2018/2019

Jounad TOURVIEILLE

.Net

Découverte de .Net et C#

Table des matières

[Premiers pas 3](#_Toc534814221)

[Comment créer un programme informatique 3](#_Toc534814222)

[Les frameworks .Net 3](#_Toc534814223)

[Visual Studio 5](#_Toc534814224)

[Les Langages 5](#_Toc534814225)

[Les variables 5](#_Toc534814226)

[Déclaration et affectation 5](#_Toc534814227)

[Les principales variables 6](#_Toc534814228)

[Type référence et type valeur 6](#_Toc534814229)

[Boxing/Unboxing 7](#_Toc534814230)

[Les instructions conditionnelles 8](#_Toc534814231)

[if-else 8](#_Toc534814232)

[switch 9](#_Toc534814233)

[Les boucles 10](#_Toc534814234)

[La boucle for 10](#_Toc534814235)

[La boucle foreach 11](#_Toc534814236)

[La boucle while 11](#_Toc534814237)

[Les instructions break et continue 12](#_Toc534814238)

[Les tableaux 13](#_Toc534814239)

[Portée du code 14](#_Toc534814240)

[Les blocs de code 14](#_Toc534814241)

[La portée d’une variable 14](#_Toc534814242)

[Accessibilité 14](#_Toc534814243)

[Les classes 15](#_Toc534814244)

[Membres 15](#_Toc534814245)

[Constructeur 15](#_Toc534814246)

[Méthodes 15](#_Toc534814247)

[Propriétés 17](#_Toc534814248)

[Instanciation 17](#_Toc534814249)

[Héritage 17](#_Toc534814250)

[Class abstraite 18](#_Toc534814251)

[Délégués 19](#_Toc534814252)

[Les lambda expressions 21](#_Toc534814253)

[Les generics 21](#_Toc534814254)

[Définition 21](#_Toc534814255)

[Méthode 22](#_Toc534814256)

[Classe 22](#_Toc534814257)

[Particularités 22](#_Toc534814258)

[Les exceptions 22](#_Toc534814259)

[Linq 24](#_Toc534814260)

[Introduction 24](#_Toc534814261)

[Linq To Object 24](#_Toc534814262)

[Linq à travers une requête 24](#_Toc534814263)

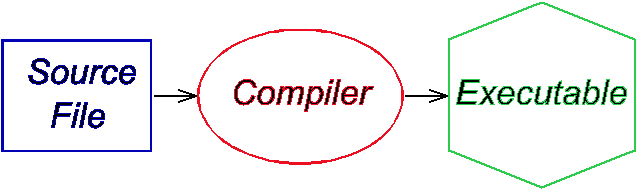
[Linq à travers les lambda expressions 26](#_Toc534814264)

# Premiers pas

## Comment créer un programme informatique

Traditionnellement, pour réaliser un programme informatique, on utilise un langage de programmation. Différents langages existent. Parmi les plus populaires, nous pouvons citer C, C++, Java, C#... Une différence majeure existe entre C, C++ d’un côté et Java, C# de l’autre. Les premiers sont des langages compilés, les seconds interprétés.

Les langages compilés le sont pour une seule plateforme sous-jacente. On les appelle langages natifs.



Langage compilé

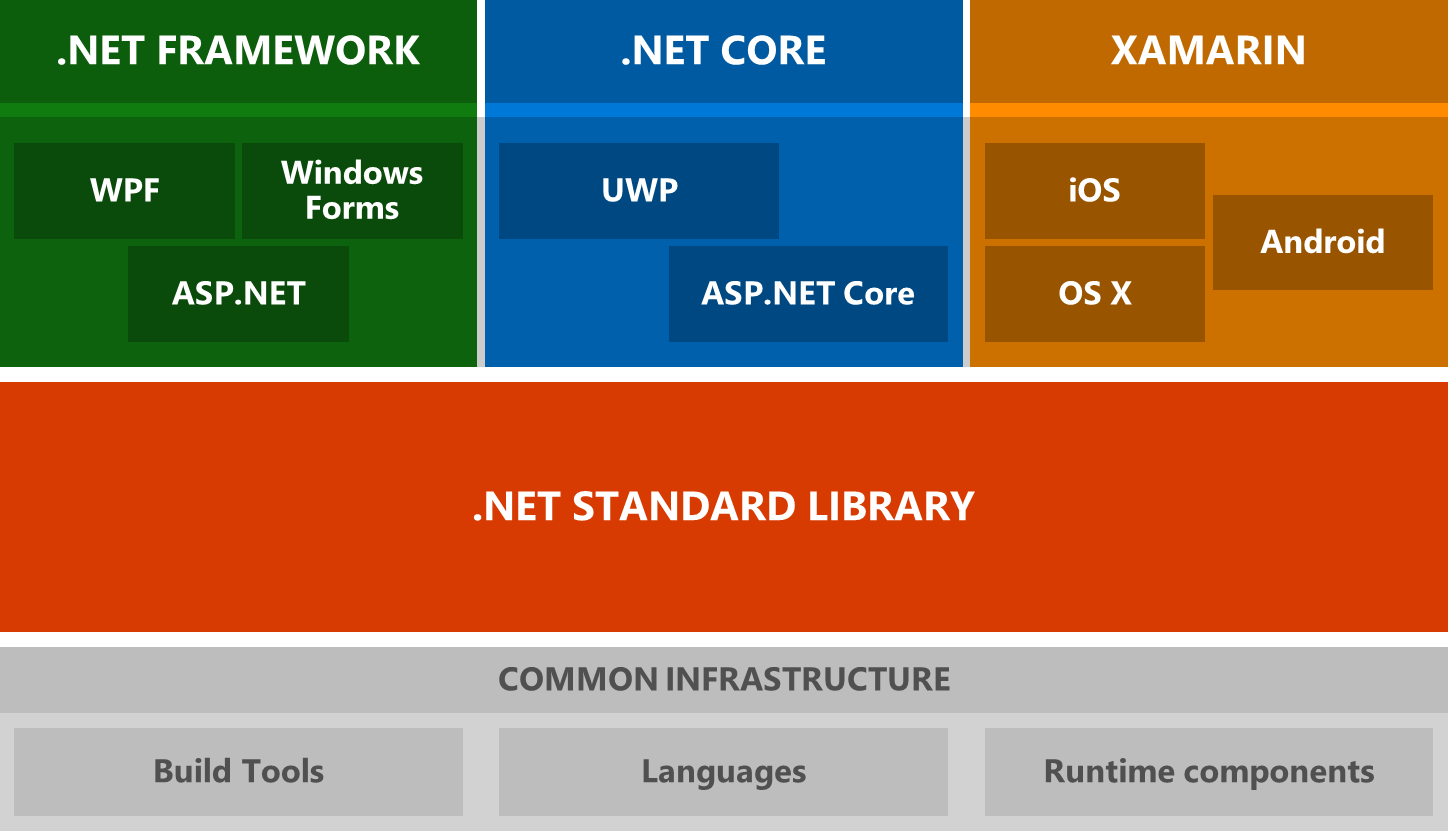
Les langages interprétés reposent sur une machine virtuelle qui se charge d’exécuter le code sur une plateforme spécifique.



Langage interprété

## Les frameworks .Net

Il existe actuellement 3 frameworks .Net principaux : .Net framework, .Net Core et Xamarin. Chacun d’entre eux repose sur un ensemble de librairies appelé la .Net standard library.



.Net Stack

Le .Net framework est le framework historique de .Net. Il est disponible depuis février 2002 dans sa version 1.0. Il a depuis subi de nombreuses améliorations pour arriver aujourd’hui à la version 4.7.



Evolution des frameworks .Net

Le framework .Net core est apparu en juin 2016. Il offre une ouverture et une interopérabilité accrue au .Net framework. C’est le framework à privilégier dans le cadre d’un nouveau projet.

Enfin le framework Xamarin, plus spécialisé, se concentre sur la production d’application mobiles depuis 2011.

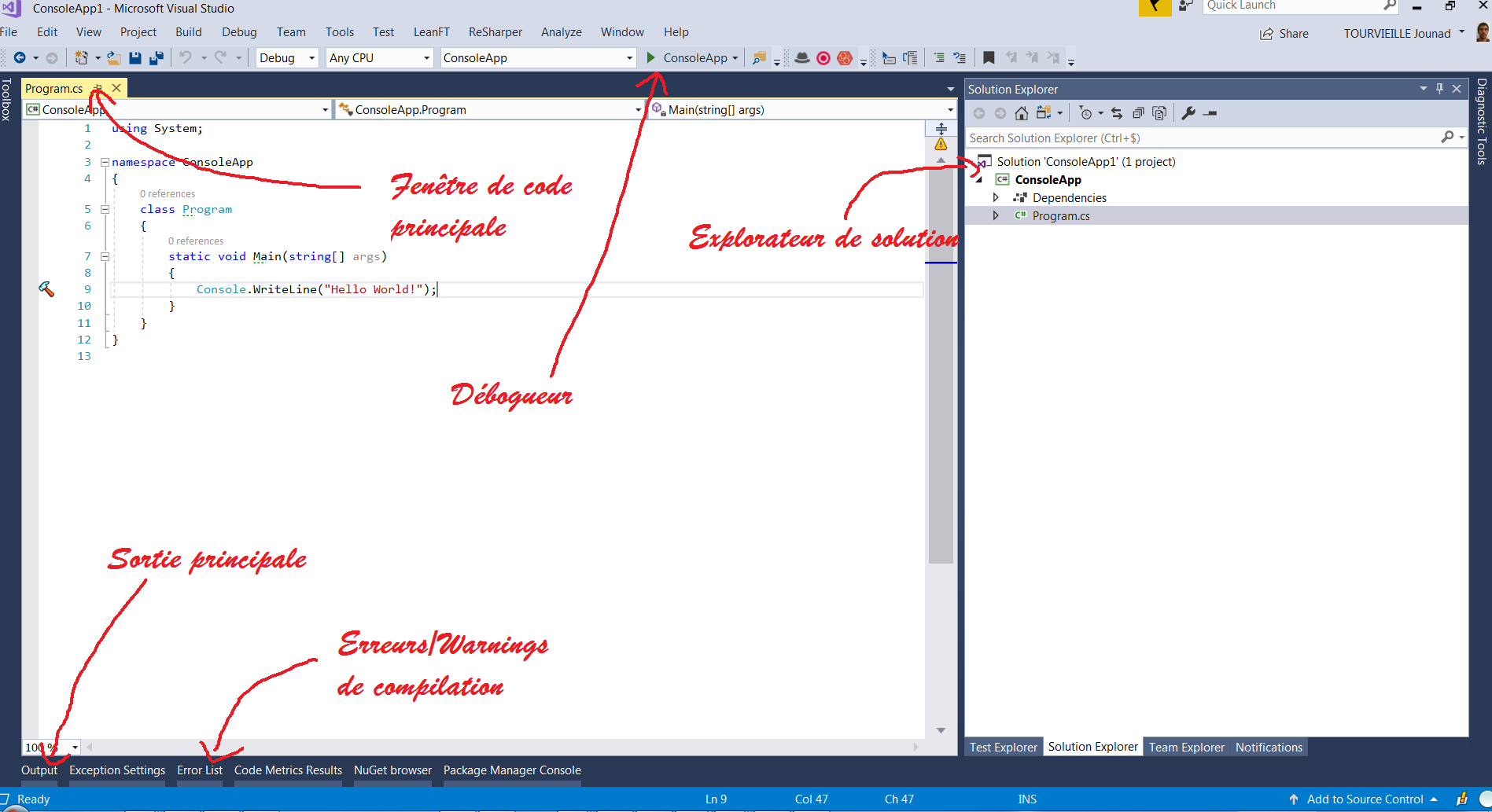
## Visual Studio

Entrons un peu plus dans le vif du sujet. Afin de mettre en œuvre nos applications, nous devons nous reposer sur un IDE. En théorie, il est tout à fait possible de s’en passer, mais la complexité des projets actuels les a rendus indispensables.

L’IDE phare de Microsoft est Visual studio. Ce dernier permet de programmer dans n’importe quel langage de programmation compatible avec les framework dotnet. C’est cet IDE que nous utiliserons par la suite. Il existe plusieurs versions, l’outil étant régulièrement mis-à-jour.

Citons également VS Code, un IDE également fourni par Microsoft, plus modulaire, plus léger et adapté au web. Cet IDE rencontre un beau succès auprès des développeurs « front », toutes technologies confondues.

Voici une copié d’écran avec les principaux espaces de Visual Studio.



Visual Studio

## Les Langages

Une application développée pour la plateforme .Net peut utiliser n'importe quel langage de programmation si celui-ci possède un compilateur produisant des instructions IL (Intermediate Language). Bien qu’il existe un nombre important de langages compatibles, les deux principaux langages utilisés sont **VB.NET**, et surtout **C#**. C’est sur ce dernier, largement répandu, que nous allons nus pencher plus longuement.

## Les variables

### Déclaration et affectation

Une variable est un élément qui contient une information qui, comme son nom l’indique, peut varier. En C#, les variables sont dites typées, c’est-à-dire qu’elles ont un format défini.

Une variable est représentée par son **nom**, caractérisée par son **type** et contient une **valeur**.



Ici, on affecte la valeur 20 à la variable age de type entier.

En général, une variable commence par une minuscule et si son nom représente plusieurs mots, on démarrera un nouveau mot. On appelle ceci le lower camel case.



### Les principales variables

Les principaux types de base du framework .NET sont :

|  |  |
| --- | --- |
| **Type** | **Description** |
| byte | Entier de 0 à 255 |
| short | Entier de -32768 à 32767 |
| int | Entier de -2147483648 à 2147483647 |
| long | Entier de -9223372036854775808 à 9223372036854775807 |
| float | Nombre simple précision de -3,402823e38 à 3,402823e38 |
| double | Nombre double précision de -1,79769313486232e308 à 1,79769313486232e308 |
| decimal | ±1,0 x 10^28 to ±7,9228 x 10^28 avec 28-29 chiffres significatifs |
| char | Représente un caractère unique |
| string | Représente une chaine de caractère |
| bool | Représente une valeur booléenne (vrai ou faux) |

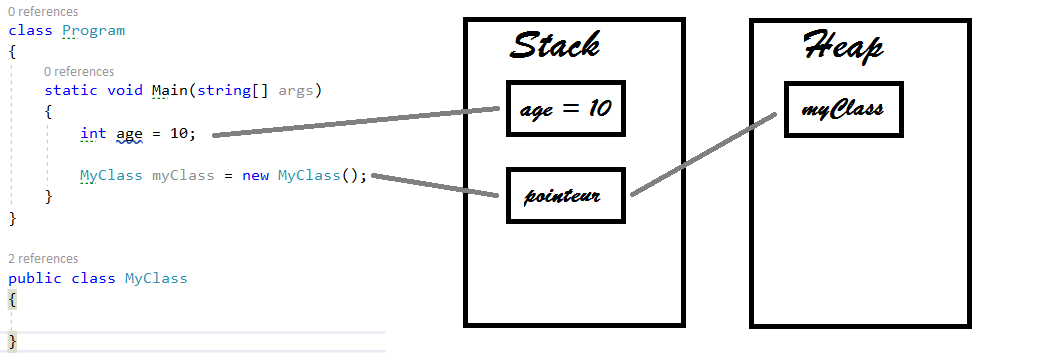
### Type référence et type valeur

Il existe deux familles de variables en C# : les *types référence* et les *types valeur*. Les variables des types valeur contiennent directement leurs données alors que les variables des types référence contiennent des références à leurs données, connues sous le nom d’objets. Avec les types référence, deux variables peuvent faire référence au même objet et, par conséquent, les opérations sur une variable peuvent affecter le même objet référencé par l'autre variable. Avec les types valeur, les variables possèdent leur propre copie de données, et les opérations sur une variable ne peuvent absolument pas affecter l'autre (sauf pour les variables de paramètre *ref* et *out*).

Les types de valeur de C# sont divisés entre *types simples*, *types enum*, *types struct* et *types valeur nullable*. Les types de référence de C# sont encore divisés en *types de classes*, *types d’interfaces*, *types de tableaux* et *types délégués*.

* Types de valeur
  + Types simples
    - Entier signé : *sbyte, short, int, long*
    - Entier non signé : *byte, ushort, uint, ulong*
    - Caractères Unicode : *char*
    - Virgule flottante IEEE : *float, double*
    - Décimale haute précision : *decimal*
    - Booléen : *bool*
  + Types d'enum
    - Types définis par l'utilisateur de la forme *enum E {...}*
  + Types struct
    - Types définis par l'utilisateur de la forme *struct S {...}*
  + Types valeur Nullable
    - Extensions de tous les autres types de valeurs avec une valeur *null*
* Types référence
  + Types de classes
    - Classe de base fondamentale de tous les autres types : *object*
    - Chaînes Unicode : *string*
    - Types définis par l'utilisateur de la forme class *C {...}*
  + Types interface
    - Types définis par l'utilisateur de la forme interface *I {...}*
  + Types de tableaux
    - Uni et multidimensionnels, par exemple *int[] et int[,]*
  + Types délégués
    - Types définis par l'utilisateur de la forme *delegate int D(...)*

Les types référence sont stockés dans un espace mémoire appelé Heap, tandis que les types valeur sont stockés sur la Stack.



Stack VS Heap

### Boxing/Unboxing

Le boxing est la conversion d’un [type valeur](https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/value-types) en type object ou en un type interface implémenté par ce type valeur. Lorsque le CLR exécute un boxing d’un type valeur, il inclut la valeur dans un wrapper, à l’intérieur d’un System.Object, et la stocke sur le tas managé. L'unboxing extrait le type valeur de l'objet. La conversion boxing est implicite ; la conversion unboxing est explicite. Le concept de boxing et de unboxing repose sur la vue unifiée par C# du système de type, dans lequel une valeur de n'importe quel type peut être traitée en tant qu'objet.



## Les instructions conditionnelles

Une condition se construit en utilisant des opérateurs de comparaison. Elle ne peut être que vraie ou fausse. Les principaux opérateurs de comparaison sont listés ci-dessous :

|  |  |
| --- | --- |
| Opérateur | Description |
| == | Egalité |
| != | Différence |
| >= | Supérieur ou égal à |
| <= | Inférieur ou égal à |
| && | Et logique |
| || | Ou logique |
| ! | Négation |

### if-else

Une instruction if identifie l’instruction à exécuter en fonction de la valeur d’une expression *Boolean*.

Il est également possible de spécifier le contraire du cas passant par l’instruction *if*. Il faut pour ceci spécifier la condition *else*.

L’exemple suivant décrit différents usages opérateurs de comparaison.



### switch

switch est une instruction de sélection qui choisit une *section de commutation* unique à exécuter à partir d’une liste de candidats en fonction d’une mise en correspondance de modèle avec l’*expression de correspondance*.

Une instruction switch peut inclure un nombre quelconque de sections de commutation, et chaque section peut contenir une ou plusieurs étiquettes case, comme dans l’exemple ci-dessous. Toutefois, deux étiquettes case ne doivent pas contenir la même expression.

L’étiquette case default spécifie la section de commutation à exécuter si l’expression de correspondance ne correspond à aucune autre étiquette case. En l’absence d’une étiquette case default, si l’expression de correspondance ne correspond à aucune autre étiquette case, le flux de programme traverse l’instruction switch.



## Les boucles

Les instructions d’itération entraînent une exécution des instructions incorporées un certain nombre de fois, selon les critères de terminaison des boucles. Ces instructions sont exécutées dans l’ordre, sauf quand une [instruction de saut](https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/jump-statements) est rencontrée.

### La boucle for

L’instruction for exécute une instruction ou un bloc d’instructions jusqu’à ce qu’une expression booléenne spécifique corresponde à la valeur true.

À tout moment dans le bloc d’instructions for, vous pouvez sortir de la boucle à l’aide de l’instruction [break](https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/break), ou passer à l’itération suivante de la boucle à l’aide de l’instruction [continue](https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/continue).

Chaque instruction for définit des sections *initialiseur*, *condition* et *itérateur*.



Ces trois sections sont facultatives. Le corps de la boucle est une instruction ou un bloc d’instructions.

L’exemple suivant montre l’instruction for avec toutes les sections définies :



* La section **initialiseur** : les instructions figurant dans la section *initialiseur* sont exécutées une seule fois, avant d’entrer dans la boucle. La section *initialiseur* contient au choix
  + La déclaration et l’initialisation d’une variable de boucle locale, qui n’est pas accessible à partir de l’extérieur de la boucle
  + Zéro, une ou plusieurs expressions d’instruction
* La section **condition** : la section *condition*, si elle est présente, doit être une expression booléenne. Cette expression est évaluée avant chaque itération de boucle. Si la section *condition* est absente ou que l’expression booléenne est true, l’itération suivante de la boucle est exécutée ; sinon, la boucle est fermée.
* La section **itérateur** : la section *itérateur* définit ce qui se produit après chaque itération du corps de la boucle. La section *itérateur* contient zéro, une ou plusieurs des expressions d’instruction suivantes, séparées par des virgules

### La boucle foreach

L’instruction foreach exécute une instruction ou un bloc d’instructions pour chaque élément d’une instance du type qui implémente l’interface [System.Collections.IEnumerable](https://docs.microsoft.com/fr-fr/dotnet/api/system.collections.ienumerable) ou [System.Collections.Generic.IEnumerable<T>](https://docs.microsoft.com/fr-fr/dotnet/api/system.collections.generic.ienumerable-1).L’instruction foreach n’est pas limitée à ces types et peut être appliquée à une instance de n’importe quel type répondant aux conditions suivantes :

* comporte la méthode GetEnumerator sans paramètre publique dont le retour est de type classe, struct ou interface,
* le type de retour de la méthode GetEnumerator contient la propriété publique Current et la méthode sans paramètre publique MoveNext, dont le type de retour est [Boolean](https://docs.microsoft.com/fr-fr/dotnet/api/system.boolean).

À tout moment dans le bloc d’instructions foreach, vous pouvez sortir de la boucle à l’aide de l’instruction [break](https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/break), ou passer à l’itération suivante de la boucle à l’aide de l’instruction [continue](https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/continue).



### La boucle while

L’instruction while exécute une instruction ou un bloc d’instructions jusqu’à ce qu’une expression booléenne spécifique corresponde à la valeur true. Dans la mesure où cette expression est évaluée avant chaque exécution de la boucle, une boucle whiles’exécute plusieurs fois ou pas du tout. Cela diffère de la boucle [do](https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/do), qui s’exécute une ou plusieurs fois.

À tout moment dans le bloc d’instructions while, vous pouvez sortir de la boucle à l’aide de l’instruction [break](https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/break).

Vous pouvez passer directement à l’évaluation de l’expression while à l’aide de l’instruction [continue](https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/continue). Si l’expression correspond à true, l’exécution passe à la première instruction de la boucle. Sinon, l’exécution passe à la première instruction après la boucle.



### Les instructions break et continue

L’instruction break termine la boucle englobante ou l’instruction [switch](https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/switch) la plus proche dans laquelle elle figure. Le contrôle est passé à l’instruction qui suit l’instruction terminée, le cas échéant.



L’instruction continue passe le contrôle à l’itération suivante de l’instruction [while](https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/while), [do](https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/do), [for](https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/for) ou [foreach](https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/foreach-in) englobante dans laquelle elle apparaît.

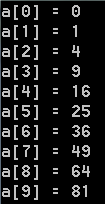


## Les tableaux

Un tableau est une structure de données qui contient un certain nombre de variables qui sont accessibles par des indices calculés. Les variables contenues dans un tableau, également appelé éléments du tableau, sont tous du même type, et ce type est appelé type d’élément du tableau.

Les types tableau sont des types référence, et la déclaration d’une variable tableau réserve simplement un espace pour une référence à une instance de tableau. Les instances de tableau réelles sont créées dynamiquement lors de l’exécution à l’aide de l’opérateur new.





Cet exemple crée et utilise un **tableau unidimensionnel**. C# prend également en charge les **tableaux multidimensionnels**. Le nombre de dimensions d’un type tableau, également appelé **rang** du type tableau, est de un plus le nombre de virgules entre les crochets du type tableau.

## Portée du code

### Les blocs de code

Les blocs de code permettent de grouper plusieurs instructions qui vont s’exécuter dans le même contexte.  
Cela peut être le cas par exemple après un if, nous pourrions souhaiter effectuer plusieurs instructions.

La « portée d’une variable » est la zone de code dans laquelle une variable est utilisable. Elle correspond en général au bloc de code dans lequel est définie la variable.

### La portée d’une variable

La « portée d’une variable » est la zone de code dans laquelle une variable est utilisable. Elle correspond en général au bloc de code dans lequel est définie la variable. Ainsi, une variable locale déclarée dans une méthode sera invisible d’une autre méthode de même niveau.

### Accessibilité

Chaque membre d’une classe a une accessibilité associée, qui contrôle les régions du texte du programme qui sont en mesure d’accéder au membre. Il existe six formes possibles d’accessibilité. Ils sont résumés ci-dessous.

* public
  + Accès non limité
* protected
  + Accès limité à cette classe ou aux classes dérivées de cette classe
* internal
  + Accès limité à l’assembly actuel (.exe, .dll, etc.)
* protected internal
  + Accès limité à la classe conteneur, aux classes dérivées de la classe conteneur ou aux classes dans le même assembly
* private
  + Accès limité à cette classe
* private protected
  + Accès limité à la classe conteneur ou aux classes dérivées du type conteneur dans le même assembly

## Les classes

### Membres

Les membres d’une classe sont des membres statiques ou membres d’instance. Les membres statiques appartiennent à des classes, et les membres d’instance appartiennent à des objets (instances de classes).

Vous trouverez ci-dessous une vue d’ensemble des types de membres qu'une classe peut contenir.

* Constantes
  + Valeurs constantes associées à la classe
* Champs
  + Variables de la classe
* Méthodes
  + Calculs et les actions qui peuvent être effectués par la classe
* Propriétés
  + Actions associées à la lecture et l’écriture des propriétés nommées de la classe
* Indexeurs
  + Actions liées à l’indexation des instances de la classe comme un tableau
* Événements
  + Les notifications qui peuvent être générées par la classe
* Opérateurs
  + Les opérateurs de conversion et d’expression pris en charge par la classe
* Constructeurs
  + Les actions requises pour initialiser les instances de la classe ou la classe elle-même
* Finaliseurs
  + Actions à effectuer avant que les instances de la classe soient abandonnées de façon définitive
* Types
  + Types imbriqués déclarés par la classe

### Constructeur

Un constructeur est déclaré comme une méthode sans aucun type de retour et le même nom que la classe contenante. Les constructeurs d’instance peuvent être surchargés et avoir des paramètres facultatifs. Les constructeurs d’instance sont appelés en utilisant l’opérateur new. Si aucun constructeur d’instance n’est fourni pour une classe, un constructeur vide sans paramètre est fourni automatiquement.

### Méthodes

Une méthode est un membre qui implémente un calcul ou une action qui peut être effectuée par un objet ou une classe. Les méthodes statiques sont accessibles à travers la classe. Les méthodes d’instance sont accessibles via des instances de la classe.

Les méthodes peuvent avoir une liste de paramètres, qui représentent des valeurs ou des références variables passées à la méthode et un type de retour, qui spécifie le type de la valeur calculée et retournée par la méthode. Le type de retour d’une méthode est void si elle ne retourne pas de valeur.

La *signature* d’une méthode doit être unique dans la classe dans laquelle la méthode est déclarée. La signature d’une méthode se compose du nom de la méthode, du nombre de paramètres de type et du nombre, des modificateurs et des types de ses paramètres. La signature d'une méthode n'inclut pas le type de retour.

Les paramètres d’une méthode sont utilisés pour passer des valeurs ou des références variables aux méthodes. Les paramètres d’une méthode obtiennent leurs valeurs réelles à partir des arguments qui sont spécifiés lorsque la méthode est appelée.

Le corps d’une méthode spécifie les instructions à exécuter lorsque la méthode est appelée.

Un corps de méthode peut déclarer des variables qui sont spécifiques à l’appel de la méthode. Ces variables sont appelées variables locales. Une déclaration de variable locale spécifie un nom de type, un nom de variable et éventuellement une valeur initiale.

C# requiert qu’une variable locale soit assignée de manière définitive avant de pouvoir obtenir sa valeur. Par exemple, si la déclaration du i précédent n’inclut pas de valeur initiale, le compilateur signale une erreur pour les utilisations ultérieures de i, car ine serait pas assigné de manière définitive à ces points dans le programme.

Une méthode peut utiliser les instructions return pour retourner le contrôle à son appelant. Dans une méthode retournant void, les instructions return ne peuvent pas spécifier une expression. Dans une méthode avec un type de retour non void, les instructions return doivent inclure une expression qui calcule la valeur de retour.

Une méthode déclarée avec un modificateur statique est une méthode statique. Une méthode statique n’opère pas sur une instance spécifique et permet uniquement d’accéder directement à des membres statiques.

Une méthode déclarée sans un modificateur statique est une méthode d’instance. Une méthode d’instance opère sur une instance spécifique et peut accéder aux membres statiques et d’instance. L’instance sur laquelle une méthode d’instance a été appelée est explicitement accessible en tant que this. Une erreur consiste à faire référence à this dans une méthode statique.



Dans l’exemple ci-dessus, la class Computer possède 2 méthodes : AddTwoNumbers est une méthode d’instance. Elle est appelable via une instance de la classe Computer. En revanche, la méthode MultiplyTwoNumbers est une méthode statique. Elle ne peut pas être appelée via une instance de la classe Computer.

### Propriétés

Les *propriétés* sont une extension naturelle des champs. Les deux sont des membres nommés avec des types associés, et la syntaxe pour accéder aux champs et propriétés est la même. Toutefois, contrairement aux champs, les propriétés ne désignent pas des emplacements de stockage. Au lieu de cela, les propriétés ont des *accesseurs*qui spécifient les instructions à exécuter lorsque les valeurs sont lues ou écrites.

Une propriété est déclarée comme un champ, sauf que la déclaration se termine avec un accesseur get et/ou un accesseur set écrit entre les délimiteurs { et } au lieu de se terminer par un point-virgule. Une propriété qui a un accesseur get et un accesseur set est une *propriété en lecture-écriture*, une propriété qui possède uniquement un accesseur get est une *propriété en lecture seule*, et une propriété qui possède uniquement un accesseur set est une *propriété en écriture seule*.

La classe List<T> déclare deux propriétés, Count et Capacity, qui sont en lecture seule et en lecture-écriture, respectivement. Voici un exemple d’utilisation de ces propriétés.



### Instanciation

La création d’une instance se fait en deux étapes. Tout d’abords, la déclaration, puis l’instanciation.



L'instanciation est l'opération qui consiste à créer un nouvel objet à partir d'une classe, au moyen de l'opérateur new.

### Héritage

Une déclaration de classe peut spécifier une classe de base en faisant suivre les paramètres de nom et de type de classe par un signe deux-points et le nom de la classe de base. L’omission d’une spécification de classe de base revient à dériver du type object.

Une classe hérite des membres de sa classe de base. L’héritage signifie qu’une classe contient implicitement tous les membres de sa classe de base, à l’exception des constructeurs d’instance et statiques et des finaliseurs de la classe de base. Une classe dérivée peut ajouter des membres hérités, mais ne peut pas supprimer la définition d’un membre hérité.



Dans l'exemple ci-dessus, la classe de base de Point3D est Point, et la classe de base de Point est object. L’accès au constructeur parent se fait à travers le mot clé base.

La classe Point3D possède 3 propriétés : x et y par héritage, et z de manière intrinsèque.

### Class abstraite

Le modificateur abstract indique que l’élément en cours de modification a une implémentation manquante ou incomplète.

Dans une déclaration de classe, utilisez le modificateur abstract pour indiquer qu’une classe doit uniquement servir de classe de base pour d’autres classes. Les membres définis comme abstraits, ou inclus dans une classe abstraite, doivent être implémentés par des classes dérivées de la classe abstraite.



Les classes abstraites présentent les caractéristiques suivantes :

Une classe abstraite ne peut pas être instanciée.

Une classe abstraite peut contenir des méthodes et accesseurs abstraits.

Il n’est pas possible de modifier une classe abstraite à l’aide du modificateur [sealed](https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/sealed), car les deux modificateurs ont des significations opposées. Le modificateur sealed empêche qu’une classe soit héritée et le modificateur abstract exige qu’une classe soit héritée.

Une classe non abstraite dérivée d’une classe abstraite doit inclure des implémentations réelles de tous les accesseurs et méthodes abstraits hérités.

## Délégués

Un **type délégué** représente des références aux méthodes avec une liste de paramètres et un type de retour particuliers. Les délégués permettent de traiter les méthodes en tant qu’entités qui peuvent être affectées à des variables et passées comme paramètres. Les délégués sont similaires au concept de pointeurs de fonction dans d’autres langages, mais contrairement aux pointeurs de fonction, les délégués sont orientés objet et de type sécurisé.



Une instance du type délégué Function peut faire référence à toute méthode qui accepte un argument double et retourne une valeur double. La méthode Apply applique une fonction donnée aux éléments d’un double[], et retourne un double[] avec les résultats. Dans la méthode Main, Apply sert à appliquer trois fonctions différentes à un double[].

Un délégué peut faire référence à une méthode statique (comme Square ou Math.Sin dans l’exemple précédent) ou à une méthode d’instance (comme m.Multiply dans l’exemple précédent).

Les délégués peuvent également être créés à l’aide de fonctions anonymes, qui sont des « méthodes inline » créées à la volée. Les fonctions anonymes peuvent voir les variables locales des méthodes qui l’entourent. Par conséquent, l’exemple de multiplicateur ci-dessus peut être écrit plus facilement sans utiliser une classe Multiplicateur.



## Les lambda expressions

Une expression lambda est une [fonction anonyme](https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/statements-expressions-operators/anonymous-methods) que vous pouvez utiliser pour créer des types [délégués](https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/delegates/using-delegates) ou des types d' [arborescence d'expression](https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/concepts/expression-trees/index) . En utilisant des expressions lambda, vous pouvez écrire des fonctions locales qui peuvent être passées comme des arguments ou retournées comme la valeur d'appels de fonction.

Pour créer une expression lambda, vous spécifiez des paramètres d'entrée (le cas échéant) à gauche de l'opérateur lambda [=>](https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/operators/lambda-operator), et vous placez l'expression ou le bloc d'instructions de l'autre côté. Par exemple, l'expression lambda x => x \* x spécifie un paramètre nommé x et retourne la valeur x élevée au carré.

Les parenthèses sont facultatives uniquement si le lambda comporte un paramètre d'entrée ; sinon, elles sont obligatoires. Les paramètres d'entrée sont séparés par des virgules entre parenthèses :



Une instruction lambda ressemble à une expression lambda, mais l'instruction ou les instructions sont mises entre accolades :

(paramètres d’entrée) => { instruction ; }

Le corps d'une instruction lambda peut se composer d'un nombre illimité d'instructions.

Voici un exemple d’expression lambda couplé à Linq :



## Les generics

### Définition

Les generics sont une fonctionnalité du framework .NET apparus avec la version 2. Ils permettent de créer des méthodes ou des classes qui sont indépendantes d'un type. Il est très important de connaître leur fonctionnement car c'est un mécanisme clé qui permet de faire beaucoup de choses, notamment en termes de réutilisabilité et d'amélioration des performances. Le framework dotnet regorge de types et méthodes génériques.

### Méthode

Soit une méthode statique retournant la valeur maximale entre deux valeurs passées en paramètre.



Le compilateur détermine le type utilisé pour T d'après les valeurs des arguments. Dans le cas précédent, il génère deux versions surchargées de la méthode statique.



### Classe

Les classes génériques encapsulent des opérations qui ne sont pas spécifiques à un type de données particulier.Les classes génériques sont le plus souvent utilisées avec les collections telles que les listes liées, les tables de hachage, les piles, les files d’attente, les arborescences, etc. Les opérations telles que l’ajout et la suppression d’éléments dans la collection sont exécutées fondamentalement de la même manière quel que soit le type des données stockées.

Pour la plupart des scénarios qui nécessitent des classes de collections, l’approche recommandée consiste à utiliser celles fournies dans la bibliothèque de classes .NET.

En général, vous créez des classes génériques en commençant par une classe concrète existante et en changeant des types en paramètres de type individuellement jusqu’à ce que vous atteigniez l’équilibre optimal au niveau de la généralisation et de la facilité d’utilisation.

### Particularités

Il est possible d’appliquer des contraintes au type des méthodes/classes génériques.

Les contraintes informent le compilateur sur les fonctionnalités que doit avoir un argument de type. Sans contrainte, l’argument de type peut être n’importe quel type.

Plusieurs contraintes peuvent être appliquées au même paramètre de type, et les contraintes elles-mêmes peuvent être des types génériques



## Les exceptions

Les fonctionnalités de gestion des exceptions du langage C# vous aident à gérer les situations inattendues ou exceptionnelles qui se produisent lorsqu’un programme est en cours d’exécution. La gestion des exceptions utilise les mots clés try, catch et finally pour tenter des actions susceptibles de ne pas réussir, pour gérer les défaillances lorsque vous pensez que c’est justifié et pour nettoyer ensuite les ressources. Les exceptions peuvent être générées par le common language runtime (CLR), par .NET Framework ou des bibliothèques tierces, ou par le code de l’application. Les exceptions sont créées avec le mot clé throw.

Dans de nombreux cas, une exception peut être levée non pas par une méthode appelée directement par votre code, mais par une autre méthode plus loin dans la pile des appels. Dans ce cas, le CLR déroule la pile à la recherche d’une méthode avec un bloc catch pour le type d’exception concerné et exécute le premier bloc catch de ce type qu’il trouve. S’il ne trouve pas de bloc catch dans la pile des appels, il termine le processus et affiche un message à l’utilisateur.

Dans cet exemple, une méthode teste la division par zéro et intercepte l’erreur. Sans la gestion des exceptions, ce programme se terminerait avec une erreur DivideByZeroException non gérée.



# Linq

## Introduction

Linq est l’abréviation de Language INtegrated Query. Les outils apportés par Linq sont très appréciés des développeurs. Ils permettent de procéder à des opérations sur des ensembles de données. En pratique, l’utilisation de Linq ressemble fort à l’utilisation d’une requête SQL. En outre, la syntaxe de la requête est vérifiée lors de l’étape de compilation. Ceci constitue un énorme avantage.

Linq prend différentes formes tout en gardant la même syntaxe. Il peut s’appliquer à différentes cibles :

* Linq to Entities : Liée à l’utilisation de l’entity model de .NET.
* Linq to Object : Liée à l’utilisation de collection.
* Linq to SQL : Liée à des requête SQL.
* Linq to Dataset : Liée à l’utilisation de dataset.
* Linq to XML : Liée à l’utilisation de donnée XML.

Nous allons ici détailler l’utilisation de Linq to Object, qui est largement répandu.

## Linq To Object

Linq to Object nous permet d’interagir facilement avec nos collections d’objet. Il existe deux façons d’utiliser Linq : sous forme de requête, ou sous forme de lambda expressions.

### Linq à travers une requête

#### From

Une expression de requête doit commencer par une clause from. Elle spécifie une source de données avec une variable de portée. La variable de portée représente chaque élément consécutif dans la séquence source comme la séquence source parcourue. La variable de portée est fortement typée selon le type d’éléments dans la source de données. Dans l’exemple suivant, comme countries est un tableau d’objets Country, la variable de portée est également de type Country. Étant donné que la variable de portée est fortement typée, vous pouvez utiliser l’opérateur point pour accéder à tous les membres disponibles du type.



Une expression de requête peut contenir plusieurs clauses from. Utilisez des clauses from supplémentaires quand chaque élément de la séquence source est lui-même une collection ou contient une collection. Par exemple, supposons que vous avez une collection d’objets Country et que chacun d’entre eux contient une collection d’objets City nommés Cities. Pour interroger les objets City dans chaque Country, utilisez deux clauses from comme indiqué ici :



#### Group, Select

Une expression de requête doit se terminer par une clause group ou une clause select.

Utilisez la clause group pour générer une séquence de groupes organisée par une clé que vous spécifiez. La clé peut correspondre à tout type de données. Par exemple, la requête suivante crée une séquence de groupes qui contient un ou plusieurs objets Country dont la clé est une valeur char.



Utilisez la clause select pour générer tous les autres types de séquences. Une clause select simple génère simplement une séquence du même type d’objets que les objets contenus dans la source de données.



La clause select peut être utilisée pour transformer des données sources en séquences de nouveaux types. Cette transformation est également appelée *projection*. Dans l’exemple suivant, la clause select *projette* une séquence de types anonymes qui contient uniquement un sous-ensemble des champs dans l’élément d’origine. Notez que les nouveaux objets sont initialisés à l’aide d’un initialiseur d’objet.



#### Where

Utilisez la clause where pour éliminer par filtrage des éléments des données sources selon une ou plusieurs expressions de prédicat. La clause where de l’exemple suivant a un prédicat avec deux conditions.



#### OrderBy

Utilisez la clause orderby pour trier les résultats par ordre croissant ou décroissant. Vous pouvez également spécifier des ordres de tri secondaires. L’exemple suivant effectue un tri principal sur les objets country en utilisant la propriété Area. Il effectue ensuite un tri secondaire en utilisant la propriété Population.



#### Join

Utilisez la clause join pour associer et/ou combiner des éléments d’une source de données avec des éléments d’une autre source de données en fonction d’une comparaison d’égalité entre des clés spécifiées dans chaque élément. Dans LINQ, les opérations de jointure sont effectuées sur les séquences des objets dont les éléments sont des types différents. Après avoir joint deux séquences, vous devez utiliser une instruction select ou group pour spécifier quel élément stocker dans la séquence de sortie. Vous pouvez également utiliser un type anonyme pour combiner des propriétés de chaque jeu d’éléments associés dans un nouveau type pour la séquence de sortie. L’exemple suivant associe des objets prod dont la propriété Category correspond à l’une des catégories dans le tableau de chaînes categories. Les produits dont la propriété Category ne correspond pas à une chaîne quelconque dans categories sont éliminés par filtrage. L’instruction select projette un nouveau type dont les propriétés sont extraites de cat et de prod.

Vous pouvez également effectuer une jointure groupée en stockant les résultats de l’opération join dans une variable temporaire à l’aide du mot clé into.



### Linq à travers les lambda expressions

Les lambda expressions sont des expressions anonymes facilitant l’écriture d’expressions. Ici, elles sont utilisées en complément des méthodes d’extensions.

#### Méthode d’extension

Les méthodes d'extension vous permettent d'« ajouter » des méthodes à des types existants sans créer un type dérivé, ni recompiler ou modifier le type d'origine. Les méthodes d'extension sont un type particulier de méthode statique appelées comme s'il s'agissait de méthodes d'instance sur le type étendu. Pour le développeur, il n’y a aucune différence apparente lors de l’appel entre une méthode d’extension et les méthodes qui sont réellement définies dans un type.

Les méthodes d’extension sont définies comme méthodes statiques mais sont appelées en utilisant la syntaxe de méthode d’instance. Leur premier paramètre spécifie les types sur lesquels la méthode s’applique et le paramètre est précédé du modificateur [this](https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/this). Les méthodes d'extension sont uniquement dans la portée lorsque vous importez explicitement l'espace de noms dans votre code source avec une directive using.

L'exemple suivant présente une méthode d'extension définie pour la classe [System.String](https://docs.microsoft.com/fr-fr/dotnet/api/system.string). Notez qu'elle est définie à l'intérieur d'une classe statique, non imbriquée et non générique :



#### Syntaxe de méthode

Linq vient avec son lot de méthode d’extensions. Parmi elles, nous pouvons citer [Sum](https://docs.microsoft.com/fr-fr/dotnet/api/system.linq.enumerable.sum), [Max](https://docs.microsoft.com/fr-fr/dotnet/api/system.linq.enumerable.max), [Min](https://docs.microsoft.com/fr-fr/dotnet/api/system.linq.enumerable.min), [Average](https://docs.microsoft.com/fr-fr/dotnet/api/system.linq.enumerable.average), Where…

Nous allons ici simplement détailler l’utilisation du Where, qui est l’opération la plus répandue.

Le mot clef Where permet de filtrer une collection selon un ou plusieurs critères.

Il s’utilise assez naturellement de la manière suivante :

