Cyril Seguenot

2016

Résumé

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

Formation à C# 3.0

Support de cours

Toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l’auteur ou de ses ayants droit ou ayants cause est illicite selon le Code de la propriété intellectuelle (Art L 122-4) et constitue une contrefaçon réprimée par le Code pénal.

Seules sont autorisées (Art L122-5) les copies ou reproductions strictement réservées à l’usage privé du copiste et non destinées à une utilisation collective, ainsi que les analyses et courtes citations justifiées par le caractère critique, pédagogique ou d’information de l’œuvre à laquelle elles sont incorporées, sous réserve, toutefois, du respect des dispositions des articles L 122-10 à L 122-12 du même Code, relatives à la reproduction par reprographie.

©Cyril Seguenot 2016

SOMMAIRE

[1 Introduction 5](#_Toc475053106)

[2 Présentation de C# et .net 5](#_Toc475053107)

[3 Création d’une application dans Visual Studio 7](#_Toc475053108)

[3.1 Solution, projet, assembly 7](#_Toc475053109)

[3.2 Références 8](#_Toc475053110)

[3.3 Compilation, exécution, débogage 9](#_Toc475053111)

[4 Les héritages du langage C 10](#_Toc475053112)

[4.1 Premier exemple console 10](#_Toc475053113)

[4.2 Types primitifs du C# (alias de types .NET) 11](#_Toc475053114)

[4.3 Littéraux / format des constantes 12](#_Toc475053115)

[4.4 Fonctions et paramètres ref et out 13](#_Toc475053116)

[4.5 Types structurés 15](#_Toc475053117)

[5 Erreurs et exceptions 16](#_Toc475053118)

[5.1 Lever une exception 16](#_Toc475053119)

[5.2 Intercepter une exception 17](#_Toc475053120)

[5.3 L’Instruction using 20](#_Toc475053121)

[6 La programmation orientée objet en C# 21](#_Toc475053122)

[6.1 Le paradigme objet en bref (vocabulaire) 21](#_Toc475053123)

[6.2 Types valeur et types référence 22](#_Toc475053124)

[6.3 Différences entre structures et classes 25](#_Toc475053125)

[6.4 Types nullables 25](#_Toc475053126)

[6.5 Typage implicite avec var 26](#_Toc475053127)

[6.6 Portée des variables et fonctions 26](#_Toc475053128)

[6.7 La classe string 27](#_Toc475053129)

[6.8 Les énumérations 31](#_Toc475053130)

[7 L'écriture de classes en C# 32](#_Toc475053131)

[7.1 Les espaces de noms (namespaces) 32](#_Toc475053132)

[7.2 Accessibilité des membres d’une classe 34](#_Toc475053133)

[7.3 Les propriétés 35](#_Toc475053134)

[7.4 Le principe d’encapsulation 38](#_Toc475053135)

[7.5 Signature et surcharge des méthodes 38](#_Toc475053136)

[7.6 Champs constants / en lecture seule 39](#_Toc475053137)

[7.7 Constructeurs 39](#_Toc475053138)

[7.8 Membres statiques 42](#_Toc475053139)

[8 Composition et agrégation 44](#_Toc475053140)

[9 L’héritage (la dérivation) 44](#_Toc475053141)

[9.1 Appels des constructeurs 45](#_Toc475053142)

[9.2 Méthodes virtuelles et redéfinies 46](#_Toc475053143)

[9.3 Propriétés virtuelles et redéfinies 48](#_Toc475053144)

[9.4 Polymorphisme d’héritage 49](#_Toc475053145)

[10 Classes abstraites et interfaces 50](#_Toc475053146)

[10.1 Les classes et méthodes abstraites 50](#_Toc475053147)

[10.2 Les interfaces 54](#_Toc475053148)

[11 Classe Object, boxing, conversion de type 58](#_Toc475053149)

[11.1 Boxing / unboxing 58](#_Toc475053150)

[11.2 Transtypage et opérateurs is et as 59](#_Toc475053151)

[11.3 Métadonnées et réflexion 60](#_Toc475053152)

[12 Les classes conteneurs 61](#_Toc475053153)

[12.1 Les tableaux 61](#_Toc475053154)

[12.2 Les collections non génériques 64](#_Toc475053155)

[12.3 Introduction aux génériques 65](#_Toc475053156)

[12.4 Interfaces génériques de collections 66](#_Toc475053157)

[12.5 Collections génériques 69](#_Toc475053158)

[12.6 Enumérations des collections 70](#_Toc475053159)

[13 Délégués et événements 71](#_Toc475053160)

[13.1 Délégués 71](#_Toc475053161)

[13.2 Evénements 75](#_Toc475053162)

[13.3 Méthodes anonymes et expressions lambda 79](#_Toc475053163)

[14 Requêtes LINQ 80](#_Toc475053164)

[14.1 Présentation 80](#_Toc475053165)

[14.2 Les 2 syntaxes LINQ 82](#_Toc475053166)

[14.3 Opérateurs courants de LINQ 82](#_Toc475053167)

[15 Divers 83](#_Toc475053168)

[15.1 Types imbriqués 83](#_Toc475053169)

[15.2 Méthodes d’extensions 84](#_Toc475053170)

[15.3 Surcharge des opérateurs 85](#_Toc475053171)

[15.4 Indexeurs 86](#_Toc475053172)

[15.5 Attributs 87](#_Toc475053173)

[16 Architecture en couches 90](#_Toc475053174)

[16.1 Caractéristiques 90](#_Toc475053175)

[16.2 Intérêt 91](#_Toc475053176)

[16.3 Mise en œuvre 92](#_Toc475053177)

[17 Références bibliographiques 93](#_Toc475053178)

# 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.

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.

# Présentation de C# et .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).

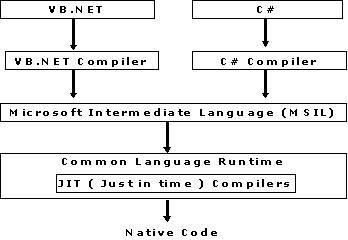
Le [**.NET Framework**](https://msdn.microsoft.com/fr-fr/library/zw4w595w(v=vs.110).aspx)est constitué d’un environnement d'exécution (CLR Common Language Runtime) et d’une bibliothèque de classes, interfaces et types valeur (plusieurs milliers répartis dans 53 espaces de noms principaux dans .net 4.6). 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.



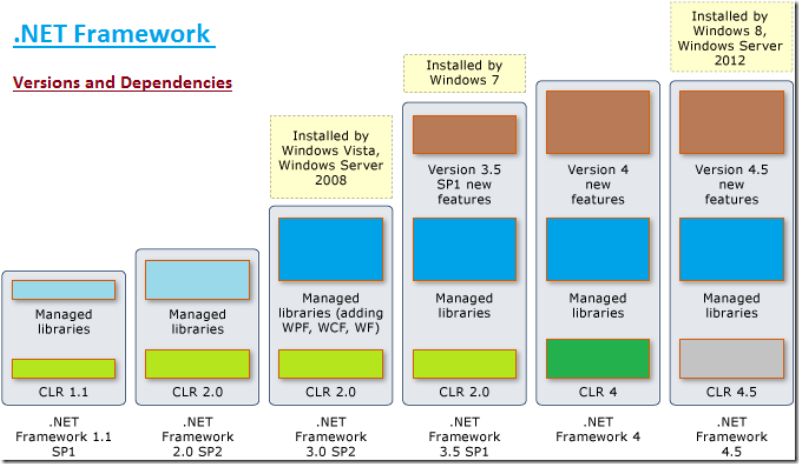
Le runtime est un environnement d'exécution qui fournit des services aux programmes qui s'exécutent sous son contrôle (on parle de code « managé ») :

* 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 :**



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 :

* Les performances du code managé sont généralement moins bonnes que celle du code natif
* Jusqu’ici, l’environnement de développement Visual Studio était certes très puissant, mais aussi très volumineux et gourmand en ressources. Microsoft a toutefois beaucoup travaillé sur ces points depuis la version 2017, dans laquelle l’installation est beaucoup plus personnalisable.
* La cohabitation d’applications utilisant des versions différentes du framework peut parfois poser problème.

# Création d’une application dans Visual Studio

## 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.

## 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 crées 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)

## Compilation, exécution, débogage

### 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.

#### 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.

# Les 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

## 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);

}

}

## 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 :

**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.

## 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.

S**équences d’échappement les plus courantes (liste complète** [**ici**](https://msdn.microsoft.com/fr-fr/library/aa691090(v=vs.71).aspx)**) :**

\n (newline) : nouvelle ligne

\t (horizontal tab) : tabulation horizontale

\b (backspace) : retour arrière

\r (carriage return) : retour charriot

\\ (backslash) : antislash

\” (double quote) : double côte

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 doubles côtes 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

## Fonctions et paramètres ref et out

**Définition des fonctions**

En C#, les fonctions sont nécessairement définies au sein de types structurés (struct ou class). Il n'y a pas réellement de fonction “globale” au sens du C. Néanmoins, les fonctions membres suivent la même logique de définition et d'appel que les fonctions du C. La principale différence est que le compilateur C# n'a pas besoin de fichier de prototypes (les fichiers .h header du C). La définition des fonctions suffit. Le compilateur retrouve seul la signature de la fonction à l'intérieur de données de description de type, ce qui élimine beaucoup de problèmes de compilation.

**Passage de paramètres**

Le passage des paramètres se fait par valeur (comme en C) :

class Program

{

static void Main(string[] args)

{

int n = 3;

Console.WriteLine("Multiplie2(n) = {0}, n = {1}", Multiplie2(n), n);

Console.ReadKey(true);

}

static int Multiplie2(int nombre) // nombre est initialisé à partir de n

{

nombre \*= 2; // sans effet sur n

return nombre;

}

}

Sortie console :

Multiplie2(n) = 6, n = 3

La fonction Multiplie2 n’a pas modifié la valeur initiale de n, car elle a utilisé comme paramètre 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.

**Passage par référence : mots clé ref et out**

Par défaut, les paramètres sont passés par valeur, comme en C. Mais le C# introduit 2 mots clés pour que des paramètres d'appel puissent être modifiables dans une fonction : ref et out

NB/ cela reprend l'idée des références du C++

**ref** indique que le paramètre formel est un alias (= une référence) du paramètre d'appel, plutôt qu’une copie. C’est-à-dire que c'est la même variable mais avec un autre nom localement. Pour utiliser un paramètre ref, il faut que :

* Le paramètre d’appel soit initialisé **avant** l’appel de la fonction.
* Le mot clé ref soit placé devant le paramètre, dans la signature de la fonction **et** lors de l’appel de la fonction

class Program

{

static void Main(string[] args)

{

int n = 3; // initialisation obligatoire avant le passage par ref

Multiplie2(ref n); // Appel de la fonction avec passage par ref

Console.WriteLine("n = {0}", n);

Console.ReadKey(true);

}

static void Multiplie2(ref int nombre) // nombre est un alias de n

{

nombre \*= 2; // modifie réellement n

// Pas besoin de valeur de retour pour la fonction

}

}

La sortie console est bien :

n = 6

Notons que comme le paramètre d’appel est réellement modifié par la fonction, celle-ci n’a pas besoin de renvoyer une valeur (retour void).

Dans certains cas, on souhaite pouvoir passer à la fonction un paramètre non initialisé, afin qu’elle lui affecte elle-même une valeur. Dans ce cas, on utilise le mot clé **out**.

class Program

{

static void Main(string[] args)

{

int n; // variable non initialisée

InitNombreAleatoire(out n);

Console.WriteLine("n = {0}", n);

Console.ReadKey(true);

}

static void InitNombreAleatoire(out int nombre)

{

// Affecter une valeur aléatoire au nombre

Random rd = new Random();

nombre = rd.Next();

}

}

Comme précédemment, il faut que le mot clé out soit placé devant le paramètre, dans la signature de la fonction, et lors de l’appel de la fonction.

## Types structurés

Le mot clé **struct** permet la définition d'un type similaires aux structures du C.

Ci-dessous, le type **Point** contient deux champs entiers, **X** et **Y**, auxquels on peut accéder à l'aide de la notation **p.X** et **p.Y** où **p** est une variable de type **Point**. C'est très proche des types structurés du langage C.

C# permet d’ajouter en plus des fonctions dans le type structuré.

using System;

namespace Exemple

{

struct Point

{

public int X;

public int Y;

public void SetXY(int pX, int pY)

{

X = pX;

Y = pY;

}

public string GetXY()

{

return "(" + X.ToString() + "," + Y.ToString() + ")";

}

}

class Program

{

static void Main(string[] args)

{

Point p1 = new Point();

p1.SetXY(2, 4);

Point p2 = new Point();

p2.SetXY(5, 7);

Console.WriteLine("p1 = {0}, p2 = {1}", p1.GetXY(), p2.GetXY());

Console.ReadKey(true);

}

}

}

Sortie console :

p1 = (2,4)

p2 = (5,7)

# Erreurs et exceptions

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’applications.

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…)

## 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](https://msdn.microsoft.com/query/dev14.query?appId=Dev14IDEF1&l=FR-FR&k=k(System.Exception);k(TargetFrameworkMoniker-.NETFramework,Version%3Dv4.5.2);k(DevLang-csharp)&rd=true). L’ancêtre de plus haut niveau est la classe Exception.

On peut utiliser directement ces classes, ou bien créer une nouvelle classe dérivée de 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).

## Intercepter une exception

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

Ex : 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 de 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.

Au final, 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.

**Clause 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 :

class Program

{

static void Main(string[] args)

{

try

{

EcritFichier();

}

catch (Exception e)

{

// permet de gérer notamment l'erreur System.UnauthorizedAccessException

Console.WriteLine(e.Message);

}

// Attend l’appui d’une touche afin d’empêcher la console de se fermer

Console.ReadKey(true);

}

static void EcritFichier()

{

StreamWriter outputFile = null;

try

{

outputFile = new StreamWriter(@"C:\Temp\essai.txt", true);

outputFile.WriteLine("Coucou !");

}

catch(DirectoryNotFoundException e)

{

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.

## 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 :

StreamWriter outputFile = null;

try

{

outputFile = new StreamWriter(@"C:\Temp\essai.txt", true);

outputFile.WriteLine("Coucou !");

}

finally

{

if (outputFile != null) outputFile.Close();

}

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 bloc 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.

# La programmation orientée objet en C#

## Le paradigme objet en bref (vocabulaire)

Un **objet** est une variable structurée qui regroupe :

* Des données, représentées par des **champs**
* Des traitements/interactions sur ces données, représentées par des **méthodes**

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

Une **classe** est un modèle (=type) d’objet

Une **instance** de classe est un exemplaire de cette classe, sa représentation en mémoire. On peut créer de multiples instances d’une classe.

Exemple :

* Renault Clio est un modèle de voiture
* La Renault Clio de Paul et celle de Marie sont 2 exemplaires de ce modèle

Le modèle décrit complètement toutes les caractéristiques permettant de créer des exemplaires.

Les mots « objet » et « instance » sont synonymes. Quand on crée un objet, on dit aussi qu’on **instancie** la classe.

Les structures et les classes sont des **types structurés**. Les classes permettent de faire bien plus de choses que les structures, et sont à la base de la POO.

La **programmation orientée objet** (POO) consiste à décrire un système sous forme de classes et d’interfaces, en utilisant 3 notions fondamentales :

* 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

## Types valeur et types référence

### Allocation mémoire, pile et tas

Les variables nécessitent qu’on leur alloue de la place en mémoire. Il existe 2 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 dits 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 développeur

Différences majeures entre la pile et le tas (cf. [cette page](http://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap) 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 2 gérés dans la RAM.

Les programmes Java et .net utilisent un **ramasse-miettes (Garbage Collector)**, qui permet d'utiliser 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.

### Types valeur et types référence

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 numériques primitifs du C# (byte, char, long …), les énumérations et les structures.

**Quels sont les types référence ?** Les classes (string est une classe).

Quand on instancie une classe (i.e. qu'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 libéré ?** Pour les variables de type valeur, à la fin du bloc de déclaration. Pour les objets de type référence, quand il n'y a plus aucune référence sur l’objet (il n'y a alors plus aucun moyen d'y accéder). Dans ce cas, le Garbage Collector (un programme qui s'exécute périodiquement pour libérer de la mémoire) récupère la mémoire de l'objet.

### Distinctions concernant l'affectation

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

Même pour les variables de type struct, l'affectation affecte la valeur, champ par champ. Dans l'exemple ci-dessous où **Point** est le type struct défini précédemment, l'affectation **p2=p1** réalise l'affectation attribut par attribut.

p2 = p1 équivaut à p2.X = p1.X et p2.Y = p1.Y (même si les champs X et Y étaient privés! )

static void Main(string[] args)

{

Point p1 = new Point(); // p1 et p2 sont dans la pile car ce sont des structures

Point p2 = new Point();

p1.SetXY(2, 3); // p1 a l'état (2,3)

p2 = p1; // l'état (2,3) est affecté à p2

Console.WriteLine("p2 = " + p2.ToString()); // p2 = (2,3)

}

**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 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, dans lequel PointC est une classe comportant les mêmes champs que la structure Point vue précédemment :

static void Main(string[] args)

{

PointC p3 = new PointC(); // une instance

PointC p4 = new PointC(); // une seconde

p3.SetXY(2, 3); // l'instance désignée par p3 vaut (2,3)

p4.SetXY(6, 1); // celle désignée par p4 vaut (6,1)

p4 = p3; // p4 désigne désormais le même objet que p3,

// c'est à dire l'instance d'état (2,3)

// L'instance dont l'état est (6,1) n'est plus accessible !

// le Garbage Collector peut libérer sa mémoire

}

Dans cet exemple, quand on affecte p3 à p4, puisque **PointC** est un type référence, les variables p3 et p4 sont des références. Par conséquent, après l'affectation p4 = p3, p4 fera désormais référence au même objet que p3. Il y aura alors deux références au même objet (celui d'état (2,3)). L'objet dont l'état est (6,1) n'aura donc plus de référence qui le désigne. Il n'est plus référencé, et le Garbage Collector pourra donc libérer sa mémoire au moment opportun.

### Distinctions concernant l'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.

## Différences entre structures et classes

Les structures (crées avec le mot-clé **struct)** sont des types valeur, alors que les classes sont des types référence.

Points communs entre les 2 :

* Notions de méthodes, propriétés et champs
* Peuvent implémenter des interfaces

Différences :

* 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.

Le principal intérêt des structures par rapport aux classes est leur accès mémoire plus rapide, et le fait qu’elles ne nécessitent pas le garbage collector.

Les structures doivent être utilisées principalement pour les objets transportant peu de données membres. Au-delà de quelques octets, le bénéfice obtenu du fait que l'allocation se fait dans la pile est perdu lors des passages de paramètres, puisqu'un objet de type valeur est dupliqué lorsque l'on fait un passage par valeur.

## Types 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.

Ex : si on reprend l’exemple de la classe PointC vue précédemment, on pourrait écrire :

static void Main(string[] args)

{

PointC p3 = null; // p3 a la valeur null par défaut

P3 = new PointC(); // on lui affecte une référence d’instance

...

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**

La possibilité d'assigner null à des types numériques et booléens est particulièrement utile lorsqu’on récupère le résultat d’une requête sur une base de données. En effet, dans une base de données les champs peuvent être définis comme nullables. Seuls des variables de types nullables sont en mesure de stocker leurs valeurs à coup sûr (sans plante), sans faire de vérification préalable.

## Typage 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 la 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.

## Portée des variables et fonctions

La portée d’un identificateur (variable ou méthode) est la région du programme dans laquelle cet identificateur 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 alors dans la portée de cette classe. Le nom exact en C# pour une variable définie dans une classe est **champ**. Contrairement aux variables locales, on peut utiliser les champs pour partager des informations entre les méthodes (les champs peuvent être utilisés par toutes les méthodes).

## La classe string

### Généralités

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.

### 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 des types numériques (Lettre + nombre) :**

Les lettres possibles sont : C = Currency, D = Decimal, G = général, F = fixed point…etc.

La liste exhaustive est fournie sur la page MSDN de la classe [NumberFormatInfo](https://msdn.microsoft.com/fr-fr/library/system.globalization.numberformatinfo(v=vs.100).aspx)

Le nombre représente le nombre de chiffres après la virgule.

**Chaînes de formats de dates et durées :**

En plus de formats prédéfinis tels que d (date courte), D (date longue), f (date et heure courtes) …etc., il est possible de définir des formats personnalisés.

L’ensemble des possibilités est décrit par la page MSDN de la classe [DateTimeFormatInfo](https://msdn.microsoft.com/fr-fr/library/system.globalization.datetimeformatinfo(v=vs.100).aspx)

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 3 formats différents à la même date. Les 2 premiers sont des formats prédéfinis, et le 3ème est un format personnalisé.

**Chaînes de formats d’énumérations :**

Il est possible d’afficher soit le texte, soit la valeur entière d’un énuméré. On utilise pour cela les formats G, F, D ou X. Cf. [cette page](https://docs.microsoft.com/fr-fr/dotnet/articles/standard/base-types/enumeration-format) docs.microsoft.com pour plus de détails sur chaque format.

### Méthode Format

Cette méthode permet de construire des chaînes de caractères formatées contenant des valeurs paramétrables. Elle accepte en paramètre un nombre variable d’objets :

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

{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 méthode Format accepte un nombre variable d’arguments grâce au fait qu’elle utilise un tableau d’objets params. C’est un concept que nous verrons dans le chapitre sur les tableaux.

La méthode Format appelle la méthode ToString sur chacun des paramètres qu’on lui passe. Pour spécifier la culture et les formats appliqués aux paramètres, on utilisera la syntaxe suivante :

var s4 = string.Format(CultureInfo.GetCultureInfo("en-US"),

" Un réel : {0:C1}\n Une chaîne : {1}\n Une date : {2:d}",

123.1548, "coucou", DateTime.Now);

Console.WriteLine(s4);

Sortie console :

Un réel : $123.2  
 Une chaîne : coucou  
 Une date : 2/11/2017

**Remarque**

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);

## Les énumérations

Un type créé avec le mot clé **enum** représente des variables pouvant prendre un nombre fini de valeurs prédéfinies. Une variable d'un type énuméré peut prendre pour valeur une des valeurs définies dans la liste décrivant l'énumération :

enum Feu {vert,orange,rouge}

...

Feu f; // f est une variable

f=Feu.vert; // Feu.vert est une valeur

Les valeurs d'une énumération correspondent à des valeurs entières qu'on peut choisir.

On peut transtyper une valeur énumérée en valeur entière. La conversion doit être explicite.

using System;

namespace SyntaxeCSharp

{

enum Color { red, blue, green };

enum Access { personal = 1, group = 2, all = 4 };

class Programme

{

static void Main()

{

Color c = Color.blue;

Console.WriteLine("Couleur {0} : {1}, (int)c, c);

Access a1 = Access.group;

Access a2 = Access.personal;

Console.WriteLine((int)a1);

Console.WriteLine(a2);

}

}

}

Sortie console :

Couleur 1 : blue

2

personal

Par défaut, les valeurs énumérées forment une suite 0, 1, 2…N Mais on peut spécifier soi-même chaque valeur.

**Enumérations comme bits indicateurs**

Ce type d’énumération est indiqué par la présence de l’attribut [Flags] juste au-dessus de sa déclaration. Une variable énumérée de ce type peut stocker n'importe quelle **combinaison** des valeurs définies dans la liste.

[Flags]

enum Jours

{

Aucun = 0, // 0

Lundi = 1, // 2^0

Mardi = 2, // 2^1

Mercredi = 4, // 2^2

Jeudi = 8, // 2^3

Vendredi = 16, // 2^4

Samedi = 32, // 2^5

Dimanche = 64 // 2^6

}

class MaClasse

{

Jours weekEnd = Jours.Samedi | Jours.Dimanche;

}

Les combinaisons de valeurs sont réalisées à l’aide de l’opérateur | (opérateur ou bit à bit).

/!\ Pour que cela fonctionne, il faut affecter explicitement les valeurs de la liste avec les puissances de 2. Sinon, les résultats des opérations avec les opérateurs binaires | et & seront faux.

Pour vérifier si une valeur énumérée fait partie d’une combinaison, on utilise l’opérateur &.

Exemple :

Jours weekEnd = Jours.Samedi | Jours.Dimanche

Jours j = Jours.Samedi;

// Vérifie si le jour j fait partie du week-end

if ((j & weeEnd) == j)

{

Console.WriteLine("{0} est un jour de week-end", s);

}

# L'écriture de classes en C#

## Les espaces de noms (namespaces)

**Un espace de noms** désigne une portée qui contient un ensemble d’objets connexes. Les espaces de noms permettent simplement d’organiser les éléments de code de façon cohérente, et de les isoler pour éviter des conflits de noms. On peut ainsi avoir un même nom de type dans différents espaces de noms.

Le nom complet d’un type est composé comme ceci : <espace de nom>.<nom du type>

Dans l’exemple ci-dessous, on a par exemple les types Espace1.A et Espace2.Divers.A

using System;

namespace Espace1 // namespace avec deux types A et B

{

class A { }

class B { }

}

namespace Espace2 // namespaces imbriqués

{

namespace ClassesImportantes

{

class A { }

}

}

namespace Espace2.Divers

// Autre façon d’imbriquer les namespaces

// equivaut à namespace Espace2 { namespace Divers{...} }

{

class A { }

}

namespace Espace2.ClassesImportantes

{

class B { }

}

namespace ExempleNamespace

{

class Program

{

static void Main(string[] args)

{

Espace1.A p1 = new Espace1.A();

Espace2.ClassesImportantes.A p2 = new Espace2.ClassesImportantes.A();

Espace2.Divers.A p3 = new Espace2.Divers.A();

Espace2.ClassesImportantes.B p4;

}

}

}

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.

**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 nom, sans avoir à les qualifier par leur nom complet (espace de noms + noms).

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);

}

}

}

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](https://msdn.microsoft.com/fr-fr/library/mt472912(v=vs.110).aspx) de l’aide en ligne MSDN, 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.

## Accessibilité des membres d’une classe

Rappel : la portée d’un identificateur est la région du programme dans laquelle cet identificateur 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

private float \_dureeVie;

private float \_poids;

// Propriété publique

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

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 de 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.

## Les propriétés

### 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 2 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 2 accesseurs **get** et **set** d’une propriété ne sont pas obligatoires. Si on ne met pas d’accesseur set, la propriété est alors en lecture seule.
* On peut mettre un modificateur d’accès devant un des 2 accesseurs. Typiquement, on peut rendre l’accesseur set protégé, de façon à ce 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 sur uniquement un des deux accesseurs.

NB/ Pour respecter le principe d’encapsulation, attention à ne rendre les propriétés accessibles en écriture que lorsque c’est nécessaire. La plupart des classes du .net framework fournissent des propriétés en lecture seule.

### 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. :

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 façon à ce qu’elle soit en lecture seule. Ex :

public float DureeVie { get; }

### 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 2 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 le 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 en lecture seule (sans bloc set).**

## 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.

Les modificateurs d’accès permettent l’encapsulation, en ne rendant publiques que les propriétés et méthodes qui permettent de manipuler l’objet d’un point de vue extérieur.

**Tous les champs doivent rester privés !**

Les propriétés doivent être en lecture seule (pas de bloc set) lorsque c’est possible.

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.

## 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 d'une fonction ne fait pas partie de la 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 : nous avons vu plus haut la méthode String.Format, qui possède plusieurs surcharges.

## Champs constants / en lecture seule

Un champ constant est un champ statique dont la valeur ne peut jamais changer. Sa valeur est définie à la compilation.

class Math

{

public const double PI = 3.14159265358979323846;

}

Un littéral doit être utilisé pour initialiser un champ constant.

Un champ en lecture seule est 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.

class MaClasse

{

public readonly int \_valRO;

public MaClasse(int val)

{

\_valRO=val;

}

}

## Constructeurs

Un *constructeur* est une méthode qui initialise l'état d'un objet au moment de son instanciation avec new.

Caractéristiques d’un constructeur :

* Il porte le même nom que la classe
* Il peut recevoir des paramètres (fournis au moment de l’instanciation de l’objet avec new)
* Il n’a pas de type de retour (même pas void)
* Il peut être surchargé
* 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. Ce constructeur par défaut ne fait aucun traitement, mais permet d'instancier la classe.
* Si on définit au moins un constructeur, le compilateur ne génère pas de constructeur par défaut.

L’exemple ci-dessous montre une classe avec 3 constructeurs :

using System;

namespace SyntaxeCSharp

{

class Personne

{

private string \_nom;

private int \_age;

public Personne() // constructeur sans argument

{

\_nom = "Nestor";

\_age = 42;

}

public Personne(int age) // constructeur avec un argument de type int

{

\_nom = "Nestor";

\_age = age;

}

public Personne(int age, string nom)

{

\_nom = nom;

\_age = age;

}

}

class Program

{

static void Main(string[] args)

{

var p1 = new Personne ();

var p2 = new Personne (23);

var p3 = new Personne (17, "Paul");

}

}

}

### Objet courant (référence this)

Dans une méthode, on appelle « Objet Courant », l'objet sur lequel a été appelée la méthode. On peut le désigner par le mot clé **this**.

Dans l'exemple ci-dessous, quand on invoque p**1.Compare(p2)**, dans la méthode Compare, l'objet courant est p**1**.

using System;

namespace ObjCourant

{

class Personne

{

private string prenom;

// Constructeur

public Personne(string prenom)

{

this.prenom = prenom;

}

public bool Compare(Personne p)

{

return (p.prenom == this.prenom);

}

}

class Program

{

static void Main()

{

var p1 = new Personne("Robert");

var p2 = new Personne("John");

Console.WriteLine("p1 et p2 se ressemblent ? {0}", p1.Compare(p2));

}

}

Dans cet exemple, this désigne l’instance courante de personne et this.nom, la valeur de 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.

### 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 également le mot clé this :

// Constructeur de la classe MaClasse, qui appelle le constructeur par défaut

public MaClasse(int val):this()

{

}

Exemple de mise en œuvre :

public class Personne

{

private string \_prenom;

private string \_nom;

private DateTime \_dateNais;

public Personne(string prenom, string nom)

{

\_prenom = prenom;

\_nom = nom;

}

// Ce constructeur appelle le premier pour initialiser le prénom et le nom

public Personne(string prenom, string nom, DateTime dateNais) : this(prenom, nom)

{

this.\_dateNais = dateNais;

}

}

### Initialiseurs d’objets

Avec l’exemple précédent, on peut initialiser les champs des objets Personne grâce aux constructeurs qu’on a fournis :

var p1 = new Personne("Harry", "Cover");

var p2 = new Personne("Jacques", "Ouille", new DateTime(1245, 11, 25));

Si les données sont accessibles via des propriétés en écriture, C# propose également une autre technique pour initialiser des objets : les initialiseurs d’objets :

public class Personne

{

public string Prenom { get; set; }

public string Nom { get; set; }

public DateTime DateNais { get; set; }

}

class Program

{

static void Main(string[] args)

{

// Utilisation d’initialiseurs d’objets

var p1 = new Personne { Prenom = "Harry", Nom = "Cover" };

var p2 = new Personne { Prenom = "Jacques", Nom = "Ouille",

DateNais = new DateTime(1245, 11, 25));

}

}

Les initialiseurs permettent d’affecter une ou plusieurs propriétés en les nommant, et en les séparant par des virgules. On n’a ainsi pas besoin d’écrire plusieurs constructeurs.  
Noter qu’un initialiseur utilise des accolades et non des parenthèses, car ce n’est pas un appel de méthode.

## Membres statiques

**Un *champ statique*** (appelée aussi *variable de classe*) n'existe qu'en un seul exemplaire, partagé par tous les objets créés à partir de la classe (même si aucun objet n'existe).

Pour rappel, les constantes sont des champs statiques.

**Une *méthode statique*** (appelée aussi *méthode de classe*) est une méthode indépendante de toute instance spécifique de classe. Elle n’a accès qu’aux champs statiques de la classe.  
Idem pour les propriétés statiques.

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));

**Classe statique**

On peut déclarer toute une classe comme statique, en spécifiant le mot clé static devant son nom. Dans ce cas, elle ne peut contenir que des membres statiques, et ne peut pas être instanciée.

Une classe non statique peut en revanche contenir des membres statiques et des membres non statiques.

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);

MaClasse m = new MaClasse();

Console.WriteLine(MaClasse.NbObjets);

}

}

}

Cet exemple affiche 0 puis 1.

### Constructeur statique

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("Création d'une couleur");

}

// Constructeur ordinaire

public Couleur(string c)

{

Console.WriteLine(c);

}

}

class Program

{

static void Main(string[] args)

{

var c = new Couleur("Vert");  
 }  
}

Sortie console :

Création d’une couleur

Vert

# 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.

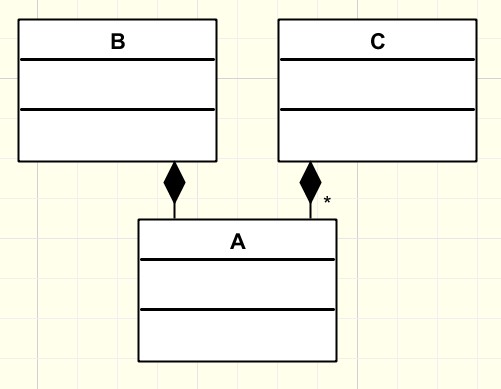
**La composition** décrit la situation où un objet est lui-même constitué d'objets d'autres classes. Par exemple un livre est constitué de différents chapitres, une unité centrale est composée d’une carte mère, d’un processeur, de barrettes de RAM…

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

class A { }

class B

{

 private A \_objet;

public B()

{

\_objet = new A();

}

}

class C

{

private A[] \_objets;

public C(int nb)

{

\_objets = new A[nb];

}

}

Les classes ci-dessus ont les relations suivantes :

* Un objet de la classe B est constitué d'un objet de la classe A
* Un objet de la classe C est constitué de plusieurs objets de la classe A.

**L’agrégation** est semblable à la composition, sauf que les objets agrégés peuvent exister sans l’agrégat, et être partagés entre plusieurs agrégats. Dans le code, cela se traduit par le fait que les objets agrégés sont instanciés en dehors de la classe agrégat.

En pratique, même si la relation entre 2 classes est une agrégation du point de vue du sens, il est courant de l’implémenter comme une composition, sans que cela pose problème.

# L’héritage (la 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

}

}

## 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 désigne l’instance courante d’une classe, alors que base désigne la classe ancêtre d’une classe.

## 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.

### 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().

### 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 !

## 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.

## 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.

# Classes abstraites et interfaces

## Les classes et méthodes abstraites

### 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.

### 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.

### 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);

}

## Les interfaces

### 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 toutes 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.

### 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.

### 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.

### Polymorphisme d’interface

Toutes 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.

# Classe Object, boxing, conversion de type

## Boxing / 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 petite explication pour apprécier ce qui se passe vraiment :  
d est un type valeur qui réside donc sur la pile. Si la référence à l’intérieur de o faisait directement référence à d, elle ferait référence à la pile, ce qui créerait un défaut de sécurité potentiel, si bien que cela est interdit. Par conséquent, le runtime réserve un bloc de mémoire sur le tas, y copie la valeur du réel d, puis référence l’objet o à cette copie. Cette copie automatique d’un élément de la pile vers le tas est appelée conversion boxing. Elle est illustrée par le schéma suivant :

**Pile**

double d

12.34

Object o

@

12.34

**Tas**

**Boxing**

**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

**Pile**

double d

12.34

Object o

@

12.34

**Tas**

**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.

## 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

}

**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 opérateur 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;

}

var 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.

## 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**](https://msdn.microsoft.com/fr-fr/library/system.type(v=vs.110).aspx), 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. Pour cela, 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 de 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.

# Les classes conteneurs

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

## Les 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

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.

## 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](https://msdn.microsoft.com/query/dev14.query?appId=Dev14IDEF1&l=FR-FR&k=k(System.Collections);k(TargetFrameworkMoniker-.NETFramework,Version%3Dv4.5.2);k(DevLang-csharp)&rd=true).

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

## Introduction 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 stockées 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 une paire de valeurs dont le type est paramétrable :

namespace Generique

{

class Paire<T> // classe générique : T est un type paramétrable

{

private T \_premier; // les deux champs sont du même type

private T \_second;

public Paire(T prem, T sec) // constructeur

{

\_premier = prem;

\_second = sec;

}

public override string ToString()

{

return "(" + \_premier.ToString() + "," + \_second.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 2 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)

{

// paire d'entiers

Paire<int> p1 = new Paire<int>(7, 2);

Console.WriteLine(p1.ToString());

// paire de chaînes

Paire<string> p2 = new Paire<string>("premier", "second");

Console.WriteLine(p2.ToString());

// paire de paires d’entiers

Paire<Paire<int>> p3;

p3 = new Paire<Paire<int>>(new Paire<int>(2, 4), new Paire<int>(5, 7));

Console.WriteLine(p3.ToString());

}

}

}

Sortie console :

(7,2)

(premier,second)

((2,4),(5,7))

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 Paire pour chaque type réel demandé.

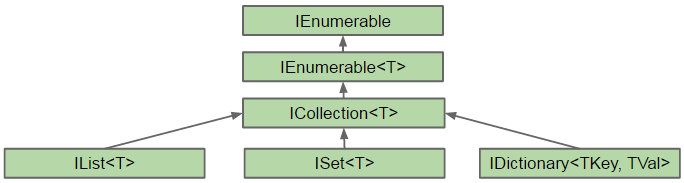
NB/ on peut imbriquer les génériques, comme le montrent les dernières lignes de code.

## Interfaces 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);

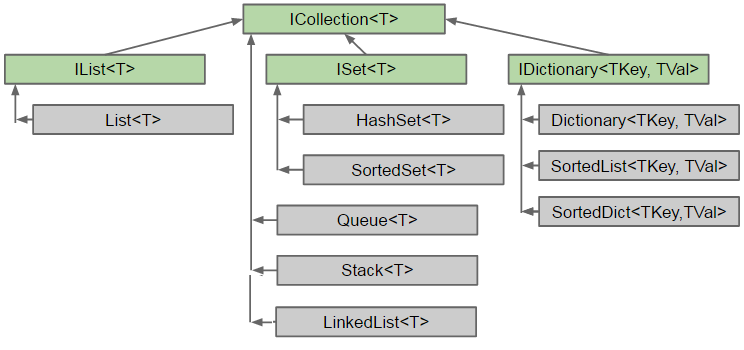
}

}

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.

## Collections 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.

## Enumérations des collections

Nous avons vu plus haut que, pour être énumérable, une classe conteneur doit implémenter l’interface S**ystem.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](https://msdn.microsoft.com/fr-fr/library/dscyy5s0(v=vs.110).aspx)) :

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.

# Délégués et événements

## Délégués

**Un délégué** est un représentant d’une ou plusieurs méthodes. Lorsqu’il exécute un délégué, le runtime exécute en fait les méthodes représentées par ce délégué. Les délégués sont semblables aux pointeurs de fonctions du C++.

### Création et exécution d’un délégué

Dans l’exemple qui suit, on souhaite fournir un moyen de trier un tableau d’animaux de différentes façons, sans connaître à l’avance la méthode de comparaison entre 2 animaux (un tri est toujours basé sur une méthode de comparaison entre 2 éléments).

Voici comment faire cela au moyen d’un délégué :

Namespace Delegues

{

public class Animal

{

public string Nom { get; set; }

public double DureeVie { get; set; }

public override string ToString()

{

return String.Format("{0} vit {1}ans", Nom, DureeVie);

}

// Trie un tableau d'animaux, en utilisant la méthode de comparaion

// représentée par le délégué passé en paramètre

public static void TrierAnimaux(Animal[] anx, DelegueCompare compare)

{

if (compare == null) return;

bool trié;

do

{

trié = true;

for (int i = 1; i < anx.Length; i++)

{

// Exécution du délégué pour comparer les 2 animaux

if (compare(anx[i - 1], anx[i]) > 0)

{

trié = false;

// permutation des 2 animaux

var temp = anx[i - 1];

anx[i - 1] = anx[i];

anx[i] = temp;

}

}

}

while (!trié);

}

}

// Type délégué pour la comparaison d'animaux

public delegate int DelegueCompare(Animal a1, Animal A2);

}

La méthode TrierAnimaux prend en paramètres :

* Le tableau d’animaux à trier
* Un délégué, qui représente la méthode de comparaison à utiliser pour faire le tri

Dans le corps de cette méthode, l’exécution du délégué appelle automatiquement la méthode de comparaison représentée par le délégué.

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 représentées par le délégué (ici, retour de type int, et 2 paramètres de type Animal).

### Branchement de méthodes à un délégué

Regardons maintenant comment utiliser la méthode de tri créée précédemment :

using System;

namespace Delegues

{

class Program

{

static void Main(string[] args)

{

var anx = new Animal[5];

anx[0] = new Animal { Nom="Souris", DureeVie = 1.5 };

anx[1] = new Animal { Nom="Tortue", DureeVie = 110 };

anx[2] = new Animal { Nom = "Chien", DureeVie = 12 };

anx[3] = new Animal { Nom = "Vache", DureeVie = 17 };

anx[4] = new Animal { Nom = "Corbeau", DureeVie = 50 };

DelegueCompare comparateur = null; // Déclaration d'un délégué

Console.WriteLine("Tri par nom :");

comparateur += CompareNoms; // Branchement d'une méthode de comparaison

Animal.TrierAnimaux(anx, comparateur); // Tri du tableau

AfficherAnimaux(anx); // Affichage de son contenu

Console.WriteLine();

Console.WriteLine("Tri par durée de vie :");

comparateur -= CompareNoms; // On débranche la comparaison par nom

comparateur += CompareDureesVie; // et on branche la comparaison par durée de vie

Animal.TrierAnimaux(anx, comparateur); // Tri du tableau

AfficherAnimaux(anx); // Affichage de son contenu

Console.ReadKey();

}

// Affiche le contenu du tablau d'animaux passé en paramètre

static void AfficherAnimaux(Animal[] anx)

{

for (int i =0; i< anx.Length; i++)

Console.WriteLine(anx[i].ToString());

}

// Différentes méthodes de comparaison d'animaux

static int CompareNoms(Animal a1, Animal a2)

{

return a1.Nom.CompareTo(a2.Nom);

}

static int CompareDureesVie(Animal a1, Animal a2)

{

return a1.DureeVie.CompareTo(a2.DureeVie);

}

}

}

Les méthodes CompareNoms et CompareDureesVie sont 2 méthodes de comparaisons d’animaux qu’on souhaite utiliser pour trier le tableau initialisé au début de la méthode Main.

Dans cette méthode Main, on crée un délégué nommé « comparateur », avec le type défini précédemment :

DelegueCompare comparateur = null;

La ligne suivante (comparateur += CompareNoms) permet de dire que la méthode CompareNoms est représentée par le délégué comparateur. Cette méthode sera donc appelée lorsque le délégué sera exécuté.

On peut ensuite appeler la méthode de tri, en lui passant en paramètre le tableau d’animaux et le délégué :

Animal.TrierAnimaux(anx, comparateur);

Le bloc de code suivant débranche la méthode CompareNoms du délégué au moyen de l’opérateur -=, et lui branche la méthode CompareDureesVie. Puis il appelle de nouveau la méthode de tri.

Voici le résultat obtenu à l’écran :

Tri par nom :

Chien vit 12 ans

Corbeau vit 50 ans

Souris vit 1,5 ans

Tortue vit 110 ans

Vache vit 17 ans

Tri par durée de vie :

Souris vit 1,5 ans

Chien vit 12 ans

Vache vit 17 ans

Corbeau vit 50 ans

Tortue vit 110 ans

**Autre syntaxe :**

La syntaxe que nous avons utilisée pour créer un délégué et lui brancher une méthode :

DelegueCompare comparateur = null;   
comparateur += CompareNoms;

… est valable depuis C#2.0.

Il existait en C#1.0 une autre syntaxe qu’on rencontre encore souvent, et qu’il est bon de connaître :

DelegueCompare comparateur = null;  
comparateur += new DelegueCompare(ComapreNoms);

Cette syntaxe est notamment utilisée dans le code généré automatiquement par Visual Studio pour associer une méthode à un évènement émis par un contrôle graphique.

**Intérêt des délégués :**

Un délégué est un bon moyen de découpler le code qui exécute (lance) un traitement, du code qui implémente ce traitement.

Dans l’exemple précédent, la méthode TrierAnimaux trie les animaux sans savoir comment ils seront comparés entre eux. Et surtout, l’implémentation de la comparaison est fournie par une classe complètement étrangère à la classe Animal (ici la classe Program).

Les délégués sont notamment (et surtout) utilisés pour exécuter des gestionnaires d’évènements.

## Evénements

La notion d’évènement est très utilisée dans le .net framework, notamment avec les composants d’interfaces graphiques (boutons, zones de saisies, fenêtres…)

Exemples d’événements : le clic sur un bouton, la prise de focus par une zone de saisie, la fermeture d’une fenêtre modale…etc.

L'utilisation des événements n'est pas limitée aux interfaces graphiques. D'une manière générale, ils fournissent un moyen pour des objets de signaler (=notifier) quelque-chose à leur environnement extérieur.

Les objets de l’environnement extérieur doivent s’abonner à l’évènement s’ils souhaitent pouvoir y réagir.

Ex : Un bouton placé dans une fenêtre émet un évènement « click » lorsqu’on clique dessus avec la souris. Dans le code lié à cette fenêtre, on peut s’abonner à cet évènement pour exécuter une action (ex : charger des données, faire un calcul, afficher ou masquer une zone de saisie…etc.)

L’objet qui émet l’évènement n’a pas besoin de connaître les objets qui y réagiront. Ceci permet de rendre le code modulaire, car les objets émetteurs d’évènements sont indépendants et peuvent être réutilisés partout. C’est le cas des classes qui modélisent les composants d’interfaces graphiques.

### 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**»

Exemple : abonnement à l’évènement Click d’un bouton dans une application WPF



On souhaite qu’un clic sur le bouton Afficher modifie le libellé qui est à côté., et qu’un clic sur le bouton Fermer ferme la fenêtre.

Pour cela, on peut ajouter le code suivant dans le fichier MainWindows.xaml.cs :

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent(); // Code généré automatiquement par VS

// Branchement de gestionnaires aux évènements Click

// des boutons Afficher et Fermer

btnAfficher.Click += BtnAfficher\_Click;

btnFermer.Click += BtnFermer\_Click;

}

private void BtnAfficher\_Click(object sender, RoutedEventArgs e)

{

// Affichage d'un message dans le libellé

lblAffichage.Content = "Tu as cliqué sur le bouton " + ((Button)sender).Content;

}

private void BtnFermer\_Click(object sender, RoutedEventArgs e)

{

this.Close(); // Fermeture de la fenêtre

}

}

Dans le constructeur de la fenêtre MainWindow, après l’appel au code d’initialisation du visuel généré automatiquement par Visual Studio, on branche 2 gestionnaires pour répondre aux évènements Click émis par les boutons Afficher et Fermer.

btnAfficher.Click et btnFermer.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.

### 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, la vue du document défile jusqu’au titre cliqué. On peut imaginer modéliser cela de la façon suivante (juste pour l’exemple, je ne sais pas du tout comment cela est modélisé en réalité) :

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 :

namespace Exemples

{

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.

## Méthodes anonymes et expressions lambda

### 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.

### 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.

# Requêtes LINQ

## 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 paires 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 »

## Les 2 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](https://msdn.microsoft.com/fr-fr/library/bb882642.aspx) 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.

## Opérateurs 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 3 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());

**Tri**

// Chat le plus vieux

var ani = anx.OfType<Chat>().OrderBy(a => a.DateNaissance).First();

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.

# 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.

## Méthodes 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 string Inverser(this string s)

// le mot clé this, suivi du type à étendre caractérise une méthode d’extension

{

char[] ar = s.Reverse().ToArray();

return new string(ar);

}

}

class Program

{

static void Main(string[] args)

{

string s = "hello!";

Console.WriteLine(s.Inverser()); // Renvoie “!olleH”

}

}

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));

## Surcharge 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 :

using System;

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

## Indexeurs

Un indexeur permet d’accéder à un objet comme à un tableau. Il est donc en général implémenté dans une classe qui encapsule une collection.

Pour l’implémenter, il faut définir une méthode **this[]** dont le paramètre est la variable qui sert d'indice. Par exemple, l'indexeur ci-dessous utilise un indice entier et retourne un entier. La partie get gère l'accès en lecture, et la partie set l'accès en écriture.

class MaClasse

{

int this[int idx]

{

get{ ... }

set{ ... }

}

}

Ci-dessous un exemple montrant la création de deux indexeurs dans une classe nommée **Tableau**.

using System;

using System.Collections;

namespace SyntaxeCSharp

{

class Tableau

{

private ArrayList \_tab;

public Tableau()

{

\_tab = new ArrayList();

}

// Indexeur avec indice de type entier et renvoyant une chaîne

public string this[int idx]

{

get

{

return (string)\_tab[idx];

}

set

{

// Si l’élément d’indice idx existe, on modifie sa valeur chaîne

if (idx < \_tab.Count)

\_tab[idx] = value;

else

{

// sinon, on crée de nouveau éléments jusqu’à l’indice idx  
 while (idx >= \_tab.Count) \_tab.Add(value);

}

}

}  
  
 // Indexeur avec indice de type chaîne et renvoyant un entier

public int this[string str]

{

get

{

for (int i = 0; i < \_tab.Count; i++)

{

if ((string)\_tab[i] == str) return i;

}

return -1;

}

}

}

class Programme

{

static void Main()

{

Tableau tab = new Tableau();

tab[2] = "abc";

tab[1] = "def";

Console.WriteLine(T1[0]);

Console.WriteLine("Indice de \"def\" : {0}", tab["def"]);

}

}

}

## Attributs

Les attributs permettent d’associer des métadonnées à des éléments de code (types, méthodes, propriétés, etc.). Les métadonnées peuvent être des instructions de compilateur ou des descriptions de données.

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 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’arboresence.

Exemple : nous allons charger le fichier xml ci-dessous :

<?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)

{

XmlSerializer deserializer = new XmlSerializer(typeof(Data));

Data data;

using (var fs = new FileStream("Voitures.xml", FileMode.Open))

{

data = (Data)deserializer.Deserialize(fs);

}

foreach (var v in data.Voitures)

Console.WriteLine("{0} : {1} {2} de {3}",

v.Id, v.Marque, v.Modèle, v.Année);

Console.ReadKey();

}  
 }

[XmlRoot("Voitures")]

public class Data

{

[XmlElement("Voiture")]

public List<Voiture> Voitures { get; set; }

}

[XmlRoot("Voiture")]

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 XmlRoot, XmlElement et XmlAttribute placés au-dessus des différentes classes et propriétés permettent de faire le lien entre les éléments et attributs xml, et les classes et propriétés C#.

Le chargement du fichier dans la classe Data 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 classe Data en appelant la méthode Serialize du XmlSerializer.

# Architecture en couches

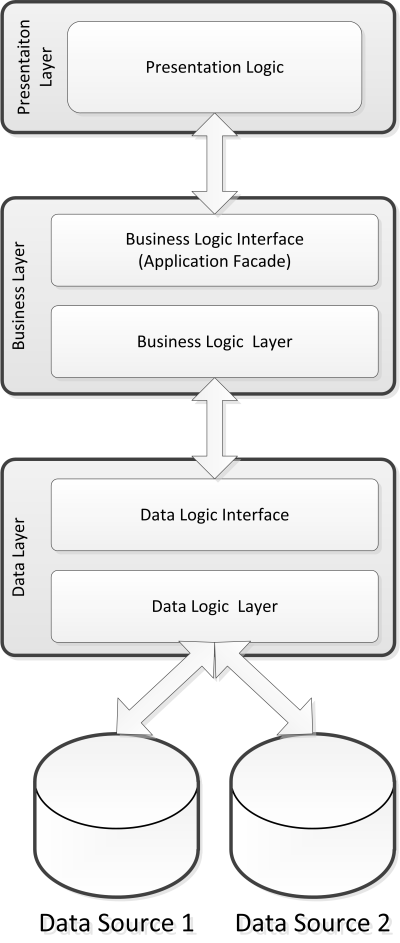
## Caractéristiques

Une architecture en couche se caractérise par le partage des responsabilités entre plusieurs couches bien définies. Une architecture en couches classique se compose de :

* Une couche de présentation
* Une couche de logique métier (BLL - Business Logic Layer), qui peut être exposées sous différentes formes (services WCF internes ou exposés à l’extérieur, services REST…)
* Une couche d'accès aux données (DAL - Data Access Layer)

Une couche ne connaît que la couche immédiatement en dessous d'elle :

* La couche de présentation ne connaît pas la couche d'accès aux données, mais uniquement la couche Business
* La couche Business, ne connaît pas de la couche de présentation, mais uniquement la couche d'accès aux données.
* La couche d'accès aux données ne connaît aucune autre couche.



## Intérêt

**Souplesse et évolutivité**

Bien que ce découpage puisse paraître assez lourd, voire contraignant, il apporte beaucoup de souplesse et d’évolutivité à l’application. Exemples :

* On peut changer de système de gestion de base de données relativement facilement, car seule la couche DAL est à faire évoluer. L’application peut même être compatibles avec plusieurs SGBD différents, pour peu qu’elle fournisse une couche DAL adaptée à chacun.
* On peut créer plusieurs couches de présentations utilisant des technologies différentes (ASP.NET, HTML/JS, WPF…), adaptées à différents types de terminaux ou d’usages, sans que cela n’impacte la couche DAL et la logique métier. Seules les accès à la logique métier (les protocoles de communication) sont à adapter.

**Testabilité**

Ce découpage facilite également les tests unitaires, c’est-à-dire ciblant de petites portions de code.

Dans une application monolithique, on est contraint à simuler des interactions avec l'interface graphique pour pouvoir tester des règles de gestion métiers. Dans une architecture en couches, chaque couche est isolée et peut donc être testée indépendamment. Le code de test peut facilement remplacer l’interface visuelle.

## Mise en œuvre

Le découplage entre les couches est réalisé au moyen d’interfaces et du principe d’injection de dépendances :

**Interfaces**

Une couche publie ce qu'elle est capable de faire sous forme d’une interface. La couche qui est au-dessus possède une référence à cette interface, mais sans savoir qui l’implémente. Ainsi, il n’y a pas de couplage fort entre les 2 couches.

**Injection de dépendances**

La mise en relation d’une couche avec quelque-chose qui implémente l’interface qu’elle consomme, est réalisée de façon dynamique (c’est à dire au moment de l’exécution), par une brique spécialisée. Celle-ci se base sur des fichiers de configuration, qui décrivent les différentes briques logicielles disponibles, et leur rôle.

La relation d’une couche avec une autre n’est donc pas réalisée en dur dans le code de l’application, mais décrite dans des fichiers de configuration externes, et réalisée dynamiquement par une brique logicielle.

Ce mécanisme facilite aussi la mise en place des tests : on peut en effet par exemple coder une couche logicielle qui remplace la couche de présentation, et qui pilote tous les scénarios de tests qu’on souhaite réaliser. C’est ce que l’on appelle un mock.

**Frameworks**

Les briques logicielles permettant de réaliser l’injection de dépendance sont appelées « conteneurs IoC »

IoC est l’acronyme de « Inversion of Control », et désigne un modèle d’architecture dans lequel l'exécution de l’application n'est plus sous son contrôle direct, mais sous celui d'une couche logicielle sous-jacente (un framework).

[Prism](https://github.com/PrismLibrary/Prism) est un des framework les plus utilisés pour implémenter une architecture IoC est Il a été créé par Microsoft, et est désormais open source et maintenu par la communauté. Il est décliné pour les applications WPF, UWP et Xamarin.Forms, et compatible avec différents conteneurs IoC, notamment [Unity](https://github.com/unitycontainer/unity) (créé par Microsoft) et MEF.

Prism est un framework vaste qui facilite la mise en œuvre de plusieurs principes d’architecture, dont l’injection de dépendance, le modèle MVVM, les vues composites, le modèle observateur…etc. On trouvera une vue d’ensemble de ses possibilités sur [cette page](http://prismlibrary.readthedocs.io/en/latest/WPF/Appendix-B-Patterns/).

Lorsqu’on utilise un conteneur IoC, l'interface graphique de l’application se lance de manière traditionnelle, puis demande au conteneur une instance de couche business. De la même façon, la couche business demande au conteneur une instance de DAL.

# Références bibliographiques

Documentation en ligne de [Microsoft Developer Network](https://msdn.microsoft.com/fr-fr/library/default.aspx) (MSDN)

Livre « [Visual C# 2010 Etape par étape](https://www.bookeenstore.com/ebook/9782100560943/visual-c-2010-etape-par-etape-john-sharp) » de John Sharp

Site [Developpez.com](https://www.developpez.com/)