UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Base du Langage C#

Sommaire

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Découverte du C# et Installations

Le C Sharp ?

- Le **C#** (se prononce CSharp) se présente comme la suite des langages C et C++, une peu comme Java.
- C'est un langage complètement **orienté objet**.
- Permet l'écriture de programme **plus sûr et plus stable** :
  - Gestion automatique de la mémoire à l'aide du ramasse miette (garbage collector)
  - Gestion des exceptions

Nouveautés par rapport au C++

- Libération automatique des objets
- Disparition de l'obligation d'utiliser des pointeurs (remplacés par des références)
- Disparition du passage d'argument par adresse au profit du passage par référence
- Nouvelles manipulations des tableaux
- Nouvelle manière d'écrire des boucles (foreach)
- Disparition de l'héritage multiple mais possibilité d'implémenter plusieurs interfaces par une même classe

Visual Studio

Visual Studio est le **logiciel de développement** (IDE) créé par Microsoft principalement pour l'**environnement .NET** qui est actuellement dans sa version **2022**.

Il en existe plusieurs version selon les besoins, celle que nous utiliseront est la **Community** car elle est gratuite.
*A ne pas confondre avec **Visual Studio Code** qui est plus généraliste.*

Rider est une alternative payante à Visual Studio, il est développé par JetBrains et reprend les fonctionnalités de Visual Studio.

Visual Studio Installer

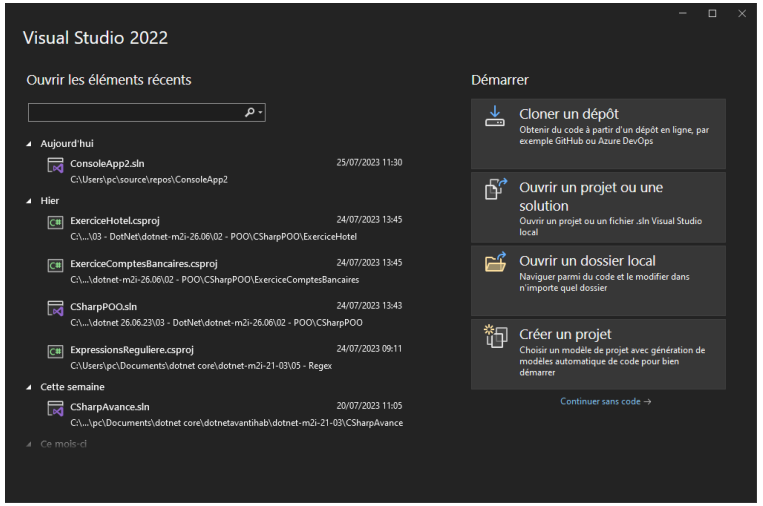
Pour l'installation des modules pour le développement avec visual studio, il faut utiliser **Visual Studio Installer**

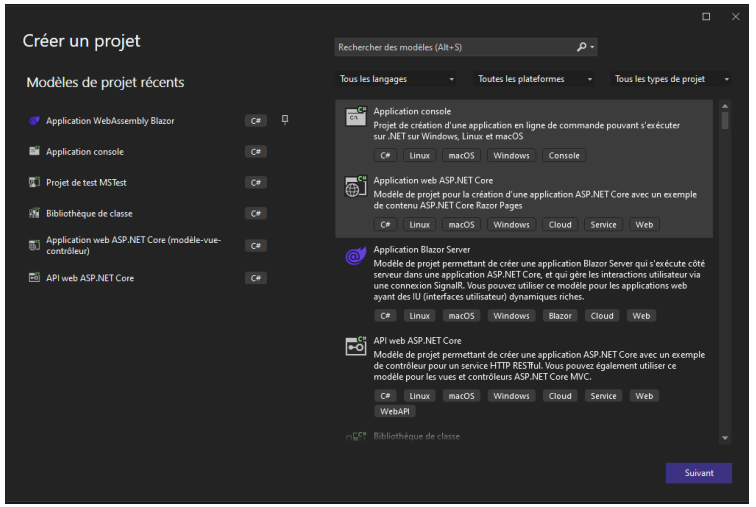
Il faudra s’assurer que les bon modules sont installés si l'on veut développer avec certaines technologies ou frameworks.

Interface de Visual Studio (Accueil et Création)

Pour la création de projets, l'IDE propose plusieurs squelettes :

- Console : Application sur terminal/shell
- Web / API : Applications utilisant le framework ASP.NET Core (MVC, Blazor, ...)
- Mobile : Applications pour IOS ou Android
- Test, Bibliothèque de classes, ...

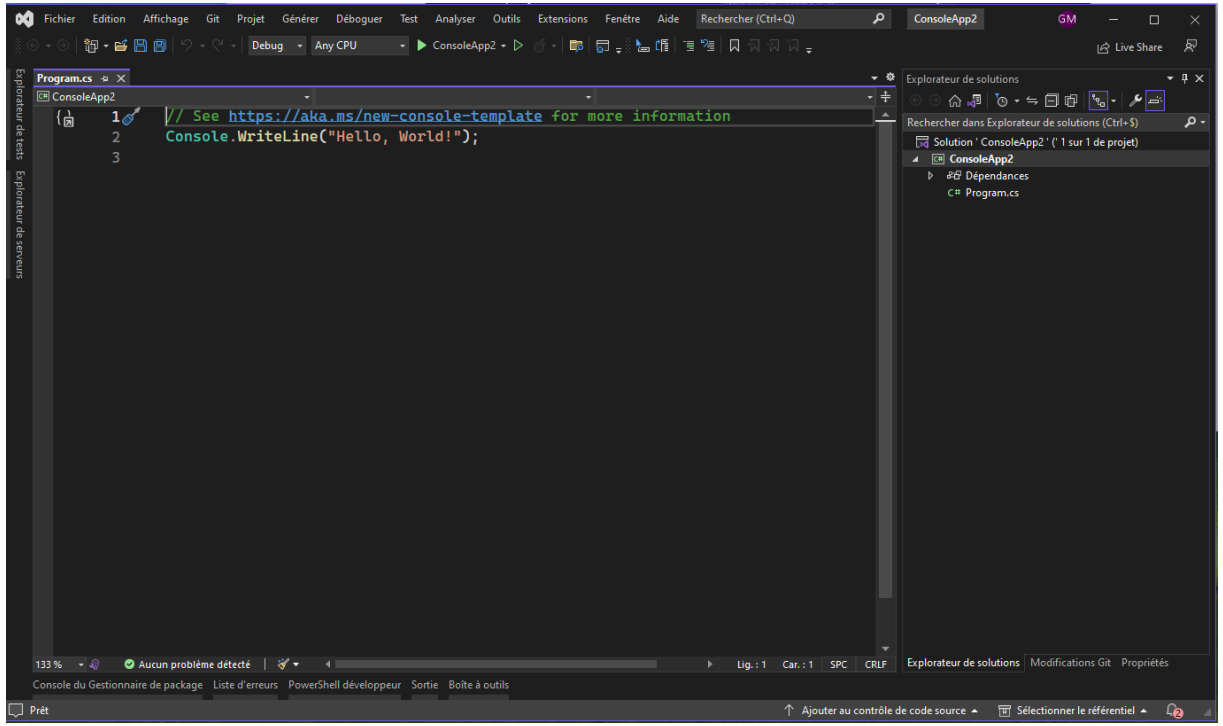




Interface de Visual Studio

L'interface Visual Studio est très personnalisable et comporte une grande quantité d'onglets et fenêtres différents.

Il est possible de les retrouver facilement (si fermé/déplacés) à l'aide de **la barre de recherche en haut à droite**.

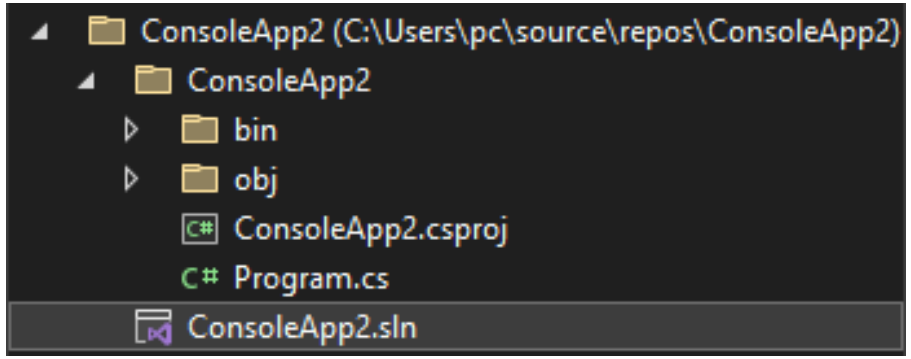


Premier projet C#

Il est temps de créer un premier projet console.

Par défaut, il sont créés dans le dossier `C:\<user>\pc\source\repos` mais il est possible de choisir un emplacement.

Voici la structure de fichiers créée pour une application Console.



Les dossiers bin et obj servent pour le fonctionnement du .NET (compilation, ...)

Solution et Projet

Parmi les fichiers générés, on retrouve le **fichier solution (.sln)** et le **fichier projet (.csproj)**.

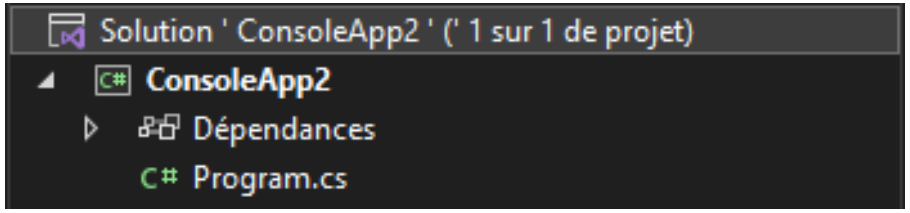
- Le **fichier projet** correspond à **un projet C#** (ex : application que l'on peut lancer).
  Après la **compilation** (**build**) du projet, on retrouve les fichiers **.dll** et **.exe** dans le dossier `<projet>\bin\Debug\net6.0`
- Le **fichier solution** est un **répertoire de projets** pouvant être ouvert directement dans visual studio.
  /!\ Attention : L'arborescence dans l'explorateur en mode solution ne correspond pas au dossiers et fichiers

Solution et Projet

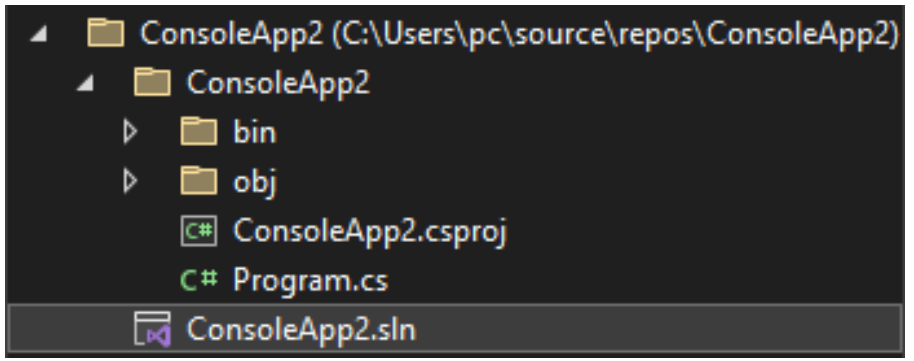
En effet il sera important de différencier les 2 modes de Visual Studio, beaucoup de fonctionnalités marchent différemment selon le mode.

On peut facilement changer entre les 2 modes avec ce bouton : 

Mode Solution (à privilégier)



Mode Fichiers

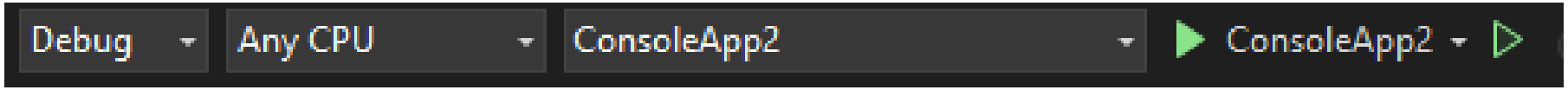


Lancement de l'application

Lorsque l'on clique sur le **triangle vert plein**, on lance l'application en **mode debug**, le **triangle vide** correspond au **mode sans débogage** (exécution comme en production).

La liste déroulante juste à gauche des boutons de lancement permet de **changer le projet de démarrage**.

Options de lancement

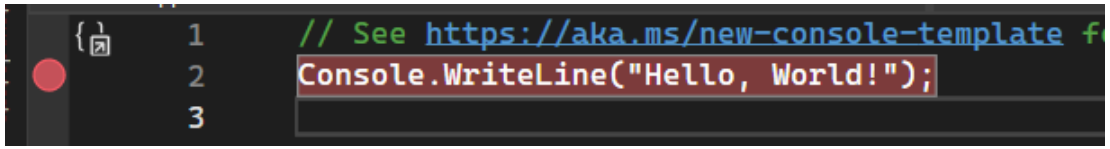


Debug

Le **mode debug** permet un gain d'efficacité en programmation :

- Arrêt et informations en cas d'**Exception** (Erreur lors du fonctionnement de l'application)
- Le **Tracker** pour suivre l'évolution des variables en mémoire
- Les **points d'arrêts** permettant d'arrêter le programme sur une instruction précise puis de naviguer dans l'exécution grace aux boutons de contrôle du debug

Un point d'arrêt



Boutons de contrôle du debug

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Syntaxe du C#

Point d'entrée d'une application

Un programme développé en C# doit respecter certaines conventions

- Il comporte obligatoirement une fonction Main, c'est le point d'entrée de notre application
- La fonction Main doit être obligatoirement membre d'une classe

Depuis le .NET 6, le fichier **program.cs**, point d'entrée de l'application, a été simplifié : nous y retrouvons désormais **uniquement le corps de la méthode Main**.

Appels à des fonction/méthodes

Il existe en programmation ce que l'on appelle des **fonctions et méthodes**, elles permettent la réutilisation d'un **ensemble d'instructions**. Pour utiliser une méthode appartenant à une classe, il faudra "**l'appeler**". En C# il existe 2 méthodes :

- Spécification du nom complet

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

- Spécification du nom relatif avec import du namespace (using)

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

/!\ Ne pas oublier les parenthèses !

Les trois type de commentaires en c#

- Le commentaire de ligne

```
// Le reste de la ligne est commenté
```

- Le commentaire multi-lignes

```
/*
Tout le texte situé entres les deux délimiteurs
*/
```

- Le commentaire servant à la documentation

```
/// Ceci est un commentaire de documentation
```

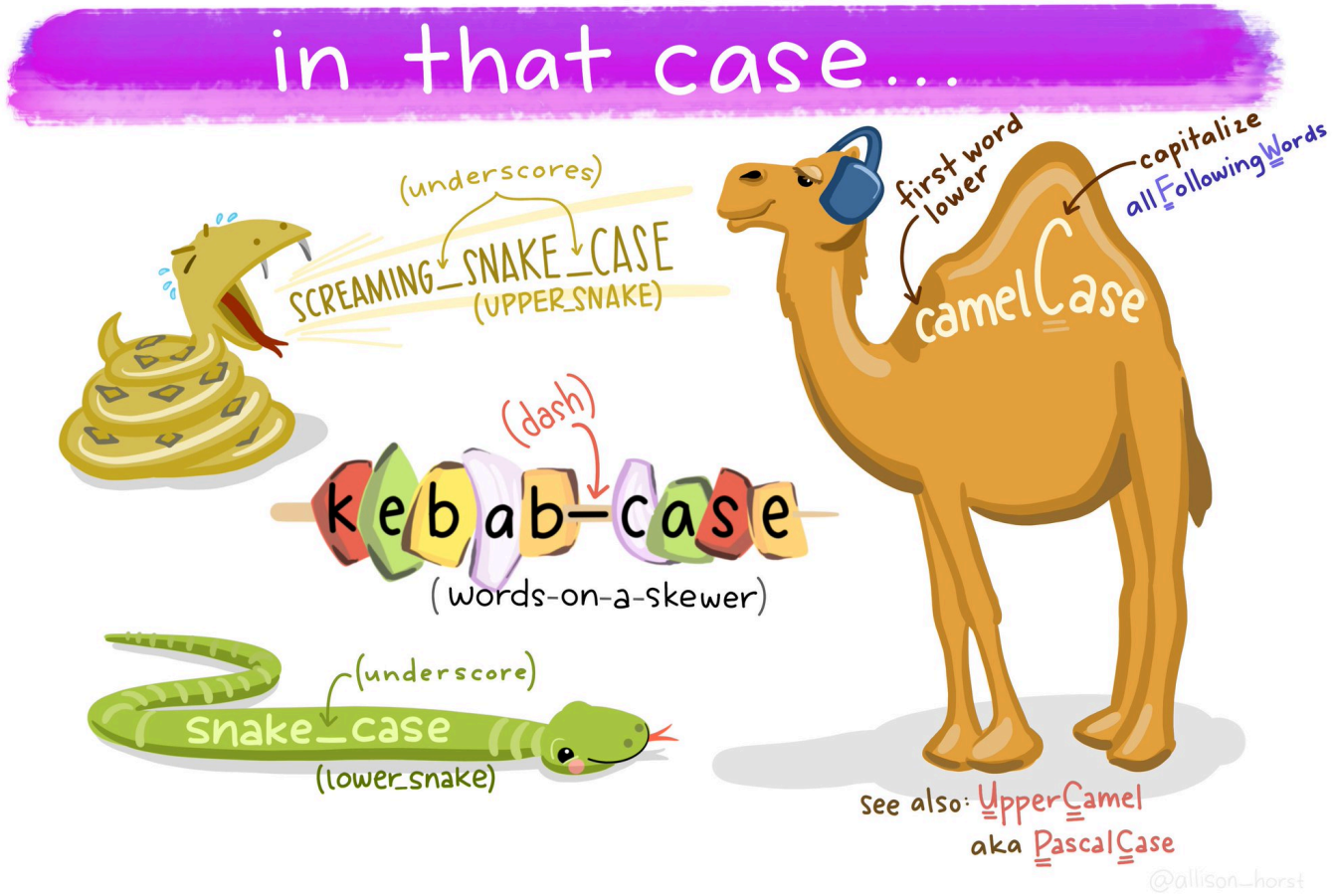
Les identificateurs

Les identificateurs sont des noms qu'on donne à des éléments de notre algorithme (variables, fonctions, classe, ...)
En C# ils doivent respecter certaines normes :

- **Premier** caractère : **lettre** ou le **underscore _**
- **Distinction** entre **minuscule** et **majuscule** (**Case Sensitive**)
- Les caractères accentués sont acceptés (mais pas recommandés)
- Un **mot réservé** du C# **peut être utilisé** comme identificateur de variable à condition qu'il soit **précédé** de **@**

Ex. d'identificateurs : `NbLignes`, `NbEcoles`, `@int`, `maVariable`

Normes de nommage



Normes en C# :

camelCase	PascalCase
variables	Fonctions
attributs	Propriétés
	Classes
	Méthodes

Les instructions en C#

Une instruction est une opération que l'on demande à l'ordinateur de réaliser. En C# elles se terminent obligatoirement par un **point virgule** « **;** ».

```
Console.WriteLine("Bonjour");
```

Une suite d'instructions délimitée par des **accolades {}** constitue un **bloc d'instruction**

- Les blocs définissent **les zones de validité des variables** (**portée** des variables), **en dehors du bloc** la variable **n'existe plus**
- On les retrouve dans l'utilisation des structures de contrôles, fonctions, méthodes, classes, ...

Les mots réservés du C# (keyword)

abstract
as
base
bool
break
byte
case
catch
char
checked
class
const
continue
decimal
default
delegate
do
double
else
enum
event
explicit
extern
false
finally
fixed
float
for
foreach
goto
if
implicit
in
int
interface
internal
is
lock
long
namespace
new
null
object
operator
out
override
params
private
protected
public
readonly
ref
return
sbyte
sealed
short
sizeof
stackalloc
static
string
struct
switch
this
throw
true
try
typeof
uint
ulong
unchecked
unsafe
ushort
using
virtual
void
volatile
while

Les keywords contextuels : (que dans des cas précis)

add
and
alias
ascending
args
async
await
by
descending
dynamic
equals
file
from
get
global
group
init
into
join
let
managed (function pointer calling convention)
nameof
nint
not
notnull
nuint
on
or
orderby
partial (type)
partial (method)
record
remove
required
scoped
select
set
unmanaged (function pointer calling convention)
unmanaged (generic type constraint)
value
var
when (filter condition)
where (generic type constraint)
where (query clause)
with
yield

[Documentation sur les keyword]

Les expressions en C#

Les **expression** sont un des éléments les plus importants de l'algorithmie ce sont les **composant des instructions**.

Comme en mathématiques sa réduction donne **une valeur**.

Une expression peut être composée de **plusieurs expressions**.

Exemples :

- `1` => l'entier 1
- `1 + 2` => addition de l'entier `1` avec l'entier `3` donc `3`
- `1 > 5` => comparaison de 2 entiers donnant `False` (faux)
- `"Bonjour"` => suite de caractères (chaine de caractères)

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Console

Affichage en mode console

L'affichage en mode console se fait essentiellement à l'aide des méthodes **Write** et **WriteLine**

```
Console.Write("chaîne");
//Affiche une chaîne de caractères
Console.WriteLine("chaîne");
//Affiche une chaîne de caractères puis retourne à la ligne ('\n')
```

Documentation

Saisie en mode console

La saisie en mode console se fait essentiellement à l'aide de la méthode **ReadLine**

```
Console.ReadLine();
// Lit une chaîne de caractères à partir du
// flux standard d'entrée jusqu'à l'appui sur la touche Entrée
```

- La valeur retournée est **null** si aucune donnée n'a été saisie (l'utilisateur tape directement ENTREE)
- `ReadLine()` ne peut retourner que des chaînes de caractères.
  Il faudra convertir le retour pour les autres types

Documentation

Les caractères spéciaux dans les chaînes de caractères (string)

Comment faire pour pouvoir afficher certains caractères spéciaux ?

- Utilisation du caractère spécial **backslash ** (échappement)
- Utilisation des **verbatim string** avec un **@**

```
Console.WriteLine("Je m'appelle \"Guillaume\"");
Console.WriteLine('\'');
Console.WriteLine("c:\\repertoire\\fichier.cs");
Console.WriteLine(@"c:\repertoire\fichier.cs");
```

Caractères spéciaux dans les chaînes

Séquence	Nom du caractère
\0	**Null** (caractère vide ou caractère spécial)
\a	Alerte
\b	Retour arrière
\f	Saut de page
\n	**Nouvelle ligne**
\r	Retour chariot
\t	**Tabulation horizontale**
\v	Tabulation verticale
\u	Séquence d'échappement Unicode (UTF-16)
\U	Séquence d'échappement Unicode (UTF-32)

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Variables et Types

Les variables

Les **variables** ont pour but de **stocker** des informations dans la mémoire vive de l'ordinateur. Les **variables** peuvent être de plusieurs **types**, qui sont parmi les plus fréquents :

- Les variables de type **numériques** servant à stocker des nombres. On y retrouve différents types pour les **entiers** et les **réels**
- Les **caractères** et **chaines de caractères** pour stocker du texte
- Les **booléens**, pour les valeurs binaires (Vraie=True / Fausse=False)
- Le **Vide**, **null** en C#, est une valeur à part qui ne représente **'Rien'**, il n'est ni un 0, ni un False, ni une chaine vide

Les différents types de variable

Type	Classe (BCL)	Description	Exemples
bool	System.Bool	**Booléen** (vrai ou faux : **true** ou **false**)	`true` `false`
sbyte	System.SByte	**Entier** signé sur 8 bits (1 octet)	`-128`
byte	System.Byte	**Entier** non signé sur 8 bits (1 octet)	`255`
short	System.Int16	**Entier** signé sur 16 bits	`-129`
ushort	System.UInt16	**Entier** non signé sur 16 bits	`1450`
int	System.Int32	**Entier** signé sur 32 bits	`-100000`
uint	System.UInt32	**Entier** non signé sur 32 bits	`8000000`
long	System.Int64	**Entier** signé sur 64 bits	`-2565018947302L`
ulong	System.UInt64	**Entier** non signé sur 64 bits	`8000000000000L`

Les différents types de variable

Type	Classe (BCL)	Description	Exemples
float	System.Single	**Réel** sur 32 bits	`3.14F`
double	System.Double	**Réel** sur 64 bits	`3.14159`
decimal	System.Decimal	**Réel** sur 128 bits	`3.1415926M`
char	System.Char	**Caractère** Unicode (16 bits)	`'A'` `'λ'` `'ω'`
string	System.String	**Chaîne de caractères** unicode	`"C:\\windows\\system32"`
Tuple	System.ValueTuple<>	**Regroupement** de données	`(4.5, "test")`
dynamic	NON TYPÉE	**Variable dynamique** (faiblement typée et "évaluée" à l'exécution)	`2` `"test"` `3.14` `true`

Les différents types de variable

Type	Classe (BCL)	Description	Exemples
enum	System.Enum	**Énumération** de possibilités	`enum Season {...}`
delegate	System.Action<> System.Func<>	**Fonction/Méthode anonyme**	`x => x*2`
struct		**Structure** de données (par valeur)	`struct Person {...}`
object	System.Object	Tous types d'**objets** instanciés	`new Person(...){...}`
class		**Classes** pour instancier des objets	`class Person {...}`
record		**Enregistrement** (classe simplifiée)	`record Person(...)`
interface		Définition d'un **contrat**	`interface MyContract {...}`

Nous reviendront sur ces types dans la partie C# Avancée (POO)

Type	Exemples
sbyte	-128 à 127
byte	0 à 255
short	-32 768 à 32 767
ushort	0 à 65 535
int	-2 147 483 648 à 2 147 483 647
uint	0 à 4 294 967 295
long	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807
ulong	0 à 18 446 744 073 709 551 615
float	±1,5 x 10−45 à ±3,4 x 1038 soit ~6-9 chiffres
double	±5,0 x 10−324 à ±1,7 x 10308 soit ~15-17 chiffres
decimal	±1,0 x 10-28 to ±7,9228 x 1028 soit 28-29 chiffres

Quelques précisions

Le mot clé **var** s'utilise à la place du type d'une variable, il **ne rend pas cette variable dynamique** mais **la type implicitement**, ainsi son type dépendra de **l'instruction d'affectation**

Le type **decimal** est utilisé pour les opérations financières

- Il permet une **très grande précision**
- Les opérations avec ce type plus lentes que les types **double** ou **float**

La déclaration d'une variable

Les variables sont déclarées par leur **type** suivi de leur **nom** (identificateur de la variable)

```
// <type> <nom variable>;
int age;
```

Les variables peuvent être **déclarées** et **affectées** en même temps (initialisation)

```
int age = 20;
```

La déclaration d'une variable

Toute variable doit être déclarée dans un bloc. **Elle cesse d'exister en dehors de ce bloc** (pas de variable globale), son espace mémoire sera libéré par le **Garbage Collector**

En l'absence d'une initialisation explicite les variables sont **implicitement initialisés** par le compilateur à la valeur par défaut : **numérique** à `0`, char à `'\0'` **chaîne** à `""` et **objet** à `null`
On peut utiliser l'affectation avec le mot clé `default` pour obtenir la valeur par défaut
/!\ Cependant le compilateur demandera toujours d'initialiser les variables

Nullable ?

Il est possible de passer le type d'une variable en **Nullable** en ajoutant un **point d'interrogation** après le type.
Cela aura pour effet d'ajouter la possibilité que la variable contienne `null`

```
// <type>? <nom variable>;
int? age;
age = null;
```

Les constantes symbolique

Valeur constante désignée par un **identificateur** dans le code source, qui est **remplacé par sa valeur** lors de la pré-compilation.

La déclaration des constantes se fait avec le mot-clé «**const**»

Une constante doit être obligatoirement initialisée

```
// const <type> <nom constante> = <valeur>;
const double Pi = 3.1415926535897932;
```

Documentation

Les catégories de variables

Les variables de type « **valeur** »

- Contient **directement la valeur**

Les variables de type « **référence** »

- Référence un objet (**référence mémoire vers l'objet**)

Comparaison de valeurs et comparaison de références

Une variable de type "**valeur**" possède **sa propre copie** des données qu'elle stocke. Deux variables de type valeur peuvent avoir les mêmes données mais chacune possède sa propre copie distincte (**instance**)

- **La modification de l'une des deux variables n'affecte pas l'autre**

Une variable de type "**référence**" contient une **référence à des données stockées dans un objet**. Deux variables de types références peuvent donc **référencer les mêmes données**

- **La modification de ces données à travers l'une de ces deux**

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Opérateurs

Les opérateurs arithmétiques et d'affectation

Les opérateurs arithmétiques

Opérateur	Fonction
+	Addition
-	Soustraction
/	Division
*	Multiplication
%	Modulo (reste de la division Euclidienne)

Les opérateurs d'affectation

Opérateur	Fonction
=	Affectation classique
++	Incrémente de 1*
--	Décrémente de 1*
+=	Addition (à une variable)
-=	Soustraction (à une variable)
/=	Division (à une variable)
*=	Multiplication (à une variable)

* si l'on met l'**opérateur avant** la variable l'incrémentation se fait **avant** : `int var2 = ++var;` est différent de `int var2 = var++;`

Les méthodes de la classe Math

Dans la BCL il existe une classe qui rassemble une grande partie des fonctions habituelles en mathématiques, la classe Math.
On devra les utiliser avec la syntaxe `Math.NomFonction(...)`
Quelques exemples :

- Pow Sqrt Cbrt : puissance et racines carré/cube
- Abs : valeur absolue
- Cos Sin Tan ... : fonctions de trigonométrie
- Round Celling Floor : arrondis
- Log Exp : Logarithme et Exponentielle

Les opérateurs d'égalité et de comparaison

- Ils se placent **entre 2 valeurs** et doivent **avoir du sens** pour ces deux variables
- Leur retour est un **booléen**

Opérateur	Description	Opérateur	Description
==	Égalité	!=	Inégalité
>	Supérieur à	>=	Supérieur ou égal
<	Inférieur à	<=	Inférieur ou égal

Exemples : `1 == 0` `true != false` `4 < 4` `6 >= 5`

Les opérateurs logiques (booléens)

- Ils se placent **entre 2 valeurs** de type **bool**

Opérateur	Description	Opérateur	Description
&	**ET** / AND	&&	**ET** "court-circuit"
|	**OU** / OR	||	**OU** "court-circuit"
^	OU exclusif / **XOR**	!	**NON** / Négation / NOT

- Les version "**court circuit**" sont à **privilégier** car elles simplifient et optimisent l'expression au niveau de l'exécution
- Priorité des opérateurs : **!→&→^→|→&&→||**

Les chaînes de caractères (string)

Lors d'une déclaration d'une chaîne de caractère (**string**), on déclare une référence à une sorte de **tableau de caractères** (**char**)
On peut donc interroger ce « tableau » à un numéro de cellule avec l'**opérateur []**

```
string prenom = "Guillaume";
Console.WriteLine(prenom[0]); // Affichera « G »
```

- **prenom** est une référence mémoire du « tableau » de type **char** `[G,u,i,l,l,a,u,m,e]`
  Effectivement à l'index `[0]` on a `'G'`

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Cast de type

Le casting d'une variable consiste à la **convertir le type d'une variable en au autre type**

Nous pouvons rencontrer deux possibilités lors d'un cast de type :

- Soit les deux types sont compatibles
  - Exemple: Caster un `short` en `int`
- Soit les deux types sont incompatibles
  - Exemple: Caster un `string` en `int`

Le casting entre deux types compatibles

C'est le cas le plus simple de conversion de type

- Le casting **implicite** (Le compilateur le fait pour nous)
  - Caster un **short** en **int** se fait implicitement

```
short @short = 200;
int @int = @short;
Console.WriteLine(@int); // Affichera 200
```

  - En effet rentrer un petit type dans un grand se fait sans efforts
  - Par contre, la réciproque n'est pas vraie

Le casting entre deux types compatibles

Caster un type plus grand dans un plus petit

- Il faudra employer un casting **explicite**
  - Caster un **int** dans un **short**

```
int @int = 200;
short @short = (short) @int;
Console.WriteLine(@short); // Affichera 200
```

  - Attention, un casting explicite indique au compilateur que vous savez parfaitement ce que vous faites
  - Un grand pouvoir implique de grandes responsabilités...

Le casting entre deux types compatibles

Caster un type plus grand dans un plus petit

- L'erreur est humaine et parfois...
  - Un casting **explicite** mal contrôlé et c'est le bug assuré

```
int @int = 200000;
short @short = (short) @int;
Console.WriteLine(@short); // Affichera 3392
```

  - Allez expliquer à votre client que les 196608 € manquant sont liés à une erreur de cast... Si vous la trouvez !
  - Un grand pouvoir implique de grandes responsabilités...

L'opérateurs de test de type et de cast is

L'opérateur `is` permet de vérifier si une **valeur** est **compatible** avec un **type**. Son résultat est un **booléen**.

```
bool compatible = 21 is int;
bool incompatible = 21 is string;
```

De plus il est possible avec cet opérateur de réaliser un **cast si c'est compatible**, pour ça on **ajoute un identificateur à la fin** ce qui pourra instancier une nouvelle variable au passage.

```
bool compatible = 21 is int varInt;
bool incompatible = 21 is string varString;
```

L'opérateurs de cast as

L'opérateur `as` est similaire à l'opérateur `is` mais il ne permet pas de renvoyer un booléen. Il cast directement dans un type si c'est compatible et renvoie null en cas d'incompatibilité.

```
int? varInt = 21 as int ;
string? varString = 21 as string ;
```

Ces deux opérateurs sont très utiles en Programmation Orientée Objet, notamment dans le principe de l'héritage.

Le casting entre deux types incompatibles

Incompatible peut-être, mais qui ont **le même sens** (sémantique)

Conversion d'une chaîne de caractère **string** en **int** (Parsing)

- Avec les méthodes de classe Convert, c'est possible...

```
string chaineAge = "20";
int age = Convert.ToInt32(chaineAge);
Console.WriteLine("chaineAge convertie en int : " + age);
```

- En effet `"20"` en **string** et en **int** se convertissent facilement
- Un équivalent à ces méthodes est la méthode `<type>.Parse(...)`.
- Il existe aussi la méthode `<type>.TryParse(..., out var)` mais nous y reviendrons plus tard

Le casting entre deux types incompatibles

Conversion d'une chaîne de caractère **string** en **int**

- Possible… mais quand la chaîne ne représente pas un entier, la conversion va échouer

```
string chaineAge = "vingt ans" ;
int age = Convert.ToInt32(chaineAge);
Console.WriteLine("chaineAge convertie en int :" + age);
```

  - En effet, l'exemple ci-dessus va vous faire une erreur

```
System.FormatException : 'Input string was not in a correct format.'
```

  - Le programme se retrouvera bloqué…

Résumé sur le casting entre deux types

- Il est possible, avec le casting, de **convertir la valeur d'un type vers un autre** lorsqu'ils sont **compatibles** entre eux
- Le **casting explicite** s'utilise en préfixant une variable par **un type précisé entre parenthèses**
- Le Framework .NET possède des méthodes permettant de **convertir des types incompatibles** entre eux s'ils sont **sémantiquement correspondants**
- Il existe d'autre méthodes de conversion pertinentes (cf documentation)

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Structures conditionnelles

Les structures conditionnelles

Utiles pour des opérations qui **dépendent d'un résultat précédent**

- Lors d'une opération de connexion par exemple
  - **Si** le login et le mot de passe sont **bons**, nous pouvons nous connecter
  - **Sinon** un message d'erreur s'affiche à l'écran

Il s'agit de ce que l'on appelle une **condition**

- Elle est évaluée lors de l'exécution et en fonction de son résultat (**vrai** ou **faux**) nous ferons telle ou telle chose
- Une condition peut se construire grâce à des opérateurs de **comparaison** et **logiques**

L'instruction « if »

- Elle permet d'exécuter du code **si** une condition est **vraie** (**if** = **si**)

```
decimal compteEnBanque = 300;
if (compteEnBanque >= 0)
    Console.WriteLine("Votre compte est créditeur") ;
```

- Pour plusieurs instructions les accolades son obligatoire :

```
decimal compteEnBanque = 300;
if (compteEnBanque >= 0)
{
    Console.WriteLine("Votre compte est créditeur") ;
    Console.WriteLine("Solde restant : " + compteEnBanque + " Euros");
}
```

L'instruction « if »

- Il est possible de vérifier plusieurs conditions à la suite à l'aide de plusieurs if (non corrélés)

```
decimal compteEnBanque = 300;
if (compteEnBanque >= 0)
    Console.WriteLine("Votre compte est créditeur") ;
if (compteEnBanque < 0)
    Console.WriteLine("Votre compte est débiteur") ;
```

- Une autre solution est d'utiliser le mot clé « **else** », qui veut dire « **sinon** » en anglais

L'instruction « else » (sinon)

- Elle sera toujours **à la suite d'une instruction « if »**
- **Si** la valeur est **vraie**, alors on fait quelque chose, **sinon**, on fait autre chose

```
decimal compteEnBanque = 300;
if (compteEnBanque >= 0)
    Console.WriteLine("Votre compte est créditeur") ;
else
    Console.WriteLine("Votre compte est débiteur") ;
```

Les structures conditionnelles

L'expression entre parenthèse peut être remplacée par un **booléen**

- Un type **bool** peut prendre deux valeurs, vrai ou faux, qui s'écrivent avec les mots clés `true` et `false`

```
bool estVrai = true;
if (estVrai)
    Console.WriteLine("C'est vrai !");
else
    Console.WriteLine("C'est faux !");
```

TryParse

Convertir avec **Convert** fait appel à la méthode **type.Parse()**

- Quand le `int.Parse()` échoue, il provoque une erreur
- La méthode `int.TryParse()` nous informe si la conversion s'est bien passée ou non, sans faire d'erreur

```
string chaineAge = "ABC20";
int age;
if (int.TryParse(chaineAge, out age))
    Console.WriteLine("La conversion est possible, age vaut " + age);
else
    Console.WriteLine("Conversion impossible") ;
```

L'instruction « else if » (sinon si)

- Elle sera toujours **à la suite d'une instruction « if » ou « else if »** et **avant le « else » final**
- Si la ou les conditions précédentes sont **fausses**, alors on vérifie **une autre condition**. On peut en **cumuler autant que nécessaire**.

```
if (compteEnBanque > 0)
    Console.WriteLine("votre compte est créditeur");
else if (compteEnBanque == 0)
    Console.WriteLine("Votre solde est nul") ;
else
    Console.WriteLine("Votre compte est débiteur") ;
```

Opérateurs logiques dans la condition

- Il est également possible de combiner les tests grâce aux **opérateurs logiques**
- Par exemple **&&** (ET)

```
string login = "Jeanne";
string motDePasse = "essai ";
if (login == "Jeanne" && motDePasse == "essai")
    Console.WriteLine("Bienvenue Jeanne") ;
else
    Console.WriteLine("Login incorrect") ;
```

Les ternaires ?:

Il est aussi possible d'utiliser ce qu'on appelle les **ternaires**, il s'agit d'une **expression** comportant une **condition**.
On peut le comparer à une **structure conditionnelle if**.

Il se structure comme suit :

```
<condition> ? <valeur si vrai> : <valeur si faux>
```

Exemple

```
int age = 14;
bool majeur = age >= 18;
string majeurStr = majeur ? "Il est majeur" : "Il n'est pas majeur";
```

Null coalescing operator ??

Il existe un opérateur semblable au ternaire servant à vérifier **si une valeur est nulle** et de lui **affecter une valeur** si c'est le cas.
Il s'agit de l'opérateur de coalescence nulle `??`.
Il est possible de le combiner avec l'opérateur d"affectation `??=`

Il se structure comme suit :

`<valeur> ?? <valeur si null>`

Exemple

```
int? age = null;
int ageNonNull = age ?? 18;
```

L'instruction « switch .. case »

- L'instruction **switch** peut être utilisée lorsqu'**une seule variable** peut prendre beaucoup de **valeurs différentes**
- Elle permet de simplifier l'écriture, la variable correspondra à différent **cas** (**case**) qui auront chacun leur bloc d'instructions.
- Chaque bloc doit contenir un mot clé **break** pour sortir du switch
- Il est possible d'aller plus loin dans l'expression des cas à l'aide des **opérateurs de comparaison** (ex: `case > 10:`) et des mot clés **or**, **and** et **when** (ex: `case > 10 or < 2 when true:`).

L'instruction « switch .. case »

```
string civilite = "M." ;
switch (civilite)
{
    case "M." :
        Console.WriteLine("Bonjour monsieur");
        break;
    case "Mme"
        Console.WriteLine("Bonjour madame");
        break;
    case "Mlle" :
        Console.WriteLine("Bonjour mademoiselle");
        break;
}
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Structures itératives

Les boucles conditionnelles

C'est le type de boucle qui va nous permettre d'**executer un bloc d'instruction** tant qu'une **condition est vérifiée**.
Il en existe 2 types :

- Avec la vérification de la condition **avant** d'exécuter la boucle
  « **while** » => « **tant que** »
- Avec la vérification de la condition **après** avoir exécuté **une première fois** la boucle
  « **do ... while** » => « **faire ... tant que** »

La boucle while

```
int compteur = 1;
while (compteur <= 50)
{
    Console.WriteLine("Le compteur affiche : " + compteur);
    compteur++;
}
```

- Dans cet exemple, la boucle effectuera 50 itérations
- Attention aux boucles infinies

La boucle do ... while

```
int compteur = 1;
do
{
    Console.WriteLine("Le compteur affiche : " + compteur) ;
    compteur++;
} while (compteur <= 50) ;
```

- Le résultat semble identique à la boucle « while » pourtant la boucle s'est exécutée une première fois avant la vérification de la condition
- Dans ce cas essayons avec un exemple où la condition est fausse…

La boucle do ... while

```
int compteur = 51;
do
{
    Console.WriteLine("Le compteur affiche : " + compteur) ;
    compteur++;
} while (compteur <= 50) ;
```

- La console affiche: « Le compteur affiche : 51 » et le programme se termine après avoir exécuté une première fois la boucle
- Notez que dans notre exemple après la sortie de la boucle « do ... while » la variable compteur vaut 52

Les boucles d'itération

C'est le type de boucle qui va nous permettre d'**executer un bloc d'instruction** un certain nombre de fois que l'on **connaîtra** en général **à l'avance**.

Il en existe 2 types :

- Avec une **variable d'itération associée**
  « **for** » => « **pour** »
- Avec itération sur les valeurs d'un **ensemble**
  « **foreach** » => « **pour chaque** »

La boucle for (Pour)

- Elle permet de répéter des instructions tant qu'**une condition est vraie** et avec l'utilisation d'une **variable d'itération**

```
for (int compteur = 1 ; compteur <= 50; compteur++)
    Console.WriteLine("L'instruction a été exécutée " + compteur + " fois") ;
```

- L'instruction est exécutée tant que la **condition** « compteur <= 50 » est **vraie**
- La variable compteur est incrémentée de 1 à chaque boucle (compteur++)

Attention aux boucles infinies

La boucle for

- Elle peut être utilisée pour itérer sur contenu d'un tableau et afficher son contenu
  - Dans l'exemple ci-dessous **la variable d'itération** « i » est directement déclarée dans la boucle for
  - Le nombre de fois ou la boucle est exécutée est conditionné par la longueur du tableau lui-même

```
string[] joursSem = new string[] { "Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi",
"Samedi", "Dimanche"};
for (int i = 0; i < joursSem.Length; i++)
    Console.WriteLine(joursSem[i]);
```

La boucle foreach

- C'est une structure spécialement conçue pour **parcourir** des chaines, listes et tableaux (plus généralement des énumérables)
  - Pour cela le nombre d'itération est implicite et s'adapte automatiquement à la longueur du tableau ou de la liste
  - La boucle utilisera une variable pour stocker les différents éléments

```
string alphabet = "abcdefghijklmnopqrstuvwxyz";
foreach (char lettre in joursSem)
    Console.WriteLine(lettre);
```

La boucle foreach

- Attention, la boucle « **foreach** » est une boucle en **lecture seule**.
- Cela veut dire qu'il n'est pas possible de modifier l'élément de l'itération en cours, la tentative de modification de la variable du foreach provoquera une erreur :

```
Cannot assign to 'lettre' because it is a 'foreach iteration variable'
```

Modifier le contenu d'une liste ou d'un tableau

- Il faudra passer par une boucle **for**
  - L'exemple ci-dessous permet de modifier le contenu de la liste en assignant à chaque cellule la chaine « *i* - pas de jour ! »

```
List<string> jourSem = new List<string> { "Lundi","Mardi", "Mercredi", "Jeudi",
"Vendredi", "Samedi", "Dimanche"};
for (int i = 0; i < jourSem.Length; i++)
    jourSem[i] = i + " - pas de jour ! ";
```

L'instruction break

Il est possible de **sortir prématurément** de **tout type de boucle** grâce à l'instruction **break**

- Dès qu'elle est rencontrée, on **"casse la boucle"** et on en **sort**. L'exécution du programme continue alors avec les instructions situées **après la boucle**

```
foreach (string jour in jours)
{
    if (jour == "Jeudi")
        break;
    Console.WriteLine(jour);
}
```

L'instruction continue

Il est également possible de **passer à l'itération suivante** de **tout type de boucle** grâce à l'instruction **continue**

- Dès qu'elle est rencontrée, elle passe à l'**itération suivante** sans exécuter le reste des instructions de l'itération en court

```
foreach (string jour in jours)
{
    if (jour == "Jeudi")
        continue;
    Console.WriteLine(jour);
}
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Tableaux

Déclaration d'un tableau et allocation de mémoire

Les tableaux sont des types « **référence** »

Déclaration d'un tableau

```
Type[] nomTab; // NomTab fait référence à un tableau
```

Allocation de l'espace mémoire d'un tableau

```
nomTab = new Type[Taille] ; // Taille indique le Nb éléments
```

Exemple :

```
string[] prenoms; // Déclaration d'un tableau de string
prenoms = new string[3] ; // Le tableau contient 3 éléments
```

Nb : récupérer la taille du tableau se fait avec nomTab.Length

Déclaration, allocation et initialisation de valeur

Il est possible de **déclarer** et **allouer l'espace** en même temps

```
// Type[] NomTab = new Type[Taille];
string[] prenoms = new string[3];
```

Une fois déclaré nous pouvons initialiser ses valeurs (insertion)

```
prenoms[0] = "Titi"; // comme pour les strings on commence à l'index 0
prenoms[1] = "Tata";
prenoms[2] = "Toto"; // dernier index à Length-1
```

Il est aussi possible de **déclarer**, **allouer** et **initialiser** en même temps

```
float[] valeurs = new float[] {2.5f,0.3f,5.9f};
// ou
float[] valeurs = {2.5f,0.3f,5.9f};
```

Tableaux avec des cellules de types différents

Il est possible de créer des tableaux ayant des cellules de **types différents**

- Le type de base de ces tableaux doit être le type **object**

Une cellule de type **object** peut recevoir une valeur de n'importe quel type

```
object[] tab = new object[3];
tab[0] = 12 ;
tab[1] = 1.2 ;
tab[2] = "Message" ;
```

La libération mémoire d'un tableau

La **libération mémoire** d'un tableau se fait **automatiquement** par le ramasse-miettes (Garbage Collector)
Un tableau cesse d'exister :

- Lorsqu'on **quitte le bloc** dans lequel il est déclaré
- Lorsqu'on **assigne une nouvelle valeur** (y compris null) à la variable référence qui désigne le tableau

```
float[] reels = {2.5f,0.3f,5.9f};
reels = new float[] {3.9f,1.256f,425.68f};
reels = null;
```

La copie d'un tableau

La copie d'un tableau est en fait une copie de sa référence

```
int[] t1 = {2,3,4};
int[] t2; // t2 contient la valeur « null »
t2 = t1; // t1 et t2 font référence au même tableau
```

Ainsi, si vous modifiez la première valeur de t1

```
t1[0] = 5;
```

Alors la valeur de t2 sera **{5,3,4}**

Nos 2 variables permettent en fait d'**accéder au même contenu**

La copie d'un tableau

Un autre exemple avec deux tableaux de **taille différente**

```
- int[] t1 = {2,3,4};
- int[] t2 = new int[100];
- t2 = t1; // t1 et t2 font référence au même tableau
```

t2 fait maintenant référence à **la zone mémoire contenant le tableau de trois éléments**

- La zone mémoire contenant les 100 cellules est signalée "à libérer"
- Le ramasse-miettes (Garbage Collector) la libérera lorsqu'un besoin en mémoire se manifestera

La copie d'un tableau

Pour faire **réellement une copie** de tableaux il existe deux méthodes

- La méthode CopyTo
- La méthode Clone

La copie d'un tableau avec la méthode CopyTo()

Utilisation de la méthode **CopyTo()**

```
int[] t1 = {2,3,4};
int[] t2 = new int[10]; // Toutes les valeurs de t2 sont à 0 par défaut
t1.CopyTo(t2, 0); // Fais la copie à partir de l'index 0
```

Maintenant nous avons bien deux tableaux distinct

```
t1 = {2,3,4};
t2 = {2,3,4,0,0,0,0,0,0,0};
```

La copie d'un tableau avec la méthode Clone()

Utilisation de la méthode **Clone()**

```
int[] t1 = {2,3,4};
int[] t2; // t2 = null
t2 = (int[]) t1.Clone(); // Fais la copie de t1 dans t2
t1[0] = 100;
```

De nouveau, nous avons bien deux tableaux distincts (2 instances)

```
t1 = {100,3,4};
t2 = {2,3,4};
```

Les listes (List)

Il existe en C# une classe nommée **List** dont l'utilisation est similaire au tableaux mais plus simple.

En effet les Liste sont des conteneurs **dynamiques** pour plusieurs valeurs, la notion de **taille** pouvant être bloquante pour les tableaux **disparaît**.

Count permet de connaître le **nombre d'élément** actuel dans la liste (pas de Length)

Add, **Remove**, **RemoveAt** et **Clear** permettent d'**ajouter** et **supprimer** des éléments

En réalité on dit qu'une liste est une classe générique (collection) mais cela à rapport à la POO, nous y reviendrons plus tard

Exemple

```
List<string> mesChaines = new List<string>()
{
    "chaine1", "chaine2", "chaine3", "chaine4"
};

mesChaines.Remove("chaine2");
mesChaines.RemoveAt(0);
mesChaines.Add("chaine5");

foreach (string item in mesChaines)
{
    Console.WriteLine(item);
}
mesChaines.Clear();
```

Les Tuples

Le tuple permet de **regrouper** des données, on appelle ça du **packing**.
Pour le définir, on mettra plusieurs types entre parenthèse :

```
(double, int, string) t1 = (4.5, 3, "test")
```

Contrairement aux listes et tableaux les données sont **non-modifiables** et identifiées par leurs **noms** (par défaut **ItemX**)
On peut assigner les valeurs d'un tuple à plusieurs variables, on appelle ça l'**unpacking**.
Il est possible de renommer les éléments du tuple avec cette syntaxe (PascalCase)

```
(double MonDouble, int MonInt, string MonString) t2 = t1;
```

Exemple de Tuple

```
// affectation (packing)
(double, int, string) t10 = (4.5, 3, "test");
// autre notation :
ValueTuple<double, int, string> t1 = (4.5, 3, "test");

Console.WriteLine($"Tuple with elements {t1.Item1} and {t1.Item2}.");
Console.WriteLine(t1.Item3);

// unpacking
(double monDouble, int monInt, string monString) = t1;
var (monDouble2, monInt2, monString2) = t1;

// renommage
(double MonDouble, int MonInt, string MonString) t2 = t1;
Console.WriteLine($"Tuple with elements {t2.MonDouble} and {t2.MonInt}.");
Console.WriteLine(t2.Item3);
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Fonctions

Les fonctions

Une **fonction** regroupe un ensemble d'**instructions**, elle peut prendre des **paramètres** en entrée et **retourner une valeur**

- On parle parfois de « **méthode** » à la place du mot « **fonction** », c'est un concept différent qui intervient dans la **Programmation Orientée Objet**

Le but d'une fonction est de **factoriser** du code afin d'**éviter d'avoir à le répéter**. Ce souci de factorisation est connu comme le principe « **DRY** » (Dont Repeat Yourself)

Les fonctions

Il est possible en C# de créer des **fonctions locales** dans le fichier **program.cs**.

Leur déclaration se structure comme suit :

```
<type de retour> <nom fonction>(<paramètres>) { <instructions> }
```

Pour appeler la fonction et ainsi exécuter son bloc de code, il faudra utiliser cette syntaxe :

```
<nom fonction>(<arguments>)
```

Après l'exécution de la fonction, il faut imaginer que le **résultat** (retour) de la fonction va **remplacer la syntaxe ci-dessus**

*Contrairement aux **méthodes** que nous verront dans la partie avancée, elles n'ont pas de modificateurs d'accès public/private/...*

Exemple de fonction

Si l'on avait à faire un affichage récurent on pourrait utiliser une fonction comme celle-ci :

```
void AffichageBienvenue()
{
  Console.WriteLine("Bonjour à toi !") ;
  Console.WriteLine("--------");
  Console.WriteLine("\tBienvenue dans le monde merveilleux du C#");
  console.WriteLine("--------");
}
```

Pour appeler la fonction :

```
AffichageBienvenue();
```

Exemple de fonction

- Commençons par la première ligne

```
void AffichageBienvenue()
```

- C'est ce qu'on appelle la **signature** de la fonction. Elle nous renseigne sur le **nom** et les **paramètres** de la fonction ainsi que son **type de retour**
- Le mot clé **void** signifie que la fonction ne retourne **rien** (**type de retour**)
- Les **parenthèses vides** à la fin de la signature indiquent que la fonction n'a **pas de paramètres**

Exemple de fonction

La partie entre **{}** correspond au **bloc d'instructions** qui sera exécuté à **chaque appel de la fonction**.

```
{
  Console.WriteLine("Bonjour à toi !") ;
  Console.WriteLine("--------");
  Console.WriteLine("\tBienvenue dans le monde merveilleux du C#");
  console.WriteLine("--------");
}
```

Les paramètres d'une fonction

- Il permettent d'augmenter les possibilités de réemploie d'une fonction et de la rendre adaptable
- Il se situent **entre les parenthèses** après le nom de fonction
  - Dans l'exemple ci-dessous les paramètres prénom et langage permettent de personnaliser le message de bienvenue

```
void AffichageBienvenue(string prenom, string langage)
{
  Console.WriteLine("Bonjour " + prenom + " !") ;
  Console.WriteLine("--------");
  Console.WriteLine("\tBienvenue dans le monde merveilleux du " + language);
  console.WriteLine("--------");
}
```

Les paramètres d'une fonction

- Nous pouvons désormais **appeler** cette fonction et passer plusieurs **valeurs** en **arguments**

```
AffichageBienvenue("Guillaume", "C#");
AffichageBienvenue("Antoine", "Javascript");
```

- L'appel de la fonction respecte bien la **signature** : il y a autant d'**arguments** à l'appel que de **paramètres** dans la définition.
- Nous lui passons bien deux chaînes de caractères en **arguments** : le prénom et le langage.

Le retour d'une fonction

- Une fonction peut **renvoyer une valeur**, elle effectue un « **return** » (retour en Anglais).
  Le type du return **doit correspondre au type de retour**

```
double Additionner(double nombreUn, double nombreDeux)
{
double sommeDesNombres = nombreUn + nombreDeux;
return sommeDesNombres; // équivalent : return nombreUn + nombreDeux;
}
```

- Lors de l'appel de la fonction, cette valeur de retour « **remplacera** » la fonction **après son appel**.

```
var resultat = Additionner(2, 4) * 4; // resultat = 6 * 4 = 24
```

Paramètres facultatifs ou par défaut

Il est aussi possible de passer un ou plusieurs argument(s) de manière **facultative** et de leur attribuer une valeur **par défaut**. Pour cela on utilise le =.

```
double Additionner(double nombreUn = 1, double nombreDeux = 2)
{
    double sommeDesNombres = nombreUn + nombreDeux;
    return sommeDesNombres; // équivalent : return nombreUn + nombreDeux;
}
```

- Lors de l'appel de la fonction, on pourra ne pas préciser ces arguments (attention à l'ordre)

```
var resultat = Additionner(2, 4) * Additionner(); // resultat = 6 * 3 = 18
```

Les paramètres par référence

- **Par défaut**, on dit que les paramètres sont passés par **valeur**.
  Une valeur peut être **modifiée dans la fonction**, mais **la valeur de la variable** en dehors de l'appel de la fonction **restera inchangée**.

- Les paramètres déclarés pour une fonction **avec** les mots clés **in**, **ref** ou **out** sont passés à la fonction appelée par **référence**.
  Lors du passage par **référence**, on pourra **"lier" des variables à une fonction** et **les modifier pendant son appel**.

Ce comportement est similaire à la copie de tableaux vu plus tôt

Liste des mots clés pour les paramètres de méthode

- **in** spécifie que ce paramètre est passé par **référence**, mais qu'il est en **lecture seule** par la méthode appelée et donc **non modifiable**
- **out** spécifie que ce paramètre est passé par **référence** et qu'il est **écrit** par la méthode appelée
- **ref** spécifie que ce paramètre est passé par **référence** et qu'il peut être **lu et écrit** par la méthode appelée
- **params** spécifie que ce **paramètre** de type **tableau** prendra **tout les arguments supplémentaire de la fonction**. On pourra ainsi faire des fonction avec **un nombre variable d'arguments**.

Exemple params

```
string Concataineur(int numero, params string[] chaines)
{
    string chaineFinale = numero + " - ";
    foreach (var chaine in chaines)
    {
        chaineFinale += chaine;
    }
    return chaineFinale;
}
string[] mesChaines = { "Bonjour", "Tout", "Le Monde" };

Console.WriteLine(Concataineur(1, mesChaines));
Console.WriteLine(Concataineur(2, "Bonjour", "Tout", "Le Monde"));
```

Les fonctions en résumé

- Une fonction **regroupe un ensemble d'instructions** pouvant prendre **des paramètres** et pouvant **renvoyer une valeur**
- Lors de l'appel d'une fonction, un lui passera des valeurs en **arguments** qui seront ensuite transmit aux **paramètres**. Ces arguments doivent avoir **le bon type** correspondant à leur paramètre respectif
- Une fonction qui **ne renvoie rien** est préfixée du mot-clé **void**
- Le mot-clé **return** permet de **renvoyer une valeur** correspondant au **type de retour** de la fonction. **Il met un terme à l'exécution de la fonction.**

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Merci pour votre attention

Des questions ?