contient, alors qu'une collection ne le fait pas (les éléments sont stockés en tant qu'Object).
- Un tableau a une taille fixe, alors que la taille d'une collection s'adapte dynamiquement à son contenu.
- Un tableau peut avoir plusieurs dimensions, alors qu'une collection est linéaire

12.3Introduction aux génériques

Le problème des collections d'objets

Nous avons vu que les variables de type Object pouvaient stocker n'importe quel type de donnée, valeur ou référence, grâce au fait que :

- Toutes les classes dérivent de Object
- Les types valeurs peuvent être stockés automatiquement dans des objets grâce au mécanisme de boxing

Les collections de l'espace de noms System.Collections permettent ainsi de stocker n'importe quoi, et rien n'empêche d'y mettre des éléments de natures différentes.

Ce système et très souple pour le stockage des données, mais il pose ensuite problème pour leur utilisation. En effet, nous avons vu que pour manipuler les types sous-jacents, il fallait transtyper (convertir) les objets dans le type souhaité.
D'une part, c'est une opération coûteuse en performance. D'autre part, elle n'est pas sécurisée, car même si l'on s'attend à ce que l'objet soi du type sous-jacent souhaité, rien ne permet de le garantir. Il faut alors gérer cette éventualité par le code au moyen d'exceptions et/ou des opérateurs is et as si on veut empêcher l'application de planter. Mais cette solution n'est pas satisfaisante. Il serait préférable de traiter le problème à la racine, en disposant de collections qui n'autorisent le stockage que d'éléments du même type. C'est là qu'intervient la généricité.

Les classes génériques acceptent des paramètres de type, qui spécifient le type des objets sur lequel elles opèrent. En particulier, le .NET Framework fournit de nombreuses classes de collections et interfaces génériques dans l'espace de noms System.Collections.Generic

Voici un exemple simple de classe générique, qui modélise un point du plan dont le type des coordonnées est paramétrable :

```
namespace Generique
```

```
{
   class Point<T>  // classe générique : T est un type paramétrable
   {
      private T _x, _y;  // les deux champs sont du même type
      public Point(T x, T y)  // constructeur
      {
         _x = x;
         _y = y;
      }

      public override string ToString()
      {
         return "(" + _x.ToString() + "," + _y.ToString() + ")";
      }
   }
}
```

Le type est passé en paramètre à la classe au moyen de la syntaxe **<T>**. T est un paramètre qui sera remplacé par un type réel au moment de la compilation. Le paramètre T peut être utilisé à l'intérieur de la classe comme type de champ, de propriété ou d'argument de méthode. Dans l'exemple ci-dessus, il est utilisé pour les deux champs privés, et pour les paramètres du constructeur.

Cette classe s'utilise de façon simple comme le montre l'exemple ci-dessous :

```
namespace Generique
{
   class Program
   {
      static void Main(string[] args)
      {
         // Point avec des coordonnées entières
         Point<int> p1 = new Point<int>(7, 2);
         Console.WriteLine(p1.ToString());

         // Point avec des coordonnées décimales
         Point<decimal> p2 = new Point<decimal>(1.35m, 3.54m);
         Console.WriteLine(p2.ToString());
      }
   }
}
```

```
Sortie console :
(7,2)
(1.35,3.54)
```

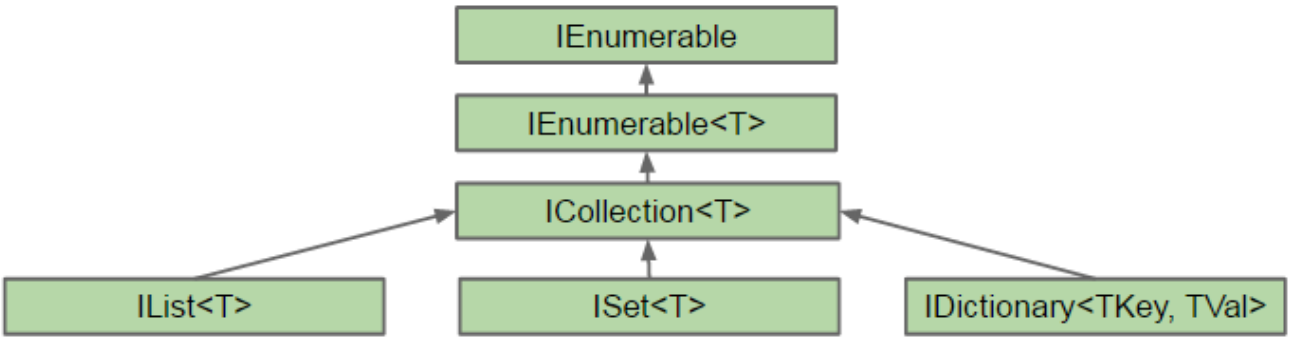
On voit dans cet exemple qu'il n'y a besoin d'aucun transtypage pour gérer les différents types de paires. Il suffit de préciser le type réel à utiliser entre <> juste après le nom de la classe. En interne, le compilateur génèrera une version de la classe Point pour chaque type réel demandé.

12.4Interfaces génériques de collections

Un problème que se pose souvent le développeur est de déterminer quelle collection est la plus adaptée à un besoin particulier.

Pour aider à faire ce choix, il est souvent pertinent de regarder en premier quelle(s) interface(s) doit implémenter la collection, et ensuite de choisir une collection parmi celles qui implémentent cette interface.

Voici un diagramme qui résume la hiérarchie des principales interfaces implémentées par les collections génériques.



Remarque : pour voir les membres de chaque interface, il suffit de saisir son nom dans Visual Studio et de presser F12 (atteindre la définition). Le détail des interfaces les plus usuelles est donné ci-dessous à titre indicatif :

IEnumerable<T> est l'interface de base pour les collections génériques. Elle contient une seule méthode non statique nommée GetEnumerator, qui retourne un IEnumerator<T>. Elle contient en revanche beaucoup de méthodes d'extensions.

Les collections qui implémentent IEnumerable<T> peuvent être parcourues à l'aide de l'instruction **foreach**.

ICollection<T> :

```
namespace System.Collections.Generic
{
   public interface ICollection<T> : IEnumerable<T>, IEnumerable
   {
      int Count { get; }
      bool IsReadOnly { get; }
      void Add(T item);
      void Clear();
      bool Contains(T item);
      void CopyTo(T[] array, int arrayIndex);
      bool Remove(T item);
   }
}
```

IList<T> permet d'atteindre des éléments par un index (comme un tableau), et d'ajouter ou supprimer des éléments au milieu de la liste

```
namespace System.Collections.Generic
{
   public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
   {
      T this[int index] { get; set; } // Indexeur
      int IndexOf(T item); // donne l'index de item ou -1 sinon
      void Insert(int index, T item); // insertion de item à l'index
      void RemoveAt(int index); // supprime l'élment à index
   }
}
```

IDictionary<TKey,TValue> est l'interface pour les paires (clé, valeur) avec clés uniques. On appelle parfois cela des tableaux associatifs. On peut accéder aux éléments à partir de leurs clés.

```
namespace System.Collections.Generic
{
   public interface IDictionary<TKey, TValue> : ICollection<KeyValuePair<TKey,
TValue>>, IEnumerable<KeyValuePair<TKey, TValue>>, IEnumerable
   {
      TValue this[TKey key] { get; set; }
      ICollection<TValue> Values { get; }
      void Add(TKey key, TValue value);
      bool ContainsKey(TKey key);
      bool Remove(TKey key);
      bool TryGetValue(TKey key, out TValue value);
   }
}
```

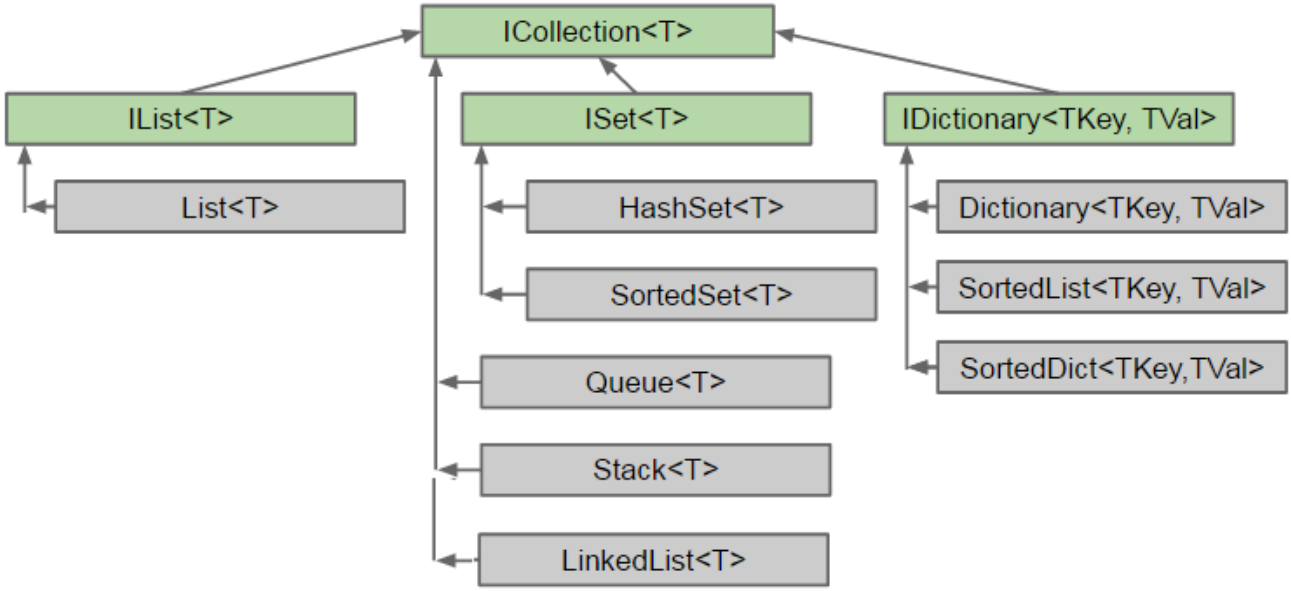
ISet<T> est l'interface de base pour les ensembles

```
namespace System.Collections.Generic
{
   public interface ISet<T> : ICollection<T>, IEnumerable<T>, IEnumerable
   {
      bool Add(T item);
      // Supprime de l'ensemble actuel tous les éléments de la collection spécifiée.
      void ExceptWith(IEnumerable<T> other);
      // Modifie l'ensemble actuel afin qu'il contienne uniquement les éléments qui
figurent également dans une collection spécifiée.
      void IntersectWith(IEnumerable<T> other);
      // Détermine si le jeu en cours est un sous-ensemble approprié (strict) d'une
collection spécifiée.
      bool IsProperSubsetOf(IEnumerable<T> other);
      // Détermine si le jeu en cours est un sur-ensemble (strict) correct de la
collection spécifiée.
      bool IsProperSupersetOf(IEnumerable<T> other);
      // Détermine si un ensemble est un sous-ensemble d'une collection spécifiée.
      bool IsSubsetOf(IEnumerable<T> other);
      // Détermine si l'ensemble actuel est un super-ensemble d'une collection
spécifiée.
      bool IsSupersetOf(IEnumerable<T> other);
      // Détermine si l'ensemble actuel recoupe la collection spécifiée.
      bool Overlaps(IEnumerable<T> other);
      // Détermine si l'ensemble actuel et la collection spécifiée contiennent les
mêmes éléments.
      bool SetEquals(IEnumerable<T> other);
      // Modifie l'ensemble actuel afin qu'il contienne uniquement les éléments
présents dans l'ensemble actuel ou dans la collection spécifiée, mais pas dans les
deux à la fois.
      void SymmetricExceptWith(IEnumerable<T> other);
      // Modifie le jeu en cours pour qu'il contienne tous les éléments qui sont
présents dans le jeu actuel ou la collection spécifiée.
      void UnionWith(IEnumerable<T> other);
   }
}
```

Il existe aussi les interfaces IReadOnlyCollection<T>, IReadOnlyList<T> et IReadOnlyDictionary<TKey, TValue>, qui sont les pendants des interfaces vues précédemment, mais avec beaucoup moins de méthodes puisqu'elles gèrent des collections en lecture seule.

12.5Collections génériques

Le diagramme ci-dessous présente la hiérarchie des collections génériques avec les interfaces qu'elles implémentent :



Voici quelques précisions à propos de certaines de ces collections :

List<T> représente une liste d'objets accessibles par index. Elle fournit des méthodes de recherche, de tri et de manipulation de listes. C'est certainement la collection la plus utilisée.

LinkedList<T> est une liste doublement chaînée ; c'est-à-dire que chaque élément connaît le précédent et le suivant. Cette classe n'implémente pas `IList<T>` puisque l'on ne peut pas accéder directement à un élément par index.

Les éléments sont stockés dans des nœuds décrits par la classe ci-dessous :

```
namespace System.Collections.Generic
{
   public sealed class LinkedListNode<T>
   {
      public LinkedListNode(T value);
      public LinkedList<T> List { get;
      public LinkedListNode<T> Next { get; }
      public LinkedListNode<T> Previous { get; }
      public T Value { get; set; }
   }
}
```

Dictionary<TKey,TValue>

Classe générique de tableaux associatifs implémentant l'interface `IDictionary<key,value>`.

```
namespace System.Collections.Generic
{
   public class Dictionary<TKey, TValue> : IDictionary<TKey, TValue>,
             ICollection<KeyValuePair<TKey, TValue>>,
```

```
            IEnumerable<KeyValuePair<TKey, TValue>>,
            IDictionary, ICollection, IEnumerable, Iserializable,
            IDeserializationCallback
    {
        public Dictionary();
        public Dictionary(IDictionary<TKey, TValue> dictionary);

        public int Count { get; }
        public Dictionary<TKey, TValue>.KeyCollection Keys { get; }
        public Dictionary<TKey, TValue>.ValueCollection Values { get; }
        public TValue this[TKey key] { get; set; }
        public void Add(TKey key, TValue value);
        public void Clear();
        public bool ContainsKey(TKey key);
        public Dictionary<TKey, TValue>.Enumerator GetEnumerator();
        public bool Remove(TKey key);
        public bool TryGetValue(TKey key, out TValue value);
    }
}
```

SortedList<TKey, TValue> et SortedDictionary<TKey, TValue> sont similaires. Leurs différences concernent la charge mémoire et les performances :

- SortedList utilise moins de mémoire
- SortedDictionary a des opérations d'insertion et de suppression plus rapides pour les données non triées
- Si la liste est remplie en une seule fois à partir de données triées, SortedList est plus rapide.

12.6 Enumérations des collections

Nous avons vu plus haut que, pour être énumérable, une classe conteneur doit implémenter l'interface **`System.Collections.IEnumerable`**. On peut alors parcourir ses éléments au moyen de l'instruction **`foreach`**. Pour être plus précis, ce parcours est rendu possible grâce à un objet de type IEnumerator, renvoyé par la méthode GetNumerator.

La création d'un objet implémentant l'interface IEnumerator est un peu fastidieuse. C'est pourquoi .net propose la notion d'itérateur, qui permet de faire générer automatiquement un IEnumerator par le compilateur.

Un itérateur est un bloc de code qui effectue une itération au sein du conteneur, c'est-à-dire qui renvoie successivement les éléments du conteneur à chaque appel. Il contient pour cela une instruction **yield return**, qui mémorise la position à laquelle il s'est arrêté, et reprend l'exécution du code à partir de cette position au prochain appel.

Examinons l'exemple suivant (issu de cette page MSDN) :

```
static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();

    foreach (string day in days)
    {
        Console.Write(day + " ");
    }
    // Output: Sun Mon Tue Wed Thu Fri Sat
```

```
    Console.ReadKey();
}

public class DaysOfTheWeek : IEnumerable
{
    private string[] days = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

    public IEnumerator GetEnumerator()
    {
        for (int index = 0; index < days.Length; index++)
        {
            // Yield each day of the week.
            yield return days[index];
        }
    }
}
```

DaysOfTheWeek est une classe conteneur pour les jours de la semaine. Elle contient un itérateur, qui est le bloc de code de la boucle for. Cette boucle contient l'instruction yield return, qui renvoie successivement chaque jour de la semaine, à chaque fois que l'élément suivant de la collection est demandé par la boucle foreach de la méthode main.

A partir de ce code, le compilateur est capable de générer lui-même un objet de type IEnumerator qui sera renvoyé par la méthode GetEnumerator.

L'appel à GetEnumerator lors de l'utilisation de l'instruction foreach est également réalisé implicitement par le compilateur.

Remarque : la compréhension du fonctionnement d'un itérateur est un peu délicate, justement à cause du fait qu'une bonne partie du code n'est pas visible, car générée par le compilateur.

13 Délégués et événements

13.1Délégués

Un délégué est un représentant d'une ou plusieurs méthodes. Lorsqu'on exécute un délégué, on exécute en fait les méthodes représentées par ce délégué.

Les délégués permettent de passer des méthodes en paramètre d'autres méthodes, et sont utilisés pour découpler certains traitements de leur code appelant. Ils sont notamment utilisés pour exécuter des gestionnaires d'évènements. Le .net framework fournit également beaucoup de méthodes utilisant des délégués pour réaliser des opérations sur les collections. L'exemple suivant illustre la méthode FindAll, qui permet de rechercher les éléments d'une liste qui correspondent à une condition :

```
class Program
{
    static void Main(string[] args)
    {
        List<Animal> animaux = new List<Animal>();
        animaux.Add(new Animal { Nom = "Souris", DureeVie = 1.5 });
        animaux.Add(new Animal { Nom = "Tortue", DureeVie = 110 });
        animaux.Add(new Animal { Nom = "Chien", DureeVie = 12 });
```

```
        animaux.Add(new Animal { Nom = "Vache", DureeVie = 17 });
        animaux.Add(new Animal { Nom = "Corbeau", DureeVie = 50 });

        List<Animal> lst = animaux.FindAll(ADureeVieElevee);
    }

    private static bool ADureeVieElevee(Animal a)
    {
        return a.DureeVie >= 50;
    }
}
```

La méthode FindAll a l'en-tête suivante :

```
public List<T> FindAll(Predicate<T> match)
```

Predicate<t> est un prédicat, c'est à dire un délégué prenant en paramètre un élément de type T, et renvoyant un booléen :

```
public delegate bool Predicate<in T>(T obj);
```

La méthode ADureeVieElevee satisfait ce délégué et peut donc être utilisée pour définir le critère de la recherche.
La méthode FindAll implémente l'algorithme de recherche, qui consiste à vérifier pour chaque élément de la collection s'il satisfait ou non une condition, tandis que cette condition est fournie par une fonction externe. On a ainsi un découplage complet entre l'algorithme de recherche et la condition à vérifier, ce qui permet d'utiliser l'algorithme avec toutes sortes de conditions différentes.

13.1.1 Création et exécution d'un délégué

Dans l'exemple qui suit, la méthode VisiteVeterinaire modélise une visite vétérinaire sur un animal, qui peut comprendre plusieurs actes médicaux :

```
public class Animal
{
    public string Nom { get; set; }
    public double DureeVie { get; set; }
    public List<string> CarnetSanté { get; }

    public Animal()
    {
        CarnetSanté = new List<string>();
    }

    public void VisiteVeterinaire(DelegueActesMedicaux actes)
    {
        // Appelle chaque méthode représentée par le délégué sur l'animal courant
       actes(this);
    }
}

// Type délégué pour les actes médicaux
public delegate void DelegueActesMedicaux(Animal a);
```

La méthode VisiteVeterinaire prend en paramètre un délégué qui représente tous les actes médicaux effectués durant la visite. Dans le corps de la méthode, l'exécution du délégué appelle ainsi automatiquement les méthodes correspondant aux différents actes.

Le type de délégué doit être déclaré. C'est ce qui est fait dans la dernière ligne, en dehors de la classe Animal. La déclaration définit le type de retour et les paramètres que devront avoir les méthodes correspondant aux actes médicaux (ici, retour de type void, et un paramètre de type Animal).

13.1.2 Branchement de méthodes à un délégué

Regardons maintenant comment appeler la méthode VisiteVeterinaire créée précédemment :

```
class Program
{
    static void Main(string[] args)
    {
        Animal capi = new Animal();

        // Création d'un délégué qui représente les différents actes médicaux
        DelegueActesMedicaux deleg = null;
        deleg += Deparasiter;
        deleg += Vacciner;

        // Déclenchement de la viste, puis affichage du carnet de santé
        capi.VisiteVeterinaire(deleg);
        foreach (var visite in capi.CarnetSanté)
            Console.WriteLine(visite);
    }

    private static void Deparasiter(Animal a)
    {
        a.CarnetSanté.Add("Déparasitage");
    }

    private static void Vacciner(Animal a)
    {
        a.CarnetSanté.Add("Vaccination");
    }
```

Dans la méthode Main, on déclare une variable de type délégué DelegueActesMedicaux0, et on branche dessus les méthodes souhaitées au moyen de l'opérateur +=. Le délégué représente ici les méthodes Deparasiter et Vacciner qui modélisent les actes médicaux. Les types de leurs paramètres et de leurs retours correspondent à ceux du type délégué.

On appelle ensuite la méthode VisiteVeterinaire sur une instance d'animal en lui passant le délégué en paramètre. L'appel de ce délégué dans le corps de la méthode appellera automatiquement les méthodes Deparasiter et Vacciner, dans cet ordre.

La méthode VisiteVeterinaire permet donc d'effectuer une visite sur un animal, sans savoir quels seront les actes médicaux pratiqués. Et surtout, l'implémentation de ces actes est fournie par les méthodes d'une classe complètement étrangère à la classe Animal (ici la classe Program).

Remarques :

- Il est possible de débrancher une méthode du délégué au moyen de l'opérateur -=.
- Lorsque le délégué ne représente qu'une seule méthode, il n'est pas nécessaire d'utilisé une variable de type délégué. On peut passer directement cette méthode en paramètre de l'autre, comme ceci :

```
capi.VisiteVeterinaire(Deparasiter);
```

Autre syntaxe

La syntaxe précédente est valable depuis C# 2.0. En C# 1.0 on utilisait une autre syntaxe, dans laquelle il fallait explicitement instancier le délégué :

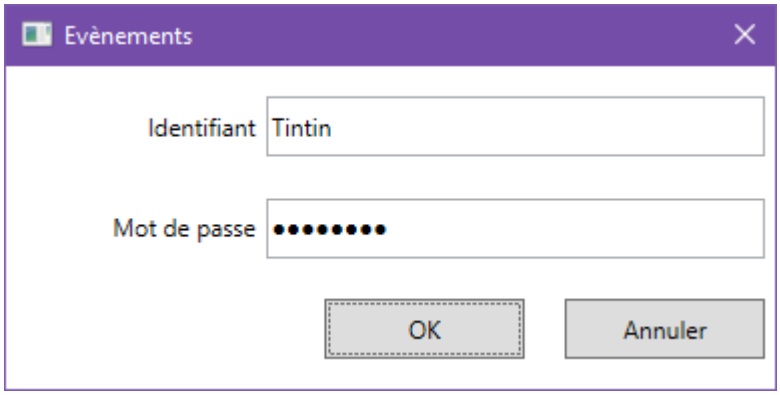
```
DelegueComparer comparateur = null;
comparateur += new DelegueComparer(ComparerNoms);
```

En effet, dans la syntaxe du C# 2.0, l'instanciation du délégué est implicite lorsqu'on branche une méthode avec l'opérateur +=.

La syntaxe du C# 1.00 est utilisée dans le code généré automatiquement par Visual Studio lors de la création d'interfaces graphiques, pour associer une méthode à un évènement émis par un contrôle graphique. C'est pourquoi il est bon de connaître, même si on ne l'utilise pas soi-même.

13.2Evénements

Une application classique possède une interface visuelle contenant des composants d'interaction avec l'utilisateur tels que des boutons, zones de saisies, barre de défilement…etc. Généralement, l'utilisateur est libre d'interagir avec ces éléments dans l'ordre qu'il veut, et il existe de nombreux scénarios d'utilisation possibles. Prenons l'exemple de la fenêtre ci-dessous :



Dans cette fenêtre, l'utilisateur peut saisir du texte dans des zones de texte et cliquer sur des boutons. Les clics sur ces boutons ne déclenchent évidemment pas le même traitement. Lors du clic sur le bouton OK, l'application doit vérifier si les informations d'authentification sont correctes et par exemple donner accès à l'interface principale de l'application. Un clic sur le bouton Annuler peut par exemple fermer l'application.

Ce type de comportement asynchrone met en œuvre des **évènements**, qui sont chargés de déclencher des traitements. Les clics sur les boutons sont des évènements. Chaque pression d'une touche à l'intérieur d'une zone de saisie est également un évènement. La prise de focus par une zone de saisie, la fermeture de la fenêtre modale…etc. sont aussi des évènements.

D'un point de vue du code, les évènements sont des points d'entrée pour l'interaction avec l'application.

L'utilisation des événements n'est cependant pas limitée aux interfaces graphiques. D'une manière générale, ils fournissent un moyen pour des objets au sens large (visuels ou non) de signaler (on dit aussi **notifier**) quelque-chose à leur environnement extérieur. D'autres objets de l'environnement extérieur peuvent **s'abonner** à l'évènement pour y réagir en déclenchant divers traitements.

Dans l'exemple ci-dessus, lorsqu'on clique sur le bouton OK, celui-ci émet un évènement « click » pour notifier ce clic à la fenêtre qui le contient. Dans le code lié à cette fenêtre, on peut s'abonner à cet évènement pour pouvoir exécuter une action (par exemple vérifier l'identité de l'utilisateur) à chaque fois que l'évènement est déclenché.

L'objet qui émet l'évènement n'a aucune connaissance des traitements qui seront exécutés en réaction à cet évènement. On peut ainsi créer des objets émetteurs d'évènements qui sont indépendants de leur environnement et réutilisables partout. Le code devient ainsi modulaire.

13.2.1 Abonnement à un évènement

Un évènement est en fait un type délégué particulier (déclaré avec le mot clé **event**)

L'abonnement à un évènement consiste donc à brancher une ou plusieurs méthodes dessus, comme nous l'avons fait plus haut pour les délégués ordinaires.

Les méthodes branchées sur des évènements sont appelées « **gestionnaires d'évènements** »

Le code suivant montre l'abonnement aux évènements Click des boutons de la fenêtre précédente :

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent(); // Code généré automatiquement par VS

        // Branchement de gestionnaires aux évènements Click
        // des boutons OK et Annuler
        btnOK.Click += BtnOK_Click;
        btnAnnuler.Click += BtnAnnuler_Click;
    }

    private void BtnOK_Click(object sender, RoutedEventArgs e)
    {
        // Vérification des informations saisies
        // ...
    }

    private void BtnAnnuler_Click(object sender, RoutedEventArgs e)
    {
        this.Close(); // Fermeture de la fenêtre
    }
}
```

Ce code est associé à la fenêtre représentée par la classe MainWindow. Dans le constructeur, après l'appel au code d'initialisation du visuel généré automatiquement par Visual Studio, on branche deux gestionnaires pour répondre aux évènements Click émis par les boutons OK et Annuler.

btnOK.Click et btnAnnuler.Click sont des délégués de type RoutedEventHandler, dont voici la déclaration :

```
public delegate void RoutedEventHandler(object sender, RoutedEventArgs e);
```

Cette déclaration est présente dans l'assembly PresentationCore.dll qui est référencée par le projet.

13.2.2 Création d'un évènement

Prenons l'exemple de l'application Microsoft Word que vous avez devant les yeux. Lorsqu'on clique sur un des titres du volet de navigation (affiché via une case à cocher du ruban Affichage), la vue du document défile jusqu'au titre cliqué. Juste pour l'exemple, on peut imaginer modéliser ce fonctionnement à l'aide de 3 classes :

- Document (objet de haut niveau contenant une liste de titres)
- Title (titre)
- DocumentViewer (vue du document)

Le code C# correspondant ressemblerait à ceci :

```
using System;
using System.Collections.Generic;

namespace Exemples
{
    public class Document
    {
        private var _titles = new Dictionary<string, Title>();

        // Evènement clic sur un titre
        public event EventHandler<StringEventArgs> TitleClickEvent;

        public void AddTitle(string id, string label)
        {
            _titles.Add(id, new Title(id, label));
        }

        // Méthode pour déclencher l'évènement
        public void SelectTitle(string titleId)
        {
            if (this.TitleClickEvent != null)
                this.TitleClickEvent(this, new StringEventArgs(titleId));
        }
    }

    // Classe décrivant l'argument pour l'événement TitleClickEvent
    public class StringEventArgs : EventArgs
    {
        public string StringValue { get; set; }
```

```
        public StringEventArgs(string s)
        {
            this.StringValue = s;
        }
    }

    // Modélise un titre de paragraphe
    public class Title
    {
        public string Id { get; }
        public string Label { get; }

        public Title(string id, string label)
        {
            this.Id = id;
            this.Label = label;
        }
    }

    // Modélise un visualiseur de document
    public class DocumentViewer
    {
        public Document CurrentDoc { get; }

        // Le document à visualiser est passé en paramètre du constructeur
        public DocumentViewer(Document doc)
        {
            CurrentDoc = doc;
            // Abonnement à l'évènement TitleClickEvent du document
            CurrentDoc.TitleClickEvent += OnDocTitleClick;
        }

        // Gestionnaire de l'évènement TitleClickEvent du document
        private void OnDocTitleClick(object sender, StringEventArgs e)
        {
            ScrollToTitle(e.StringValue);
        }

        // Méthode pour scroller jusqu'à un titre
        private void ScrollToTitle(string titleId)
        {
            Console.WriteLine("Défilement vers le titre {0}", titleId);
        }
    }
}
```

La classe Document possède un évènement `TitleClickEvent` associé à un type délégué System.EventHandler<T>, qui est fourni par le .net framework et fréquemment utilisé. Ce type délégué permet de définir la signature des méthodes capables de répondre à l'évènement.

DocumentViewer s'abonne à l'évènement Click de la classe Document en branchant sa méthode OnDocTitleClick à l'évènement (avec l'opérateur +=). La signature de cette méthode doit être identique à celle du type délégué associé à l'évènement.

Lorsqu'on clique sur un titre dans le volet de navigation, la classe Document déclenche son évènement `TitleClickEvent`, ce qui exécute la méthode OnDocTitleClick de l'objet DocumentViewer.

Le type délégué associé à l'évènement est de type EventHandler. Lorsqu'on regarde l'aide MSDN sur ce type, on voit qu'il comporte 2 paramètres : object sender et TEventArgs e.

- Le premier paramètre représente l'objet qui a émis l'évènement (ici, le document)
- Le second est de type générique, et représente les données transmises par cet objet. Dans notre exemple, nous avons créé un type StringEventArgs qui permet de stocker un identifiant de titre.

Exemple de code de mise en œuvre de ces classes :

```
class Program
{
    static void Main(string[] args)
    {
        // Crée un document et un visualiseur
        var doc = new Document();
        var view = new DocumentViewer(doc);

        // Ajoute des titres au document
        doc.AddTitle("T1", "Le premier titre");
        doc.AddTitle("T2", "Le second titre");
        doc.AddTitle("T3", "Le troisième titre");

        // Simule le clic sur un titre dans le volet de navigation
        doc.SelectTitle("T2");

        Console.ReadKey(true);
    }
}
```

On créer un document et un visualiseur, puis on ajoute des titres à la collection de titres du document.

On simule ensuite un clic sur le titre T2 en appelant la méthode SelectTitle du document (dans une application graphique, le déclenchement se ferait par un clic sur l'élément d'interface visuelle qui représente un titre). La méthode SelectTitle déclenche en interne l'évènement TitleClickEvent. Ceci produit l'affiche suivant à l'écran :

```
Défilement vers le titre T2
```

Dans une application graphique, cela ferait effectivement défiler le document.

Il est important de noter que Document ne possède aucune référence de DocumentViewer. C'est au contraire DocumentViewer qui connaît Document. Et pourtant, par l'intermédiaire de l'évènement, on arrive à déclencher automatiquement l'exécution d'une méthode de DocumentViewer à partir de Document. C'est bien là tout l'intérêt des évènements et délégués.

13.3 Méthodes anonymes et expressions lambda

13.3.1 Méthodes anonymes

Dans l'exemple du paragraphe 13.2.1, pour nous abonner à l'évènement Click du bouton Fermer, nous avons utilisé la syntaxe suivante :

```
btnFermer.Click += BtnFermer_Click;
```

```
private void BtnFermer_Click(object sender, RoutedEventArgs e)
{
    this.Close(); // Fermeture de la fenêtre
}
```

Il existe une syntaxe plus condensée utilisant une méthode anonyme :

```
btnFermer.Click += delegate(object sender, RoutedEventArgs e) { this.Close(); };
```

La méthode anonyme est constituée du code sur fond jaune.

Une méthode anonyme est simplement une méthode sans nom. Comme toute méthode, elle possède des paramètres et un corps, et peut retourner une valeur.

Comme cette méthode est branchée sur un évènement (qui est un type particulier de délégué), on la fait précéder du mot clé delegate, et sa signature doit respecter celle du délégué. Les méthodes anonymes sont quasiment toujours utilisées comme délégués.

Intérêt des méthodes anonymes

Les méthodes anonymes évitent la création de tas de petites méthodes spécifiques, et rendent le code plus concis. Un autre avantage est qu'elles servent d'adaptateur ; c'est-à-dire qu'elles peuvent appeler dans leur corps des méthodes qui ne seraient pas directement utilisables en tant que délégués du fait que leurs signatures ne respectent pas celle du délégué.

13.3.2 Expressions lambda

C# 3.0 a introduit un nouveau concept qui remplace les méthodes anonymes : les expressions lambda.

La syntaxe est la suivante :

```
// Abonnement à l'évènement TitleClickEvent du document
CurrentDoc.TitleClickEvent += (object sender, StringEventArgs e) => {
    ScrollToTitle(e.StringValue); };
```

Une expression lambda est une fonction anonyme qui peut être utilisée pour créer des types délégués ou des types d'arbres d'expressions (cf. chapitre sur LINQ).

On n'a donc plus besoin du mot clé delegate, mais on utilise par contre l'opérateur « => » appelé **opérateur lambda**, entre les paramètres et le corps de la fonction.

Pour l'abonnement à des évènements, on peut utiliser indifféremment les méthodes anonymes et les expressions lambda. Il est bon de connaître les 2 syntaxes pour ne pas être perturbé à la lecture du code.

Les expressions lambda ont cependant un domaine d'application plus large (notamment utilisées dans les requêtes LINQ). Il est donc préférable de les utiliser lorsqu'on écrit du nouveau code.

Différentes formes d'expressions lambda

Les expressions lambda peuvent prendre plusieurs formes légèrement différentes :

```
(ref int x, int y) => { x++; return x / y; }
```

Forme la plus générale. Les paramètres ont des types explicites. Certains peuvent être transmis par référence.

```
(x, y) => { x++; return x / y; }
```

Les types des paramètres sont déduits automatiquement par le compilateur (ont dit qu'ils sont inférés).

```
x => x * x
```

Forme la plus simple, sans type de paramètre, ni bloc de code

```
() => MaMethode()
```

Autre forme simplifiée, dans le cas où le corps appelle une méthode sans paramètre

Certaines syntaxes peuvent paraître déroutantes au départ, mais l'opérateur lambda (=>) permet de reconnaître qu'il s'agit d'expressions lambda, et donc de comprendre le code.

14 Requêtes LINQ

14.1 Présentation

LINQ : Language Integrated Query
LINQ permet de rechercher facilement dans une collection les éléments qui correspondent à un ensemble de critères. Par exemple, dans une collection d'objets Client, trouver tous les clients qui habitent à Londres, ou encore déterminer la ville qui a le plus de clients.
Sa syntaxe rappelle celle du SQL.

LINQ nécessite que les données soient stockées dans un conteneur qui implémente l'interface IEnumerable. C'est le cas des tableaux et de toutes les collections, génériques ou non, que nous avons étudiées précédemment.

1er exemple :

```
// Extraction des nombres pairs d'un tableau
int[] tab = new int[] { 1, 2, 3, 4, 5, 6 };
IEnumerable<int> pairs = from number in tab
                         where number % 2 == 0
                        select number;

foreach(int i in pairs) Console.WriteLine(i);
```

Cette requête simple récupère tous les nombres pairs d'un tableau d'entiers. Elle introduit déjà les concepts essentiels :

- La syntaxe ressemble à celle d'une requête SQL avec les mots clés from, where et select
- Contrairement à SQL, la clause select est toujours placée à la fin. Cet ordre est en fait plus naturel et permet à Visual Studio de proposer l'intellisense dans l'ensemble de la requête.
- Le résultat d'une requête est de type IEnumerable<T>, (ici T = int). Pour simplifier on peut tout à fait utiliser une variable typée implicitement (var).
- Number est un paramètre. En fait c'est le paramètre d'une expression lambda, comme nous le verrons plus bas.
- Dans la clause where, attention à bien utiliser l'opérateur « == » et non pas « = », car on fait bien un test et non une affectation

2d exemple

Prenons cette fois une requête retournant des types références.
Pour cela, créons un jeu de données à partir des classes créées dans les chapitres précédents. On crée ici une liste d'animaux domestiques (chiens et chats), en spécifiant leur sexe, poids, nom et date de naissance.

```
var anx = new List<AnimalDomestique>();

anx.Add(new Chat(Sexes.Femelle, 1.35, "Minette", new DateTime(2015, 11, 01)));
anx.Add(new Chat(Sexes.Male, 2.12, "Ponpon", new DateTime(2014, 02, 28)));
anx.Add(new Chat(Sexes.Femelle, 2.12, "Zica", new DateTime(2014, 09, 13)));
anx.Add(new Chien(Sexes.Male, 10.8, "Clovis", new DateTime(2011, 04, 14)));
anx.Add(new Chien(Sexes.Femelle, 8.4, "Rita", new DateTime(2012, 10, 17)));
anx.Add(new Chat(Sexes.Male, 3.4, "Grisou", new DateTime(2015, 01, 05)));
anx.Add(new Chat(Sexes.Femelle, 1.7, "Nina", new DateTime(2013, 07, 31)));
anx.Add(new Chien(Sexes.Femelle, 9.5, "Tina", new DateTime(2010, 05, 21)));
anx.Add(new Chien(Sexes.Male, 2.1, "Minus", new DateTime(2015, 10, 07)));
anx.Add(new Chien(Sexes.Male, 42.0, "Brutus", new DateTime(2011, 03, 11)));
```

Pour récupérer la liste des chats mâles, on écrira la requête suivante :

```
// Liste des chats mâles
var chats = from a in anx
            where a is Chat && a.Sexe == Sexes.Male
            select a;

// Affichage de leurs noms
foreach (var c in chats) Console.WriteLine(c.Nom);
```

Le résultat est de type IEnumerable<AnimalDomestique>. On le récupère dans une variable typée implicitement.
Dans la clause where, il est intéressant de noter qu'on peut utiliser le type sous-jacent de AnimalDomestique (qui est une classe abstraite) comme critère de sélection au moyen de « is »

14.2 Les deux syntaxes LINQ

Nous avons vu dans les exemples précédents, que les expressions de requêtes LINQ sont semblables à des requêtes SQL. Il existe cependant une autre manière de formuler les requêtes LINQ, en utilisant des méthodes et des expressions lambda. On l'appelle quelques fois syntaxe pointée, du fait que les appels de méthodes peuvent être enchaînés en les séparant par des points.

Reprenons les exemples précédents et écrivons les requêtes sous forme d'appels de méthodes :

```
// Extraction des nombres pairs d'un tableau
int[] tab = new int[] { 1, 2, 3, 4, 5, 6 };
IEnumerable<int> pairs = tab.Where(n => n % 2 == 0);
```

On voit ici qu'on utilise juste la méthode Where() ; il n'y a pas besoin de from ni de select.
Le filtre est exprimé au moyen d'une expression lambda.

```
// Liste des chats mâles
```

```
var chats2 = anx.Where(c => (c is Chat) && (c.Sexe == Sexes.Male));
```

Différences entres les 2 syntaxes
La syntaxe sous forme d'expressions de requêtes est sans doute plus intuitive, tandis que la syntaxe sous forme d'appels de méthodes est plus condensée.
Cependant, la différence majeure réside surtout dans les possibilités offertes par l'une et l'autre : la syntaxe sous forme d'expressions de requêtes ne permet pas d'utiliser tous les opérateurs de LINQ.
Cette page MSDN fournit la liste des méthodes de LINQ, et indique pour chacune la syntaxe d'expression de requête correspondante, si elle existe. On constate que pour la majorité des méthodes, il n'y a pas de correspondance.

14.3Opérateurs courants de LINQ

Les opérateurs de requête standard sont des fonctions de filtrage, de projection, d'agrégation, de tri, etc. Nous allons voir ici les plus communs :

Filtrage
La liste des chats mâles peut en fait s'obtenir également de cette façon :

```
var chats = anx.OfType<Chat>().Where(c => c.Sexe == Sexes.Male);
```

La méthode OfType<>() permet de filtrer selon le type.

La méthode Where() permet de filtrer selon n'importe quelle propriété de ce type

Calculs d'agrégation : min, max, somme, moyenne, dénombrement
Ces opérations ne sont réalisables qu'avec la syntaxe pointée. Voici quelques exemples :

```
// Poids moyen des chiens
var poidsMoyen = anx.OfType<Chien>().Average(a => a.Poids);
Console.WriteLine("Poids moyen des chiens : {0}", poidsMoyen);

// Date de naissance du chien mâle le plus vieux
var dateNais = anx.OfType<Chien>().Where(c => c.Sexe == Sexes.Male).Min(a =>
a.DateNaissance);
Console.WriteLine("Date de naissance du chien mâle le plus vieux : {0}", dateNais);

// Nombre total d'animaux
Console.WriteLine("Nombre d'animaux : {0}", anx.Count());

// Nombre d'animaux regroupés par année de naissance
foreach (var gp in anx.OrderBy(a => a.DateNaissance).GroupBy(a => a.DateNaissance.Year))
{
    Console.WriteLine("{0} animal(aux) né(s) en {1}", gp.Count(), gp.Key);
}
```

La méthode GroupBy renvoie ici une collection de groupes d'animaux. Le premier groupe contient les animaux nés en 2010, le second contient les animaux nés en 2011…etc. Sur chaque groupe, on peut obtenir la valeur de regroupement (ici l'année) avec la propriété Key, et faire une agrégation (ici le dénombrement avec la méthode Count()).

Tri

```
// Chat le plus vieux
var ani = anx.OfType<Chat>().OrderBy(a => a.DateNaissance).FirstOrDefault();
Console.WriteLine("Le chat le plus vieux est {0}. Il est né le {1})", ani.Nom,
ani.DateNaissance);
```

Dans cet exemple, on trie la liste des chats selon leur date de naissance avec la méthode Orderby(), et on prend le premier chat de la liste avec la méthode First().

Mise en garde
LINQ offre un moyen commode pour filtrer, trier et faire des calculs sur les données. Il ne doit cependant pas se substituer aux requêtes SQL sur la base de données !
Dans une architecture n-tiers où les données sont stockées dans une base de données, il est généralement beaucoup plus performant de faire le maximum d'opérations en SQL, via des requêtes ensemblistes, plutôt que de récupérer des données brutes, puis les traiter par le code C#.
Bien que les requêtes LINQ soient généralement très concises, elles peuvent engendrer des traitements très peu performants, du fait des multiples parcours de collections nécessaires. Il convient donc toujours de se demander si l'utilisation de LINQ est pertinente dans son contexte, et si oui, de vérifier les performances des requêtes.

15 Divers

Ce chapitre traite d'aspect particuliers du langage C#, qui sont utilisés moins fréquemment que les concepts présentés jusqu'ici. Ils sont présentés de façon succincte, avec pour objectifs de donner des exemples simples illustrant leur syntaxe de mise en œuvre.

15.1Méthodes d'extensions

Les **méthodes d'extensions** permettent d'étendre un type existant (structure ou classe), en lui ajoutant de nouvelles méthodes.

Intérêt : Toutes les variables déjà existantes de ce type profitent des nouvelles méthodes ajoutées.

Exemple : je souhaite avoir une méthode pour inverser les lettres d'une chaîne de caractères. Je pourrais pour cela, dériver la classe string et lui ajouter une méthode Inverser. L'inconvénient est que les variables de type string déjà existantes dans mon code ne pourront pas profiter de cette méthode. Il est donc préférable de créer une méthode d'extension, comme ceci :

```
public static class StringHelper
{
    public static int CompterMots(this string s)
    {
        string[] tabMots = s.Trim().Split(' ', '\'', '\t', '\n');
        return tabMots.Length;
    }
}
```

Utilisation de cette méthode :

```
class Program
{
```

```
    static void Main(string[] args)
    {
        string phrase = "J'aime le C#";
        int nbMots = phrase.CompterMots();

        Console.WriteLine("Il y a {0} mots dans la phrase", nbMots); // renvoie 4
        Console.ReadKey();
    }
}
```

Pour créer une méthode d'extension :

- La classe qui contient la méthode d'extension doit être statique
- La méthode d'extension elle-même doit être statique
- La méthode d'extension doit contenir un paramètre du type qu'elle étend, précédé du mot clé this (dans l'exemple ci-dessus : this string s)

Remarques :

- Généralement, on rassemble les méthodes d'extension d'un type dans une classe ayant le nom de ce type suffixé par « Helper » (ex : StringHelper, DateTimeHelper…)
- La méthode d'extension n'est pas réellement ajoutée à la classe étendue. Le code MSIL montre que le compilateur génère simplement un appel à une méthode statique : `Console.WriteLine(StringHelper.Inverser(s));`

15.2Surcharge des opérateurs

La surcharge d'un opérateur permet d'appliquer l'opération correspondante sur des opérandes de type que l'on définit soi-même (classe ou structure).

Opérateurs susceptibles d'être redéfinis :

- Opérateurs unaires (1 opérande) : **`~ ++ -- !`**
- Opérateurs binaires (2 opérandes) : **`+ - * / % & | ^ << >>`**

Un opérateur est défini avec le mot clé **operator** en tant que méthode statique sur la classe des opérandes. Il prend un ou deux paramètres en entrée, et renvoie une instance de la classe d'opérande.

Il est également possible de surcharger les opérateurs de conversion explicite.

L'exemple de code ci-dessous illustre la syntaxe à mettre en œuvre pour ces 2 cas :

```
namespace Operateurs
{
    class Point
    {
        public double X { get; set; }
        public double Y { get; set; }
        public double Z { get; set; }

        public Point(double x, double y, double z)
        {
            X = x;
```

```
            Y = y;
            Z = z;
        }

        // Surcharge de l'opérateur +
        public static Point operator +(Point c1, Point c2)
        {
            return new Point(c1.X + c2.X, c1.Y+c2.Y, c1.Z+c2.Z);
        }

        // Opérateur de conversion explicite en double
        // (retourne la distance du point par rapport à l'origine)
        public static explicit operator double(Point c)
        {
            return Math.Sqrt(c.X * c.X + c.Y * c.Y + c.Z * c.Z);
        }

        public override string ToString()
        {
            return String.Format("({0},{1},{2})", X, Y, Z);
            // renvoie une chaîne au format : (x,y,z)
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var P1 = new Point(2, 4, 6);
            var P2 = new Point(3, 5, 7);
            var P3 = P1 + P2;

            Console.WriteLine("{0} + {1} = {2}", P1, P2, P3);
            Console.WriteLine("(double){0} = {1}", P1, (double)P1);
            Console.ReadKey();
        }
    }
}
```

Sortie console :

```
(2,4,6) + (3,5,7) = (5,9,13)
(double)(2,4,6) = 7,48331477354788
```

15.3 Métadonnées et réflexion

Au moment de la compilation du code, le compilateur produit des métadonnées de description pour tous les types (valeur ou référence) utilisés dans l'application.

Ces métadonnées sont décrites par la classe **`System.Type`**, qui comporte de nombreux membres, tels que le nom du type, la liste de tous ses membres…etc.

La réflexion désigne le fait d'accéder à ces informations durant l'exécution. La réflexion est utilisée par exemple pour lire les attributs qui décorent les classes ou propriétés (cf. paragraphe suivant). Visual Studio

l'utilise également pour afficher le contenu de l'explorateur de classes, et pour l'auto-complétion (IntelliSense).

Pour accéder aux métadonnées, on utilise la méthode GetType définie sur la classe object, et donc accessible sur tout objet. Cette méthode renvoie un objet Type.

Ex : considérons la classe suivante :

```
public class Cerf : Animal
{
    public Cerf() { }
    public Cerf(Sexes sexe, double poids) : base(sexe, poids) { }

    // Propriétés issues de l'nterface IClassable
    public override string Espece { get { return "Cerf"; } }
    public override string Famille { get { return "Cervidé"; } }
    public override string Ordre { get { return "Artiodactyle"; } }
    public override string Classe { get { return "Mammifère"; } }
    public override string Embranchement { get { return "Vertébré"; } }
}
```

Le code ci-dessous permet d'afficher le type et le nom de tous les membres de la classe :

```
class Program
{
   static void Main(string[] args)
   {
      Animal cerf = new Cerf();
      Type t = cerf.GetType();
      Console.WriteLine("Membres de " + t.Name);
      foreach (var m in t.GetMembers())
      {
          Console.WriteLine("Type: {0}, Nom: {1}", m.MemberType, m.Name);
      }

      Console.ReadKey();
   }
}
```

Sortie console :

```
Membres de Cerf
Type: Method, Nom: get_Espece
Type: Method, Nom: get_Famille
Type: Method, Nom: get_Ordre
Type: Method, Nom: get_Classe
Type: Method, Nom: get_Embranchement
Type: Method, Nom: get_Poids
Type: Method, Nom: set_Poids
Type: Method, Nom: get_Sexe
Type: Method, Nom: ToString
Type: Method, Nom: Equals
Type: Method, Nom: GetHashCode
Type: Method, Nom: GetType
Type: Constructor, Nom: .ctor
Type: Constructor, Nom: .ctor
```

```
Type: Property, Nom: Espece
Type: Property, Nom: Famille
Type: Property, Nom: Ordre
Type: Property, Nom: Classe
Type: Property, Nom: Embranchement
Type: Property, Nom: Poids
Type: Property, Nom: Sexe
```

On notera que :

- Les méthodes commençant par « get_ » et « set_ » sont les accesseurs des propriétés
- Comme Cerf hérite d'Animal, qui hérite elle-même de Object, elle possède les membres de ses ancêtres : les propriétés Poids et Sexe, et les méthodes ToString, Equals, GetHashCode et GetType.

15.4Attributs

Les attributs permettent d'associer des métadonnées à des éléments de code (types, méthodes, propriétés, etc.). Il peut s'agir d'instructions pour le compilateur, de descriptions, ou d'indications sur la façon de traiter l'élément de code.

Une fois qu'il est associé à un élément de code, l'attribut peut être interrogé au moment de l'exécution par réflexion.

On spécifie un attribut en mettant son nom entre crochets devant la déclaration de l'élément auquel il s'applique :

```
[Conditional("DEBUG")]
void TraceMethod()
{
    // ...
}
```

Dans cet exemple, la méthode TraceMethod est exécutée uniquement si on est en mode debug.

Certains attributs s'appliquent sur l'assembly lui-même. On les trouve dans le fichier AssemblyInfo.cs créé automatiquement dans chaque projet.

La liste de toutes les classes d'attributs prédéfinis du .net framework peut être consultée sur cette page MSDN. Il est possible également de créer sa propre classe d'attribut en dérivant System.Attribute.

La création d'un fichier xml à partir d'une arborescence d'objets (sérialisation), et le chargement d'un fichier xml dans une arborescence d'objets (désérialisation), sont des opérations qui peuvent être grandement facilitées par l'utilisation d'attributs. Ils permettent dans ce cas d'indiquer à la classe XmlSerializer la nature des liens entre les éléments et attributs xml, et les propriétés des classes qui stockent de l'arborescence.

Exemple : nous allons charger le fichier xml ci-dessous dans une liste :

```
<?xml version="1.0" encoding="utf-8"?>
<Voitures>
   <Voiture Id="1">
      <Marque>Ferrari</Marque>
      <Modele>458 Speciale A</Modele>
      <Annee>2014</Annee>
```

```
   </Voiture>
   <Voiture Id="2">
      <Marque>Ferrari</Marque>
      <Modele>California 30</Modele>
      <Annee>2012</Annee>
   </Voiture>
      <Voiture Id="3">
      <Marque>Lamborghini</Marque>
      <Modele>Aventador</Modele>
      <Annee>2015</Annee>
   </Voiture>
   <Voiture Id="4">
      <Marque>Lamborghini</Marque>
      <Modele>Huracan</Modele>
      <Annee>2014</Annee>
   </Voiture>
</Voitures>
```

Pour cela, nous pouvons utiliser le code suivant :

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Xml.Serialization;

namespace Attributs
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Voiture> voitures;
            XmlSerializer deserializer = new XmlSerializer(typeof(List<Voiture>),
                                new XmlRootAttribute("Voitures"));
            using (var fs = new FileStream("Voitures.xml", FileMode.Open))
            {
                voitures = (List<Voiture>)deserializer.Deserialize(fs);
            }

            foreach (var v in voitures)
                Console.WriteLine("{0} : {1} {2} de {3}",
                    v.Id, v.Marque, v.Modèle, v.Année);

            Console.ReadKey();
        }
    }

    public class Voiture
    {
        [XmlAttribute("Id")]
        public string Id { get; set; }

        [XmlElement("Marque")]
        public string Marque { get; set; }

        [XmlElement("Modele")]
        public string Modèle { get; set; }
```

```
        [XmlElement("Annee")]
        public int Année { get; set; }
    }
}
```

Sortie console :

```
1 : Ferrari 458 Speciale A de 2014
2 : Ferrari California 30 de 2012
3 : Lamborghini Aventador de 2015
4 : Lamborghini Huracan de 2014
```

Les attributs XmlElement et XmlAttribute placés au-dessus des propriétés de la classe voiture permettent de faire le lien avec les éléments et attributs xml.

Le chargement du fichier xml dans la liste voitures peut alors se faire en quelques lignes de codes.

On pourrait de même générer très facilement le fichier xml à partir du contenu de la cliste en appelant la méthode Serialize du XmlSerializer.

16 Références bibliographiques

Page d'accueil de la documentation C# de Microsoft

Livre « Visual C# 2010 Etape par étape » de John Sharp

Site Developpez.com