



Concepteur Développeur en Informatique

Développer des composants d'interface

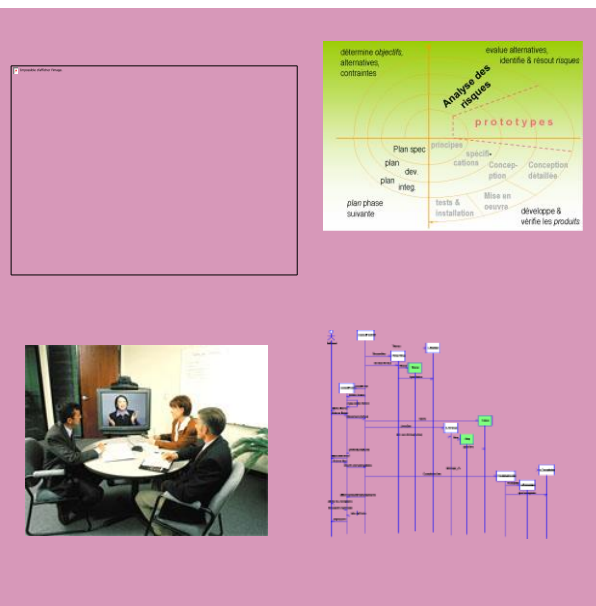
Initiation au langage C

Accueil

Apprentissage

PAE

Evaluation



Localisation : U03-E03-S01

SOMMAIRE

1. Introduction.....	5
2. Langage C# : Structure d'un programme.....	5
2.1. Un peu de vocabulaire	5
2.2. Les mots clés.....	5
2.3. Les espaces de noms	5
2.4. La directive Using.....	6
2.5. Le programme principal.....	7
2.6. Les variables.....	8
2.7. Les constantes	9
2.8. Les instructions.....	9
2.9. Les commentaires	9
2.10. Les conventions d'écriture.....	9
2.11. Les entrées sorties standard	10
2.12. Un exemple de code.....	10
3. Les types de base.....	11
3.1. Les objectifs	11
3.2. Variables de types valeur ou référence	11
3.3. Valeurs null et types nullable	12
3.4. Les types intégrés	13
3.5. Attribution de valeurs aux variables.....	14
4. Les opérateurs	15
4.1. Arithmétiques	15
4.2. Opérateurs d'affectation	16
4.3. Opérateurs d'incrémentation	16
5. Conversion implicite et explicite	17
6. Le type chaîne.....	18
6.1. Définition.....	18
6.2. Manipulation de chaîne de caractères.....	19
6.2.1. Longueur d'une chaîne de caractères	19
6.2.2. Position d'une chaîne dans une autre.....	19
6.2.3. Extraction d'une sous-chaîne	20
6.2.4. Conversion Majuscules / Minuscules.....	20
6.2.5. Suppression d'espaces	20
6.2.6. Comparaison de chaînes de caractères	20
6.2.7. Remplacement de sous chaînes dans une chaîne	21
6.2.8. Eclatement d'une chaîne en tableau de sous chaînes.....	21
7. Les opérations d'entrée sortie	22

8. Instructions conditionnelles et alternatives	23
8.1. Les structures conditionnelles.....	23
8.1.1. L'action conditionnée.....	23
8.2. La structure alternative	26
8.3. Le choix multiple	27
8.4. Expression de la condition	30
9. Les structures itératives	31
9.1. La nécessité des structures répétitives	31
9.2. Structure répétitive for	32
9.3. La structure While	33
9.4. La structure do	34
9.5. Les rupteurs.....	35
10. Les fonctions	36
10.1. Définition.....	36
10.2. Utilisation des fonctions.....	37
10.3. Le corps de méthode	38
10.3.1. Les variables	38
10.3.2. . L'instruction return	39
10.4. Le passage de paramètres	40
10.4.1. . Déclaration des paramètres	40
10.4.2. . Appel de la méthode	40
10.5. Méthode de passage de paramètres.....	40
10.5.1. Par valeur (paramètres d'entrée)	40
10.5.2. Par référence (paramètres d'entrée/sortie)	41
10.5.3. Paramètres de sortie	42
10.6. La récursivité	42
10.7. La surcharge des méthodes.....	43
11. Les tableaux.....	44
11.1. Définition.....	44
11.2. Déclaration et création.....	45
11.3. Utilisation.....	46
11.3.1. Propriétés	46
11.3.2. Méthodes.....	46
11.3.3. Utilisations particulières	46
11.3.4. L'instruction foreach.....	47
11.3.5. Opérations de tri sur un tableau	47
11.3.6. Recherche d'un élément sur un tableau trié.....	48
12. Autres types de données	49
12.1. Le type enum	49
12.1.1. Utilisation d'un type enum :.....	49
12.2. Le type struct.....	50
12.2.1. Utilisation d'un type struct :.....	50

12.2.2.	Les différences entre classe et structure.....	50
12.3.	Dates et heures	50
12.3.1.	Déclaration d'une date :.....	51
12.3.2.	Utilisation d'un type DateTime :	51
12.3.3.	Utilisation de TimeSpan :	52
13.	Le traitement des erreurs.....	53
13.1.	Les objectifs	53
13.2.	Vue d'ensemble.....	53
13.3.	Les objets Exception	54
13.4.	Les clauses try et catch.....	55
13.5.	Le groupe finally	56
13.6.	L'instruction throw.....	57
14.	Les fichiers.....	58
14.1.	Les objectifs	58
14.2.	Vue d'ensemble.....	58
14.3.	Les classes de flux	59
	Classes de Flux.....	60
14.4.	Utilisation d'un fichier texte	60
14.4.1.	Ouverture d'un fichier	60
	FileMode : énumération des modes d'ouverture de fichier.	61
	FileAccess : énumération d'autorisation de lecture/écriture.	62
	FileShare : énumération d'autorisation de partage.....	62
14.4.2.	Traitement des enregistrements du fichier	63
	Membres de StreamReader	64
	Membres de StreamWriter	65
14.4.3.	Fermeture du fichier.....	66
14.4.4.	Encodage	66
14.5.	Gestion des fichiers sur disque.....	67
14.5.1.	Gestion des répertoires	67
	Membres de Directory (extrait)	67
	Propriétés de DirectoryInfo.....	68
14.5.2.	Gestion des fichiers	68
	Membres de File (extrait).....	68
	Propriétés de FileInfo	69
15.	Annexe 1 : identificateurs du langage C#	70

1. Introduction

Ce document a pour objectifs de vous permettre :

- D'intégrer les principes généraux de la programmation en C# (règles et conventions syntaxiques, règles de présentation d'un programme),
- De vous approprier l'EDI, et les méthodes et outils de débogage par la pratique,
- De poursuivre le travail sur l'algorithmique en vérifiant par la pratique la pertinence des solutions proposées sous formes d'algorithmes,
- De découvrir par la pratique le métier de développeur.

2. Langage C# : Structure d'un programme

2.1. Un peu de vocabulaire

Il est nécessaire, dans un premier temps, de clarifier quelques éléments de vocabulaire propre à la programmation et aux langages informatiques tels que instruction, syntaxe, sémantique, mots-clés et identificateurs. Vous retrouverez fréquemment ces termes dans les chapitres suivants.

Instruction : Une instruction est une commande qui accomplit une action. Les instructions dans C# doivent respecter un ensemble de **règles** qui décrivent leur format et leur construction.

Syntaxe : La syntaxe d'un langage est l'ensemble des règles cumulées qui doivent être respectées par les instructions.

Sémantique : Désigne ce que fait une instruction.

Identificateur : Nom utilisé pour désigner un élément du programme. Dans C# les identificateurs doivent obéir aux règles suivantes :

- Seules des lettres (minuscules ou majuscules, des chiffres et des caractères de soulignement peuvent composer un identificateur
- Un identificateur doit commencer par une lettre ou un trait de soulignement

Mot-clé : il s'agit d'un identificateur réservé par le langage.

2.2. Les mots clés

Le langage C# est un langage peu implémenté. Il existe actuellement environ 80 identificateurs réservés (mots clés C#) et 15 termes fortement déconseillés connus sous le vocable de mots clés contextuels. En fait, ils ne sont pas réservés pour des raisons de compatibilité avec les versions antérieures. Mais vous ne devriez pas les utiliser dans les nouveaux programmes.

Chaque identificateur a une signification particulière que vous découvrirez tout au long de votre apprentissage du langage.

Vous trouverez en annexe 1 la liste des mots clés réservés et des mots clefs contextuels.

2.3. Les espaces de noms

Même si les concepts objets ne seront traités que dans une phase ultérieure de l'apprentissage C#, il est nécessaire d'introduire ici le concept de classe ainsi que le mot clé du langage C# qui permet de l'implémenter.

Une **classe** fournit un mécanisme qui permet de regrouper des éléments sous la forme d'une seule entité logique. Une entité peut représenter des concepts plus ou moins abstrait. Ainsi ; nous pouvons parler de l'entité client, transaction, image, string, ... qui seront autant de classes. Le mot classe est la racine du terme **classification** qui définit un principe d'arrangement et d'organisation systématique des informations.

Une classe est la définition d'un **type**.

Un programme C# est d'abord constitué par un **namespace** (espace de nom en français) Un espace de nom est un conteneur nommé pour les autres identificateurs comme les classes ou types. Un namespace est donc une collection de types, qui se déclare par le mot clé **namespace** de la façon suivante

```
namespace TPCSharp
{
    //Déclaration des classes et fonctions de l'application
}
```

Le nom du namespace (ici **TPCSharp**) est choisi par le programmeur. La construction d'un nom de namespace obéit à la règle standard régissant tous les identificateurs du langage C#.

Ce namespace est un bien un simple regroupement de classes avec leurs méthodes (fonctions) ayant un lien logique ou fonctionnel entre elles. Par exemple, le namespace prédéfini **System** de C# regroupera l'ensemble des classes ayant trait au système lui-même.

Selon le standard C# (ECMA-334), il est fortement **déconseillé** de mettre le même nom à un namespace et à une classe. Les noms de classe et de namespace commencent par une **Majuscule** : notation **PascalCase**

Vous trouverez les différentes règles de notation et les conventions d'écriture dans les documents Convention de nommage C# - Microsoft et Convention de nommage C# - référence ISO 23270 dans le répertoire de cette séance pédagogique.

Le langage C#, tout comme Java, est sensible à la casse. Ceci n'est pas la règle chez d'autres langages comme VB.Net.

2.4. La directive Using

Il existe plusieurs bibliothèques de classes ou namespaces (qui correspondent aux packages de Java) qui offrent au programmeur un grand nombre de fonctionnalités. Ces namespaces sont utilisés grâce à l'instruction **using**.

Pour pouvoir utiliser les fonctions ou les objets des namespaces, il faut que celles-ci soient reconnues par le compilateur (qui ne les connaît pas par défaut). C'est le rôle de la directive **using**. C'est pourquoi on commence très souvent (voire toujours) un programme C# par la directive suivante :

```
using System;
```

Encore une fois, ceci n'est pas obligatoire, mais facilite grandement la tâche du développeur.

En l'absence de directive using et à chaque fois que l'on souhaite utiliser une classe ou une fonctionnalité d'un namespace, il faudrait préfixer cette classe ou cette fonctionnalité par le nom du namespace où elle est définie.

Par exemple : `System.Console.ReadLine()` en l'absence de la directive `using System`.

En utilisant la directive `using`, le préfixe n'est plus obligatoire. Le compilateur ira chercher les fonctionnalités dans les différents namespaces des directives `using`. Dans notre exemple, en mettant `using System`, on pourra utiliser directement et simplement `Console.ReadLine()` dans l'ensemble du programme.

2.5. Le programme principal

Une application C# possède toujours au moins une classe dite « classe application » et une fonction **Main**.

La classe application est simplement la dénomination donnée à la classe contenant la fonction **Main**.

La fonction **Main** est obligatoire et constitue le point d'entrée du programme principal de l'application. Elle se déclare dans la classe de l'application de la façon suivante :

```
namespace TPCsharp
{
    class Program
    {
        static void Main(string[] args)
        {
            // Instructions
        }
    }
}
```

La fonction **Main** est le point d'entrée du programme. Elle est donc obligatoire pour toute application C# et doit être définie de façon unique (il ne peut pas y avoir deux fonctions **Main** dans un programme). Elle doit s'appeler obligatoirement **Main** et doit être suivie par des parenthèses ouvrante et fermante encadrant les arguments (comme toutes les fonctions en C#).

Par défaut, la fonction **Main** reçoit du système d'exploitation un seul argument (**args**) qui est un tableau de chaînes de caractères, chacune d'elles correspondant à un paramètre de la ligne de commande (unix ou MS-DOS).

Le corps du programme suit la fonction **Main** entre deux accolades ouvrante et fermante. Les instructions constituant le programme sont à insérer entre ces deux accolades.

Les mots clés **static** et **void** sont obligatoires pour la fonction **Main**. Ces notions seront plus faciles à comprendre lors de la partie objet du cours (concepts objets proprement dits), mais on peut simplement préciser que :

- **static** : permet au système d'exploitation de ne pas avoir besoin de créer une instance (un objet) de la classe définie ;
- **void** : ce mot clé sera également plus longuement explicité dans la suite du cours. Pour résumer, il s'agit de la déclaration du type de retour de la fonction, à savoir : aucun retour pour le type **void**.

2.6. Les variables

Les données en mémoire sont stockées dans des variables. Il peut par exemple s'agir de données saisies par l'utilisateur ou de résultats obtenus par le programme, intermédiaires ou définitifs.

Pour employer une image, une variable est une **boîte**, que le programme (l'ordinateur) va repérer par une **étiquette**. Pour avoir accès au contenu de la boîte, il suffit de la désigner par son étiquette

Comme tous les autres identificateurs (identificateurs de classes, entre autres) en langage C#, les noms de variables peuvent être choisis librement par le programmeur (sauf parmi les *mots-clés*). Une variable étant choisie pour jouer un rôle, il est souhaitable que l'auteur de l'algorithme lui affecte un nom significatif (évocateur du rôle) et en respectant, autant que faire se peut, l'unicité du rôle pour chaque variable.

Les caractères suivants peuvent être utilisés :

- de A à Z
- de a à z (les minuscules sont considérées comme des caractères différents des majuscules)
- de 0 à 9
- le caractère souligné `_` et le caractère dollar `$` (même en initiale).

Exemple : nom des variables à retenir dans un problème de calcul de moyenne de notes

```
totalNotes    : total des notes d'un(e) élève
nombreNotes   : nombre des notes de cet(te) élève
moyenneNotes  : moyenne des notes de cet(te) élève
```

Style de nommage des variables : appliquer la convention **CamelCase** (annexe 2)

Une variable est typée: elle peut représenter un nombre (entier ou réel), une chaîne de caractères, etc.

Une variable peut être déclarée à tout moment à l'intérieur du corps du programme. Cependant, la déclaration et l'initialisation d'une variable doivent impérativement précéder la première utilisation de celle-ci. Il est également et généralement préférable de déclarer les variables en début de bloc (juste après l'accolade ouvrante) pour une question de lisibilité

Chaque déclaration de variable est construite sur le modèle suivant :

```
type      nomDeLaVariable ;
```

Une variable peut être initialisée lors de sa déclaration :

```
type      nomDeLaVariable = valeurInitiale;
```

Les variables ne sont visibles (reconnues par le compilateur) que dans le bloc d'instructions (défini par `{ }`) dans lequel elles sont définies.

Les variables doivent être typées. Vous verrez peut-être dans certains exemples l'utilisation du mot clé **var** en lieu et place du type. Nous reviendrons sur cette notion, qui permet de définir une variable sans type à priori, par la suite. Ce mot clé doit être réservé pour des usages très spécifiques.

Une variable ne peut être utilisée sans qu'une valeur lui ait été affectée. Nous reviendrons sur ce point lors du traitement de l'affectation.

Comme toutes les instructions du langage C#, chaque déclaration de variable DOIT absolument être terminée par un point-virgule `;` Le point-virgule ne constitue pas un séparateur mais un *terminateur* d'instructions.

Nous utiliserons dans un premier temps des variables de types **primitifs**. Les types de données primitifs sont les types de données intégrés au système. Ils s'opposent aux types **dérivés**. *Ces notions évoquent quelques souvenirs mathématiques relatifs aux fonctions (primitives (antidérivées) – dérivées)*

Le chapitre suivant vous donnera plus de détails sur l'ensemble des types de données C# et .NET.

2.7. Les constantes

Une constante est un espace de stockage de données dont la valeur ne peut changer. Son rôle est de noter des repères, des dimensions, des références invariantes au cours d'un programme.

Les constantes sont définies par un identificateur, et par une valeur. Le nom représente la manière de faire référence à la valeur. La valeur représente le contenu de notre constante. Cette valeur est **invariante**.

Les constantes se déclarent comme les variables initialisées précédées du mot-clef **const**. Leur valeur ne pourra pas être modifiée pendant l'exécution du programme.

```
const Type nomDeLaConstante = valeurInitiale;
```

2.8. Les instructions

Le caractère **;** est un *terminateur*. TOUTES les instructions doivent se terminer par un « **;** ». Le compilateur se refusera de compiler toute instruction qui ne serait pas terminée par un « **;** ».

Les différents identificateurs sont séparés par un *séparateur*.

Les *séparateurs* peuvent être indifféremment l'espace, la *tabulation* et le *saut de ligne*.

Vous êtes libre de disposer vos instructions comme vous l'entendez mais, pour en faciliter la lecture et respecter une certaine cohérence, il est préférable de n'indiquer qu'une instruction par ligne. Voir convention d'écriture.

2.9. Les commentaires

Les caractères compris entre **/*** et ***/** ne seront pas interprétés par le compilateur. Cette syntaxe permet de préciser un bloc de lignes en commentaire.

Les caractères compris entre **//** et un *saut de ligne* ne seront pas interprétés par le compilateur.

Les caractères compris entre **///** et un *saut de ligne* ne seront pas interprétés par le compilateur. Le texte de commentaire peut être utilisé pour générer une documentation de façon automatique.

Nous verrons par la suite comment produire des documentations spécifiques de nos classes et composants dans un fichier xml à l'aide de balises appropriées.

2.10. Les conventions d'écriture

Ces conventions ne sont pas des contraintes de syntaxe du langage C#. Elles n'existent que pour en faciliter la lecture et font partie d'une sorte de norme implicite que tous les bons développeurs s'obligent à respecter.

Une seule instruction par ligne. Même si tout un programme en langage C# peut être écrit sur une seule ligne.

Les délimiteurs d'un bloc { } doivent se trouver sur des lignes différentes et être alignés sur la première colonne de sa déclaration.

A l'intérieur d'un bloc { } les instructions sont indentées (décalées) par un caractère *tabulation*.

A l'intérieur d'un bloc { } la partie *déclaration des variables* et la partie *instructions* sont séparées par une ligne vide.

2.11. Les entrées sorties standard

Les fonctions de lecture et d'écriture standard (clavier/écran) sont définies dans le namespace **System**.

La classe **Console** est définie dans ce namespace et dispose, en autres de trois fonctionnalités :

- Console.ReadLine() : qui retourne une chaîne de caractères (de type string) saisie au clavier jusqu'à la frappe de la touche "Entrée";
- Console.Write() : qui affiche sur l'écran a priori la chaîne de caractères (de type string) envoyée en paramètre ;
- Console.WriteLine() : qui réalise la même opération en ajoutant un caractère retour-chariot après l'affichage de la chaîne de caractères (de type string) envoyée en paramètre.

2.12. Un exemple de code

Le code ci-dessous calcule et restitue la valeur de la circonférence d'un cercle à partir de la saisie à la console de son rayon.

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

namespace TPCSharp
{
    class Cercle
    {
        static void Main()
        {
            //Déclaration de variables.
            string saisie; // variable recevant la saisie de
l'utilisateur
            double rayon; // rayon dans une unité
            double perimetre; // périmètre dans cette même unité

            // Etape 1 : lecture du rayon
            Console.WriteLine("Entrez la valeur du rayon : ");
            saisie = Console.ReadLine();
            // Etape 2 : calcul et affichage du périmètre
            rayon = Convert.ToDouble(saisie);
            perimetre = 2 * Math.PI * rayon; //Calcul du perimetre
            Console.WriteLine("Le cercle de rayon " + rayon);
            Console.WriteLine(" a pour périmetre : " +
perimetre);

            // Permet de conserver l'affichage de la console
            Console.ReadLine();
        }
    }
}
```

- ❶ La fonction Main, point d'entrée du programme
- ❷ La variable **saisie** est déclarée de type **chaîne de caractères**, les variables **rayon** et **perimetre** de type **réel**
- ❸ Affichage à la console du texte « Entrez la valeur du rayon »
- ❹ Dans la variable de nom saisie, sera stockée la chaîne de caractères entrée par l'utilisateur
- ❺ Les variables faisant partie d'une opération (+, -, *, /) doivent être des nombres, entier ou réels. Cette instruction a pour but de **convertir le contenu de la variable saisie de type chaîne en un nombre réel de type double**.
- ❻ La formule magique La valeur de PI est obtenue à partir d'une bibliothèque du framework, la classe Maths, dont le rôle est de fournir des constantes et des fonctions mathématiques et trigonométriques.
- ❼ Affichage à la console du texte « Le cercle de rayon », suivi de la valeur de la variable **rayon** : notez l'opérateur + qui est ici l'opérateur de concaténation.
- ❽ Affichage à la console du texte « a pour périmètre : », suivi de la valeur de la variable **perimetre** .
- ❾ Permet de ne pas terminer le programme pour visualiser l'affichage.

3. Les types de base

3.1. Les objectifs

L'objectif de ce chapitre est l'approfondissement de ce que nous venons de découvrir au travers de ce premier exemple.

- La déclaration et l'utilisation des variables et des constantes.
- La lecture et l'écriture des données numériques sur les flux standard et l'utilisation des manipulateurs.
- Les opérateurs de calcul.

Une variable est un emplacement de stockage en mémoire qui contient une valeur. Le système de types communs (CTS) définit deux types de variables :

- Les types valeur : variables qui contiennent directement leurs données : ce sont les types intégrés, les structures, les variables énumération.
- Les types référence : variables qui stockent des références à des données : leurs données sont stockées dans une zone séparée de la mémoire (classe String, classes).

3.2. Variables de types valeur ou référence

Il faudra retenir qu'en dehors du type string, tous les types primitifs proposés par le système sont des types gérés par valeurs.

Lorsqu'une variable de type valeur est déclarée, le compilateur génère le code qui attribue un bloc de mémoire assez grand pour contenir la valeur correspondante en fonction de la taille occupée (voir liste des types valeur au chapitre suivant).

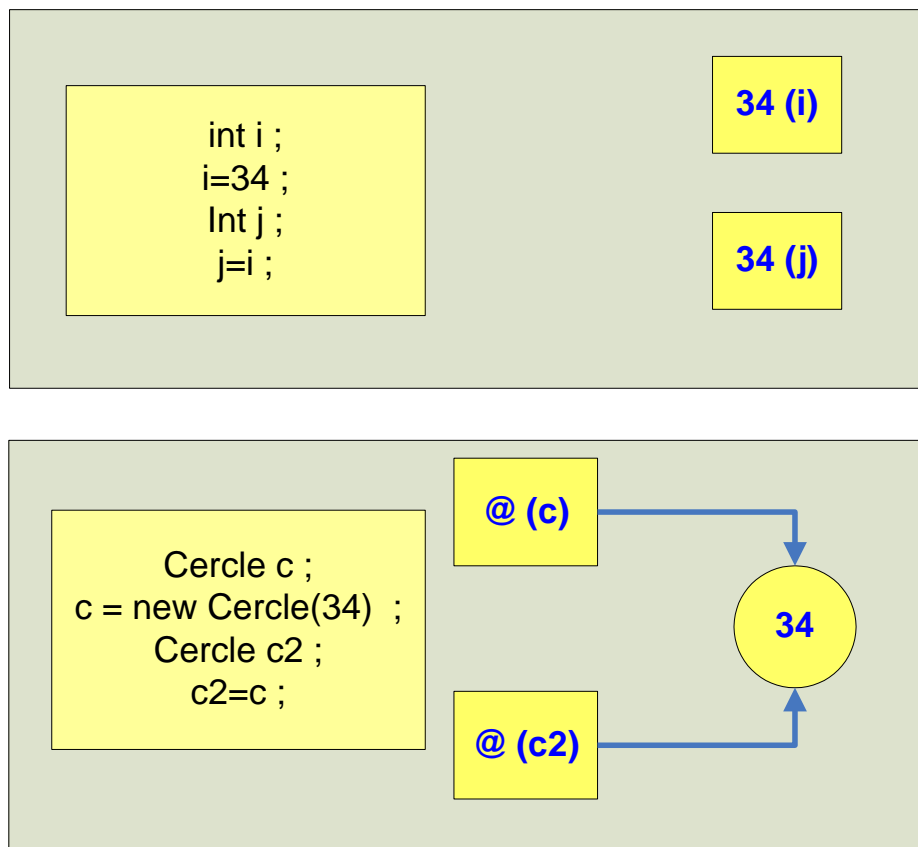
Lorsqu'un type référence est déclaré, le compilateur ne génère pas de code qui attribue un bloc de mémoire pour stocker la valeur mais un petit bloc qui peut contenir l'adresse (ou référence) d'un autre bloc mémoire qui contiendra lui la valeur de l'objet.

Soit une classe de type Cercle. Lorsque la variable de type cercle est déclarée, un bloc de mémoire est réservé pour contenir l'adresse du cercle. La mémoire de l'objet cercle sera attribuée seulement lorsque celui-ci sera instancié (créé) à l'aide du mot clé new. Une classe est de type référence.

Il est fondamental de comprendre la différence entre les types valeur et les types référence. Cela influe notamment la façon dont s'effectue les copies de variables et les modifications de valeur de celles-ci.

Nous reviendrons sur ces concepts lors du cours relatif à l'objet et à la construction des méthodes et du passage de paramètres à celles-ci.

Le schéma suivant explique les deux exemples de fonctionnement.



3.3. Valeurs null et types nullable

Lorsque l'on déclare une variable il est utile de l'initialiser. Avec les types valeurs il est fréquent de rencontrer la syntaxe suivante :

```
int i = 0 ;
```

Pour un type référence, nous trouverons la syntaxe suivante :

```
Cercle c = new Cercle(49)
```

Il est possible d'assigner une valeur spéciale **null** à toute variable de type référence. Nous indiquons ainsi que la variable ne fait pas référence à un objet en mémoire.

La valeur null utile pour initialiser les types référence ne peut être utilisée pour des types gérés par valeur. Ainsi, la syntaxe suivante est incorrecte : `int i = null`.

C# prévoit un modificateur qui peut être utilisé pour indiquer qu'un type valeur est un type valeur nullable. Un type valeur se comportera de la même manière que le type valeur original mais pourra se voir assigner null. Le point d'interrogation ? sera utilisé pour indiquer que le type valeur est nullable.

`int? i = null`

Ainsi comme pour une variable de type référence, on pourra tester si la variable contient une valeur. Vous pouvez comparer la variable à **null** ou utiliser la propriété **HasValue**.

`int? i = null`

`if (i.HasValue) ou if (i == null).`

Il est plus performant de tester directement si une valeur est null plutôt que d'utiliser la propriété HasValue, en particulier sur des types simples.

Les types nullable permettent en particulier de faciliter le traitement des données issues de bases de données. Nous y reviendrons prochainement.

3.4. Les types intégrés

Les types de données simples - de type scalaire (entiers) ou virgule flottante (réels) - sont identifiés par des mots clés réservés.

Type C#	Type .NET	Taille	Val. Min.	Val. Max.
bool	System.Boolean	1	False	True
byte	System.Byte	1	0	255
sbyte	System.SByte	1	-128	127
char	System.Char	2	0	65 535
short	System.Int16	2	-32 768	32 767
ushort	System.UInt16	2	0	65 535
int	System.Int32	4	-2 147 483 648	2 147 483 647
uint	System.UInt32	4	0	4 294 967 295
long	System.Int64	8	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
ulong	System.UInt64	8	0	18 446 744 073 709 551 615
float	System.Single	4	-1.5×10^{-45}	3.4×10^{38}
double	System.Double	8	-5.0×10^{-324}	1.7×10^{308}
decimal	System.Decimal	16	-1.0×10^{-28}	env. 7.9×10^{28}
var	Type défini implicitement lors de l'affectation. Nous verrons comment l'utiliser par la suite. Pour le moment à proscrire...			

La première colonne indique l'alias utilisé en C#.

La seconde colonne indique le type de donnée correspondant dans l'environnement .NET. A noter qu'il est également possible, bien entendu, de les utiliser directement en C# (mais ce n'est pas conseillé).

La troisième colonne donne la taille en octets (place prise en mémoire) d'une variable déclarée avec ce type.

La quatrième colonne indique la valeur minimum admise pour une donnée de ce type.

La cinquième colonne indique la valeur maximum admise pour une donnée de ce type.

Dans le code, il est possible de déclarer la variable de 2 façons :

```
double rayon ou System.Double rayon
```

Les deux instructions permettent de définir un réel signé sur 8 octets.

Les constantes numériques entières peuvent être écrites sous 3 formes :

- en décimal
- en hexadécimal (base 16)
- en caractère.

Ainsi, 65 (décimal), 0x41 (hexadécimal), 'A' désignent la même constante numérique entière (le code ASCII de A est 65).

Exemples :

```
long grandEntier ;           // entier signé sur 8 octets
float leReel ;               // réel en notation virgule flottante (mantisse, exposant)
double dblPrecision ;       // idem en double précision
const int N1=65 ;
const int N2=0x41 ;
const int N3='A';
const int N4 =N1 + N2 ;
const int N5 = N1 + 10 ;
```

3.5. Attribution de valeurs aux variables

Une valeur peut être attribuée à une variable lors de sa déclaration.

```
double pi = 3.14159 ;
```

Ce qui est équivalent à

```
double pi;
pi = 3.14159 ;
```

On parle d'instruction d'**affectation** ou d'**assignation**, et on lit pi **prend pour valeur** 3.14159.

Pour attribuer une valeur à une variable de type caractère, on écrira :

```
char lettre;
lettre = 'A' ;
```

Exemples d'affectation :

```
int resultat, Y = 3 ;           //seule la variable Y est
                                //initialisée à 3
resultat = 5 ;                  // résultat prend la valeur 5
resultat = Y ;                  // résultat prend la valeur 3
resultat = Y + 2 ;              // résultat prend la valeur 5
resultat = resultat + 2 ;       // résultat prend la valeur 7
```

Il est toujours préférable d'utiliser la syntaxe abrégée en ayant recours à la technique de **l'opérateur d'assignation composée**

Plutôt que	Ecrivez ceci
variable = variable * nombre ;	variable *= nombre ;
variable = variable / nombre ;	variable /= nombre ;
Y = Y + 2 ;	Y += 2 ;
Ceci est valable quel que soit l'opérateur. Pour les chaînes, seul l'opérateur + est utilisé pour la concaténation de chaînes.	

Attention :

Le mot clé decimal indique un type de données 128 bits. Par rapport aux types virgule flottante, le type decimal fournit une plus grande précision (valeur exacte) et une plage de valeurs plus réduite ; il est donc particulièrement approprié aux calculs financiers et monétaires. Si vous souhaitez qu'un littéral numérique réel soit considéré comme une valeur de type decimal, utilisez le suffixe m ou M :

decimal dm = 2.5m // syntaxe OK

decimal dm = 2.5 // syntaxe incorrect : est considéré comme un double.

double d = 2.5 ; // syntaxe OK

float f1 = 2.5 ; // pas bon : 2.5 est au format double.

Par défaut, un littéral numérique réel situé à droite de l'opérateur d'assignation est considéré de type double. Par conséquent, si vous souhaitez initialiser une variable de type float, utilisez le suffixe f ou F, comme indiqué ci-après :

float f2 = 2.5f ;
float f3 = (float)¹2.5 ;// syntaxe OK
float f4 = 2.5e10f;

4. Les opérateurs

4.1. Arithmétiques

+	Addition	a + b
-	Soustraction	a - b
-	Changement de signe	-a
*	Multiplication	a * b
/	Division	a / b
%	Reste de la division entière	a % b

Lorsqu'une expression contient plusieurs opérateurs, l'ordre dans lequel sont effectués les calculs dépend de l'ordre de priorité des opérateurs.

L'expression x + y * z est évaluée sous la forme x + (y * z) car l'opérateur de multiplication a une priorité supérieure à celle de l'opérateur d'addition.

¹ Voir le paragraphe relatif au casting

Règle : Les opérateurs – et + ont une priorité plus basse que celle des opérateurs *, / et %. On peut contrôler la priorité à l'aide de parenthèses.

4.2. Opérateurs d'affectation

=	Affectation	<code>a = 5;</code>
+=	Incrémentation	<code>a += b;</code>
	Les deux exemples sont équivalents	<code>a = a + b;</code>
-=	Décrémentation	<code>a -= b;</code>
	Les deux exemples sont équivalents	<code>a = a - b;</code>

4.3. Opérateurs d'incrémentement

++	Pré-incrémentement (+1)	<code>++a</code>
++	Post-incrémentement (+1)	<code>a++</code>
--	Pré-décrémentation (-1)	<code>--a</code>
--	Post-décrémentation (-1)	<code>a--</code>

Les opérateurs d'incrémentement et de décrémentement peuvent donc être placés avant ou après la variable.

Quand le symbole de l'opérateur est placé avant on parlera de la forme préfixée, quand l'opérateur est placé après on parlera de forme postfixée.

Quelle que soit la forme choisie, le résultat de l'incrémentement ou décrémentement sera identique. Mais, et il est important de le souligner ici, la valeur retournée par `compte++` sera différente de la valeur retournée par `++compte`.

Exemples :

```
int a, b;

a = 5;
b = a++;
// a = 6 et b = 5;

a = 5;
b = ++a;
// a = 6 et b = 6;
```

`a++` est incrémenté après affectation, alors que `++a` est incrémenté avant affectation. Autre exemple pour comprendre ce mécanisme.

```
int x;
x=42 ;
console.write(x++) ; // x=43 et 42 est affiché
console.write(++x) ; // x=44 et 44 est affiché
```


5. Conversion implicite et explicite

Quand deux opérandes de part et d'autre d'un opérateur binaire (qui a deux opérandes) sont de types différents, une conversion implicite, lorsqu'elle est possible, est effectuée vers le type "le plus fort" en suivant la relation d'ordre suivante :

bool < byte < char < short < int < long < float < double

Exemple : La conversion d'un type de données **int** en un type de données **long** est implicite : cette conversion réussit toujours et n'entraîne jamais de perte d'informations.

```
int    a = 78;
long   b = a;
```

Par contre, le compilateur ne voudra pas exécuter l'inverse, le code suivant

```
long   a = 78;
int     b = a;
```

Il est impossible de convertir implicitement un type **long** en un type **int**, car le risque de perdre de l'information existe. Une conversion explicite doit être mise en place.

```
long   a = 78;
int     b = (int) a;
```

Une conversion explicite (cast) peut se faire en mentionnant le type entre parenthèse avant l'expression à convertir.

Par ailleurs, un opérateur binaire dont les deux opérandes sont de même type opère dans ce type. Ce qui semble donner parfois des résultats curieux. Pour avoir le résultat attendu, il faut forcer la conversion par un **cast** (changement de type) afin de préciser dans quel référentiel on opère.

Exemples :

```
int n1 = 5;
int n2 = 2;
double x = n1 / n2;
```

Ici, la valeur de x est de 2.0. En effet, 5 et 2 sont des entiers. Le résultat de la division entière de ces deux nombres est 2 (et il reste 1) et non pas 2.5. Pour obtenir cette dernière valeur, il est possible de forcer la conversion de l'une des opérandes en un nombre virgule flottante :

```
double x = (double)n1 / n2;
```

Le fait de faire le **cast (double)** devant 5 force la conversion de 5 (type **int**) en double (type **double**). L'opérateur / va donc devoir opérer sur un **double** et un **int**. Il y a alors une conversion implicite de 2 (type **int**) en type **double**. L'opérateur peut maintenant opérer sur deux types **double** et exprimer le résultat sous cette forme.

6. Le type chaîne

6.1. Définition

En C#, une chaîne de caractères est un objet de la classe **String**, de l'espace de nom **System**.

Le texte d'une chaîne ne peut pas être modifié après sa création (immuable).

Une chaîne peut être déclarée à l'aide du type ou de l'alias. Préférer l'alias.

```
string strChaine ; ou String strChaine;
```

La variable `strChaine` ne *contient* pas la chaîne de caractères : elle est la référence d'une chaîne de caractères. Telle qu'elle est déclarée ci-dessus, la valeur de `strChaine` est **null**.

On pourra la créer en affectant un littéral de type chaîne.

```
strChaine = "Bonjour" ;    // si elle est préalablement déclarée
ou
```

```
string strChaine = "Bonjour" ;
```

Entre les doubles-quotes, le caractère `\` est utilisé comme modificateur pour coder des caractères d'échappement non-affichables comme le saut de ligne ou une tabulation ou pour empêcher l'interprétation du caractère (' ou ") ou spécifier des caractères dont nous précisons la valeur dans une notation hexadécimale ou décimale.

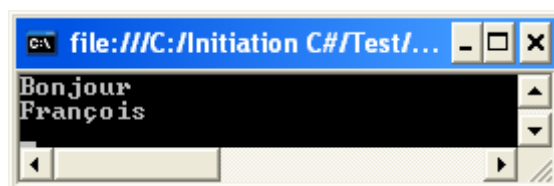
<code>\n</code>	Saut de ligne
<code>\r</code>	Retour de chariot
<code>\t</code>	Tabulation
<code>\b</code>	Retour arrière
<code>\v</code>	Tabulation verticale
<code>\f</code>	Saut de page
<code>\\</code>	Le caractère <code>\</code> lui-même
<code>\'</code>	Le caractère ' (quote) lui-même
<code>\"</code>	Le caractère " (double quote) lui-même
<code>\xHH</code>	Caractère de code HH en hexadécimal
<code>\NNN</code>	Caractère de code ASCII NNN en décimal

Exemple : Le code suivant

```
string strNom = "François";

Console.WriteLine("Bonjour \n" + strNom);
```

produira l'affichage suivant :



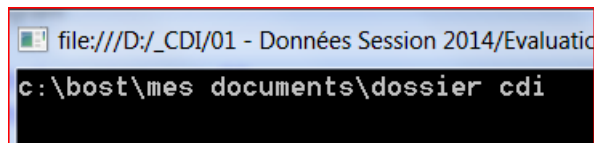
Dans ce contexte, vous devrez donc doubler le caractère \ pour qu'il soit affiché.

Vous pouvez aussi utiliser la syntaxe *verbatim* pour simplifier une expression de type chaîne où plusieurs occurrences du caractère antislash seraient présentes. Vous devez alors faire précéder votre chaîne du caractère @.

Ainsi, le code :

```
string sVerbatim = @"c:\bost\mes documents\dossier cdi";
Console.WriteLine(sVerbatim);
```

Fournira le résultat suivant :



La **concaténation** de chaînes de caractères s'obtient grâce à l'opérateur +. Le résultat de cette opération est une nouvelle chaîne de caractères qui peut être affectée à une variable de type **String** :

```
string strMsg = "Bonjour " + strPrenom;
```

On accède à un caractère de rang fixé d'une chaîne par l'opérateur [] (la chaîne est lue comme un tableau de char, le premier caractère est indicé par 0) :

```
char lettre = strNom[3] ; // lettre contient le caractère n
```

par contre

```
strNom [3] = "B"; // provoque une erreur de compilation
```

6.2. Manipulation de chaîne de caractères

La classe **String** possède intrinsèquement les méthodes (fonctions membres) nécessaires à la manipulation des chaînes de caractères. Comme beaucoup de méthodes définies dans une classe d'objet, elles s'utilisent conjointement à une variable de type **String** avec le caractère « . ».

Toutes les méthodes de la classe **String**, ne seront pas traitées ici. Pour plus de précisions la documentation en ligne.

6.2.1. Longueur d'une chaîne de caractères

La longueur d'une chaîne de caractères est calculée par la propriété **Length** d'un objet de la classe **String**.

```
string strTitre = "Support de cours C#";
int iLen = strTitre.Length; // iLen vaudra donc 19
```

6.2.2. Position d'une chaîne dans une autre

La méthode d'instance **IndexOf** permet de calculer la position d'une séquence de caractères dans une chaîne. La valeur 0 de cette position correspond au premier caractère de la chaîne. La valeur -1 est retournée si la séquence de caractères n'a pas été trouvée dans la chaîne. Il existe plusieurs versions de cette fonction dont les plus utilisées sont les suivantes :

```
int pos1 = str1.IndexOf(str2);
int pos2 = str1.IndexOf(str2, iPos);
```

Dans le premier cas **pos1** contiendra la position de **str2** dans **str1**. Dans le second cas, la position de **str2** sera cherchée à partir du **iPos^{ième}** caractère de **str1**. Ceci permet de repérer plusieurs occurrences de **str2** dans **str1** en utilisant une boucle de parcours.

Exemples:

```
string str1 = "Pascal disait: je pense, donc je suis";
int pos1 = str1.IndexOf( "je" );           // pos1 = 15
int pos2 = str1.IndexOf( "je", pos1 + 1 ); // pos2 = 30
```

6.2.3. Extraction d'une sous-chaîne

La méthode d'instance **Substring** permet d'extraire une sous-chaîne d'une chaîne de caractères :

```
string strPrenom = strNom.Substring(0, 6);
// Extraite les 6 premiers caractères de la chaîne strNom
```

6.2.4. Conversion Majuscules / Minuscules

Les méthodes d'instance **ToUpper** et **ToLower** permettent de retourner une chaîne dont tous les caractères sont convertis respectivement en majuscules et en minuscules.

```
string strSalut = "Bonjour Tout Le Monde";
Console.WriteLine(strSalut.ToUpper()); // BONJOUR TOUT LE MONDE
Console.WriteLine(strSalut.ToLower()); // bonjour tout le monde
```

6.2.5. Suppression d'espaces

La méthode d'instance **Trim** supprime tous les espaces de début et fin de chaîne.

```
string strSalut = " Bonjour ";
Console.WriteLine(strSalut.Trim()); // Bonjour
```

6.2.6. Comparaison de chaînes de caractères

La méthode **Equals** permet de comparer l'égalité de 2 chaînes : elle renvoie un booléen positionné à **true / false** si les 2 chaînes sont identiques/différentes ; elle peut être employée de 2 façons :

Exemple :

```
string str1 = "abc";
string str2 = "abc";
```

En tant que méthode d'instance

```
bool bIden = str1.Equals(str2); // bIden = true
```

En tant que méthode de classe

```
bool bIden = String.Equals(str1, str2); // bIden = true
```

Notez la différence entre une méthode de classe et une méthode d'instance : nous reviendrons sur le sujet dans un prochain chapitre.

La méthode de classe **Compare** permet de comparer les chaînes en fonction de leur ordre de tri : elle renvoie **0** si les chaînes sont identiques, **-1** si la 1ère chaîne précède la 2ème, **1** si la deuxième précède la 1ère.

```
string str1 = "abc";
string str2 = "def";
int iIden = String.Compare(str1, str2); // iIden = -1
```

Une autre version de la méthode permet de positionner un booléen en 3eme paramètre qui spécifiera si la casse doit être ignorée ou non.

```
string str1 = "abc";  
string str2 = "Abc";  
int iIden = String.Compare(str1, str2, true); // iIden = 0
```

La casse est ignorée.

6.2.7. Remplacement de sous chaînes dans une chaîne

La méthode **Replace** permet de remplacer toutes les occurrences d'une sous chaîne de la chaîne de base par une autre.

```
string str1 = "Bonjour";  
string str2 = str1.Replace("on", "ON ") ;  
// str2 contient "BON jour", str1 est inchangé
```

6.2.8. Eclatement d'une chaîne en tableau de sous chaînes

La méthode **Split** permet de retourner un tableau de mots constituant la chaîne de base en spécifiant le(ou les) séparateur(s).

```
string strSalut = "Bonjour Tout Le Monde";  
string[ ] tsc = strSalut.Split( ' ' ) ; // un seul séparateur  
// tsc[0] contient "Bonjour", tsc[1] contient "Tout",  
// tsc[2] contient "Le", tsc[3] contient "Monde",
```

7. Les opérations d'entrée sortie

L'affichage des données se fait à travers le flux de sortie de l'objet Console. via ses méthodes Write() ou WriteLine().

La méthode WriteLine() permet de passer à la ligne suivante.

Ces méthodes affichent à l'écran la représentation textuelle de l'objet défini en argument.

```
int i=5 ;
Console.WriteLine(i);
```

La méthode affiche ici la représentation textuelle de l'entier i. Il existe autant d'implémentation de cette méthode que de types de base.

La méthode Writeline() fait appel à la méthode de mise en forme des chaînes de caractères.

Soit deux variables de type int i et j

```
int i=5 , j=4 ;
```

Nous pouvons afficher les valeurs de i et j ainsi :

```
Console.WriteLine(" i = " + i + ", j= " + j);
```

Mais il est préférable de les afficher ainsi :

```
Console.WriteLine(" i = {0}, j= {1} ", i, j);
```

Les valeurs de i et j sont respectivement positionnées en place des paramètres 0 et 1

Mise en format de chaînes de caractères

Chaque spécificateur a la forme {N [, M] : *chaîne de format* } où :

- N désigne le numéro de l'argument (0,1)
- M, facultatif désigne le nombre de positions d'affichage, valeur positive pour un alignement à droite, ou négative pour un alignement à gauche
- La chaîne de format, facultative peut représenter un format standard

```
double unDouble = 123456789;
Console.WriteLine(" le double = {0: e} ", unDouble);
// rend le double = 1.2345678e+008
Console.WriteLine(" le double = {0: f} ", unDouble);
// rend le double = 123456789.00
Console.WriteLine(" le double = {0:n}", unDouble);
// rend le double = 123 456 789.00
```

ou personnalisé, où les caractères suivants ont une signification particulière :

- 0 représente un chiffre
- # représente un chiffre ou rien
- % le nombre sera multiplié par 100
- . représente le séparateur de milliers
- E indique une représentation scientifique

```
double unDouble = 34567.89;
Console.WriteLine(" le double = {0:##,###.##} ", unDouble);
// le double = 34 567,89

int unEntier = 34;
Console.WriteLine(" l'entier = {0,5:000} ", unEntier);
// l'entier = 034,00 La valeur 34 est précédée de 2 espaces.

int unEntier = 34;
Console.WriteLine(" l'entier = {0:000.00} ", unEntier);
// l'entier = 034,00
```

(Consultez l'aide en ligne pour plus de détails)

La lecture des données numériques se fait à travers le flux d'entrée **Console**. à l'aide de la méthode **ReadLine**. Cette méthode pose un problème principal : Elle ne permet de ne lire qu'une chaîne de caractères **string** Il faut donc convertir explicitement cette chaîne de caractères pour obtenir l'objet numérique souhaité.

Exemples :

```
// Pour lire l'entier k déclaré de type int
string strLine;
strLine = Console.ReadLine();
int k = Convert.ToInt32( strLine );
//ou encore
k = Int32.Parse( strLine );
```

La classe **System.Convert** fournit un jeu complet de méthodes pour les conversions prises en charge. On peut, par exemple convertir des types **string** en types numériques, des types **DateTime** en types **string** et des types **string** en types **boolean**.

D'autre part, tous les types numériques disposent d'une méthode **Parse** statique pouvant être utilisée pour convertir une représentation sous forme de chaîne d'un type numérique en un type numérique réel.

Les deux méthodes sont équivalentes.

Attention : Le programme se plantera si l'utilisateur introduit autre chose qu'un entier correct. Le problème sera résolu dans un chapitre ultérieur.

```
// Pour lire le nombre dSalaire déclaré de type double
string strLine;
strLine = Console.ReadLine();
double dSalaire = Convert.ToDouble( strLine );
// ou encore
dSalaire = Double.Parse( strLine );
```

Le programme génère une erreur si l'utilisateur introduit autre chose qu'un réel. Le séparateur de décimales saisi doit être conforme aux caractéristiques régionales.

8. Instructions conditionnelles et alternatives

8.1. Les structures conditionnelles

8.1.1. L'action conditionnée

L'action conditionnée est une instruction élémentaire ou une suite d'instructions **exécutées** en séquence **si l'état du système l'autorise**. Le(s) critère(s) à respecter pour exécuter l'action s'exprime(nt) à l'aide d'une condition (ou **prédicat**) évaluable au moment précis où l'action doit, le cas échéant, intervenir.

Lors de l'exécution du programme, le processeur est donc amené à évaluer la condition. La condition évaluée constitue alors un énoncé (ou **proposition**) vrai ou faux.

Schéma :

```
Si prédicat alors
Instruction 1
Instruction 2
...
Instruction N
Fin si
```

Exemples :

Si Température > 38 alors Écrire "Le patient a de la fièvre" Fin si	Si Température > 41 et Tension > 25 alors Écrire "Le patient va perdre patience" Fin si
Si non Patient alors Écrire "Éconduire l'olibrius" Fin si	Si Température > 42 ou (Tension < 25 et Pouls > 180) alors Écrire "Prévenir la famille" Fin si
Si Température > 40 ou Tension ≥ 25 alors Écrire "Hospitaliser le patient" Fin si	Si Patient et Pouls = 0 alors Patient ← non Patient Fin si

Syntaxe C# :

```

if (prédicat)
{
    Instruction 1
    Instruction 2
    ...
    Instruction N
}

```

Exemple 1: réaliser le programme qui après lecture d'un entier au clavier, affiche un message si l'entier est pair.

```

class Program
{
    static void Main(string[] args)
    {
        string strLigne;
        int entier;
        int reste;
        // saisie utilisateur
        Console.WriteLine("Saisissez un entier :");
        strLigne = Console.ReadLine();
        // conversion en entier
        entier = Int32.Parse(strLigne);
        // Reste de la division par 2
        reste = entier % 2;
        if (reste == 0)
        {
            Console.WriteLine("Entier pair !");
        }

        // Affichage persistant
        Console.ReadLine();
    }
}

```

Notez l'opérateur relationnel == qui teste l'égalité. Tous les opérateurs relationnels seront vus dans le prochain paragraphe. Remarquez l'indentation (retrait de paragraphe) qui permet de voir facilement la portée de la condition.

A noter : si une seule instruction est à exécuter dans le IF, les accolades ne sont pas obligatoires ...

```

if (reste == 0)
    Console.WriteLine("Entier pair !");

```


Mais le code est tellement plus lisible avec les accolades et facilite l'ajout de nouvelles instructions.

Les conditions peuvent être imbriquées :

Exemple 2: Modifier le programme précédent qui après lecture d'un entier au clavier, affiche un message si l'entier est pair ; indiquez également si l'entier est multiple de 4.

```
// Reste de la division par 2
if (entier % 2 == 0)
{
    Console.WriteLine ("Entier pair !");
    // Reste de la division par 4
    if (entier % 4 == 0)
    {
        Console.WriteLine("Entier multiple de 4 !");
    }
}
```

Pourquoi les conditions sont-elles imbriquées ? Tout simplement parce qu'un nombre impair ne peut pas être un multiple de 4... Le traitement ne concerne donc que les nombres pairs.

Notez la notation différente du prédicat dans cet exemple ... Les 2 notations sont équivalentes, l'important étant que le prédicat puisse être évalué.

Exemple 3: Et de la même façon, si on avait voulu également savoir si l'entier était également multiple de 6, on aurait pu écrire :

```
// Reste de la division par 2
if (entier % 2 == 0)
{
    Console.WriteLine ("Entier pair !");
    // Reste de la division par 4
    if (entier % 4 == 0)
    {
        Console.WriteLine("Entier multiple de 4 !");
    }

    // Reste de la division par 6
    if (entier % 6 == 0)
    {
        Console.WriteLine("Entier multiple de 6 !");
    }
}
```

Un entier pair peut être à la fois multiple de 4 et de 6 (12).
Mais un multiple de 4 ou de 6 est toujours un multiple de 2.

8.2. La structure alternative

La structure alternative traduit la nécessité d'exécuter, à un instant donné, une action, élémentaire ou composée, ou une autre action, elle-même élémentaire ou composée, exclusivement l'une de l'autre.

Schéma :

Si prédicat alors
Instructions_si_vrai;
Sinon
Instructions_si_faux;
Fin si

Exemples :

Si Nombre = 0 alors Écrire "Impossible de diviser par 0" sinon Moyenne = Total / Nombre Écrire Moyenne Fin si	Si C= "A" ou C= "E" ou C= "I" ou C= "O" ou C= "U" alors Écrire C, " est une voyelle" sinon Écrire C, " est une consonne" Fin si
--	---

Syntaxe C# :

```
if (prédicat)
{
    Instructions_si_vrai;
}
else
{
    Instructions_si_faux;
}
```

Exemple : Modifier le programme précédent qui après lecture d'un entier au clavier, affiche un message si l'entier est pair, pour qu'il affiche un message si l'entier est impair.

```
...
// Reste de la division par 2
if (entier % 2 == 0)
{
    Console.WriteLine ("Entier pair !");
// Reste de la division par 4
if (entier % 4 == 0)
{
    Console.WriteLine("Entier multiple de 4 !");
}

// Reste de la division par 6
if (entier % 6 == 0)
{
    Console.WriteLine("Entier multiple de 6 !");
}
}
else
{
    Console.WriteLine("Entier impair !");
}
```

8.3. Le choix multiple

La sélection (ou **choix multiple**) permet de présenter une solution claire à des problèmes où un nombre important de cas, mutuellement exclusifs, sont à envisager en fonction des valeurs prises par un nombre réduit de variables (généralement une seule). Puisque chaque action est exclusive des autres, la structure sélective correspond à une **imbrication d'alternatives**.

Cette structure vise en fait à réduire la complexité engendrée par la présence d'un nombre élevé de cas à envisager. Elle privilégie ainsi l'essentiel (la recherche des différents cas et des actions à leur associer) sans négliger l'accessoire (la lisibilité de la solution).

Exemple 1: En fonction du caractère saisi à l'écran, affichez le statut de la personne en clair.

```
namespace Test
{
    class Statut
    {
        static void Main(string[] args)
        {

            string strStatut;
            // saisie utilisateur
            Console.WriteLine("Saisissez le statut :");
            strStatut = Console.ReadLine();

            if (strStatut == "C")
            {
                Console.WriteLine("Célibataire !");
            }
            else
            {
                if (strStatut == "M")
                {
                    Console.WriteLine("Marié !");
                }
                else
                {
                    if (strStatut == "V")
                    {
                        Console.WriteLine("Veuf !");
                    }
                    else
                    {
                        if (strStatut == "D")
                        {
                            Console.WriteLine("Divorcé !");
                        }
                    }
                }
            }
            // Affichage persistant
            Console.ReadLine();
        }
    }
}
```

Traité avec une imbrication de if, la solution est peu lisible, mais peut être améliorée en utilisant le **If étendu** :

Exemple 2:

```
namespace Test
{
    class Statut
    {
        static void Main(string[] args)
        {

            string strStatut;
            // saisie utilisateur
            Console.WriteLine("Saisissez le statut :");
            strStatut = Console.ReadLine();

            if (strStatut == "C")
            {
                Console.WriteLine("Célibataire !");
            }
            else if (strStatut == "M")
            {
                Console.WriteLine("Marié !");
            }
            else if (strStatut == "V")
            {
                Console.WriteLine("Veuf !");
            }
            else if (strStatut == "D")
            {
                Console.WriteLine("Divorcé !");
            }
            // Affichage persistant
            Console.ReadLine();
        }
    }
}
```

L'utilisation de l'instruction **switch** va encore apporter une autre lisibilité :

Syntaxe C#

```
switch (expression)
{
    case valeur1:
        Instructions_Valeur1;
        break;
    case valeur2:
        Instructions_Valeur2;
        break;
    case valeur3:
        Instructions_Valeur3;
        break;
    default:
        Instructions_Default;
        break;
}
```

Si **expression** est égale à **valeur1** on exécute les instructions **Instructions_Valeur1**.
 Si **expression** est égale à **valeur2** on exécute les instructions **Instructions_Valeur2**.
 Si **expression** est égale à **valeur3** on exécute les instructions **Instructions_Valeur3**.
 Si **expression** n'est égale à aucune des valeurs énumérées, on exécute **Instructions_Default**.

Pour éviter d'exécuter tous les pavés d'instructions en séquence à partir du moment où l'égalité a été trouvée, il est obligatoire de mettre une instruction **break** (rupteur) à la fin de chaque pavé : cette instruction provoque un débranchement à l'instruction suivant l'accolade fermante. Ce n'est pas le cas de d'autres langages tels que VB par ailleurs, où il n'est pas nécessaire de placer un rupteur.

Exemple 3:

```
namespace Test
{
    class Statut
    {
        static void Main(string[] args)
        {

            string strStatut;
            // saisie utilisateur
            Console.WriteLine("Saisissez le statut :");
            strStatut = Console.ReadLine();
            switch (strStatut)
            {
                case "C":
                    Console.WriteLine("Célibataire !");
                    break;
                case "M":
                    Console.WriteLine("Marié !");
                    break;
                case "V":
                    Console.WriteLine("Veuf !");
                    break;
                case "D":
                    Console.WriteLine("Divorcé !");
                    break;
                default :
                    Console.WriteLine("Saisie incorrecte !");
                    break;
            }
            // Affichage persistant
            Console.ReadLine();
        }
    }
}
```

8.4. Expression de la condition

Opérateurs relationnels

Si les opérandes peuvent être des expressions arithmétiques quelconques, le résultat de ces opérateurs est de type `bool`.

<code>==</code>	Egalité	<code>a == b</code>
<code>!=</code>	Différence	<code>a != b</code>
<code><</code>	Inférieur	<code>a < b</code>
<code>></code>	Supérieur	<code>a > b</code>
<code><=</code>	Inférieur ou égal	<code>a <= b</code>
<code>>=</code>	Supérieur ou égal	<code>a >= b</code>

Ce résultat peut, bien sur, être utilisé comme opérande des opérateurs booléens

Opérateurs booléens

Les opérateurs booléens ne reçoivent que des opérandes de type `bool`. Ces opérandes peuvent avoir la valeur vraie (`true`) ou faux (`false`).

<code>&</code>	ET logique Le résultat donne vrai (valeur <code>true</code>) si les deux opérandes ont pour valeur vrai Le résultat donne faux (valeur <code>false</code>) si l'un des deux opérandes a pour valeur faux.	<code>a > b & a < c</code>
<code>&&</code>	ET logique Cet opérateur fonctionne comme le précédent, à la différence que le deuxième opérande (à droite) n'est pas évalué (calculé) si le premier a pour valeur faux. Car quelle que soit la valeur du deuxième opérande, le résultat est forcément faux.	<code>a > b && a < c</code>
<code> </code>	OU logique Le résultat donne vrai (valeur <code>true</code>) si l'un des deux opérandes a pour valeur vrai. Le résultat donne faux (valeur <code>false</code>) si les deux opérandes ont pour valeur faux.	<code>a == b a == c</code>
<code> </code>	OU logique Cet opérateur fonctionne comme le précédent, à la différence que le deuxième opérande (à droite) n'est pas évalué (calculé) si le premier a pour valeur vrai. Car quelle que soit la valeur du deuxième opérande, le résultat est forcément vrai.	<code>a == b a == c</code>
<code>!</code>	NON logique Le résultat est faux si l'expression est vraie et inversement	<code>! (a <= b)</code>

Autre opérateur

Il existe, en C#, un opérateur à trois opérandes dit ternaire :

`? :` Structure alternative. `n = k > 3 ? 5 : 6;`

L'exemple sera interprété : Si `k` est supérieur à `3`, `n` vaudra `5` sinon `6`.

9. Les structures itératives

9.1. La nécessité des structures répétitives

L'itération (ou structure répétitive ou **boucle**) permet d'obtenir une action composée par la répétition d'une action élémentaire ou composée, répétition qui continue tant qu'une condition n'est pas remplie, ou cesse lorsqu'une condition donnée est remplie

Exemple : La table de multiplication par 5

Avec les instructions définies à ce stade, la seule possibilité d'écrire la table en totalité est donnée par le programme ci-dessous.

```
namespace Repetitives
{
    class Test1
    {
        static void Main()
        {
            Console.WriteLine("Table de multiplication par 5");
            Console.WriteLine("=====");
            Console.WriteLine("{0} * 5 = {1}", 1, 1 * 5);
            Console.WriteLine("{0} * 5 = {1}", 2, 2 * 5);
            Console.WriteLine("{0} * 5 = {1}", 3, 3 * 5);
            Console.WriteLine("{0} * 5 = {1}", 4, 4 * 5);
            Console.WriteLine("{0} * 5 = {1}", 5, 5 * 5);
            Console.WriteLine("{0} * 5 = {1}", 6, 6 * 5);
            Console.WriteLine("{0} * 5 = {1}", 7, 7 * 5);
            Console.WriteLine("{0} * 5 = {1}", 8, 8 * 5);
            Console.WriteLine("{0} * 5 = {1}", 9, 9 * 5);
            Console.WriteLine("{0} * 5 = {1}", 10, 10 * 5);

            // Affichage persistant
            Console.ReadLine();

        }
    }
}
```

A la lecture de ce programme, on s'aperçoit vite que la même action élémentaire (moyennant paramétrage) est répétée un certain nombre de fois.

En généralisant, on peut écrire que l'instruction suivante

```
Console.WriteLine("{0} * 5 = {1}", i, i * 5);
```

Se répète pour une valeur de *i*, variant de 1 à 10, la condition d'arrêt pouvant aussi s'énoncer

Pour *i* variant de 1 à 10

Ou

Tant que *i* <= 10

Ou

Jusqu'à *i* > 10

On distingue généralement plusieurs types de structures répétitives.

9.2. Structure répétitive for

Syntaxe C#

```
for ( expression_a; expression_b; expression_c )
{
    // Instructions;
}
```

expression_a représente l'initialisation des itérateurs ;

expression_b représente la condition d'itération ;

expression_c représente l'actualisation des itérateurs ;

Dans l'exemple de la table de multiplication :

```
{
class Test1
{
static void Main()
{
Console.WriteLine("Table de multiplication par 5");
Console.WriteLine("=====");
for (int i =1; i <= 10 ; i++)
{
Console.WriteLine("{0} * 5 = {1}", i, i * 5);
}
// Affichage persistant
Console.ReadLine();
}
}
```

Déroulement de l'exécution :

- Lors de la première exécution de l'instruction for, **i** est initialisée à **1** ;
- A chaque exécution, la condition d'itération (**i <= 10**) est évaluée ; si **i > 10**, la boucle s'arrête, et l'instruction suivant l'accolade fermante est exécutée.
- Lorsque la condition d'itération est vraie, les instructions entre accolades sont exécutées.
- Sur l'accolade fermante, **i** est incrémenté de **1**
- Retour sur l'instruction **for**

9.3. La structure While

Syntaxe C# :

```
while ( condition )
{
    // Instructions;
}
```

condition est une expression booléenne (type `bool`). Les **instructions** sont exécutées plusieurs fois tant que le résultat de l'expression **condition** est vraie (valeur `true`).

La condition doit pouvoir être évaluée à la première exécution de l'instruction **while**, ce qui nécessite toujours l'initialisation de la (des) variable(s) intervenant dans la condition.

Si à la première exécution du **while**, le résultat de l'expression **condition** est faux (valeur `false`), les **instructions** ne sont jamais exécutées.

Les instructions seront donc exécutées de 0 à n fois.

Exemple : La table de multiplication par 5

```
namespace Repetitives
{
    class Test2
    {
        static void Main()
        {
            int i;
            Console.WriteLine("Table de multiplication par 5");
            Console.WriteLine("=====");
            i = 1;
            while (i <= 10)
            {
                Console.WriteLine("{0} * 5 = {1}", i, i * 5);
                i++;
            }
            // Affichage persistant
            Console.ReadLine();
        }
    }
}
```

Déroulement de l'exécution :

- A chaque exécution de l'instruction **while**, la condition d'itération (`i <= 10`) est évaluée ; si `i > 10`, la boucle s'arrête, et l'instruction suivant l'accolade fermante est exécutée.
- Lorsque la condition d'itération est vraie, les instructions entre accolades sont exécutées.
- Sur l'accolade fermante, retour sur l'instruction **while**

L'instruction **while** nécessite une attention soutenue : Sa syntaxe complète est :

initialisation

```
while ( condition )
{
    // Instructions;
    // actualisation
}
```

Dans notre exemple,

```
i = 1;
while (i <= 10)
{
    Console.WriteLine("{0} * 5 = {1}", i, i * 5);
    i++;
}
```

The diagram highlights the initialization part (`i = 1;`) with a red box and a callout labeled "Initialisation". It also highlights the update part (`i++;`) with a red box and a callout labeled "Actualisation,".

Surtout, ne pas oublier la partie Actualisation

L'instruction **while** par rapport à l'instruction **for** présente l'intérêt de pouvoir évaluer une condition d'itération complexe, par exemple :

```
while ((i <= 10) && (j != 2)) {...}
```

ou

```
while (!trouve) {...}           // bool trouve
```

9.4. La structure do

Syntaxe C# :

```
do
{
    // Instructions;
} while ( condition );
```

condition est une expression booléenne (type `bool`). Les **Instructions** sont exécutées plusieurs fois tant que le résultat de l'expression **condition** est vraie (valeur `true`).

L'instruction **do** est toujours accompagnée d'une instruction **while**.

Elle est similaire à l'instruction **while**, sauf que l'évaluation de la condition d'itération s'effectue en fin de boucle, et non pas au début, ce qui signifie que, contrairement à l'instruction **while** qui est exécutée de **0** à **n** fois, une instruction **do** est exécutée **au moins une fois**.

Langage C# Partie 1

```
namespace Repetitives
{
    class Test3
    {
        static void Main()
        {
            int i;
            Console.WriteLine("Table de multiplication par 5");
            Console.WriteLine("=====");
            i = 1;
            do
            {
                Console.WriteLine("{0} * 5 = {1}", i, i * 5);
                i++;
            } while (i <= 10)
            // Affichage persistant
            Console.ReadLine();
        }
    }
}
```

Déroulement de l'exécution :

- Les instructions entre accolades sont exécutées.
- A chaque exécution de l'instruction **while**, la condition d'itération ($i \leq 10$) est évaluée ; si $i > 10$, la boucle s'arrête, et l'instruction suivant l'accolade fermante est exécutée. Lorsque la condition d'itération est vraie, retour sur l'instruction **do**

9.5. Les rupteurs

Les rupteurs, en C#, sont des instructions de branchement inconditionnel à des points stratégiques du programme.

continue	arrêt de la boucle en cours et débranchement à l'instruction responsable de la boucle.
break	arrêt de la boucle en cours et débranchement derrière l'accolade fermante.
return	dans une fonction retour au programme appelant quel que soit le niveau d'imbrication des structures. La valeur n constitue le résultat de la fonction. Si l'instruction return est trouvée dans la fonction Main , on retourne au système d'exploitation. La valeur n constitue alors la valeur de retour du programme (<i>status</i>) qui peut être utilisé par le système d'exploitation (IF ERRORLEVEL sous DOS).

Les rupteurs sont parfois dangereux pour une structuration correcte d'un programme.

Ils doivent être utilisés à bon escient. Ce sont des GOTO déguisés et ils trahissent trop souvent une structure de programmation peu rigoureuse. A proscrire dans la plupart des cas sauf bien entendu lorsqu'ils sont obligatoires comme dans le cas du switch.

10. Les fonctions

10.1. Définition

Une fonction est une partie de programme comportant un ensemble d'instructions qui a besoin d'être utilisé plusieurs fois dans un programme ou dans différents programmes.

Parce que C# est un langage orienté objet, les fonctions ne peuvent être déclarées qu'à l'intérieur d'une classe. Il est donc impossible de déclarer des fonctions isolées.

Une fonction peut ou non renvoyer un résultat. Ce résultat n'est pas forcément exploité. Certaines fonctions (déclarée `void`) ne renvoient pas de résultat.

Exemple :

```
namespace Fonctions
{
    class Test1
    {

        // définition des fonctions
        static void AfficheMessage()
        {
            Console.WriteLine("Ceci est un message");
        }
        static bool ChercheEtTrouve()
        {
            bool btrouve;
            // ....
            return btrouve;
        }

        // programme principal
        static void Main()
        {
            Console.WriteLine("Programme principal");
            // ....
            AfficheMessage();
            // ....
            AfficheMessage();
            // ....
            bool btrouve = ChercheEtTrouve()
        }
    }
}
```

Dans cet exemple, 4 fonctions ou méthodes sont nommées :

- `Main` est le point d'entrée de l'application
- `WriteLine` est une méthode de la classe `System.Console` du Framework.
- `AfficheMessage` est une méthode de la classe `Fonctions.Test1` qui ne renvoie aucune valeur
- `ChercheEtTrouve` est une méthode de la classe `Fonctions.Test1` qui renvoie une valeur booléenne.

Une fonction ou méthode est définie par

- Un nom
- Une liste de paramètres entre parenthèses
- Des instructions codées entre accolades, formant le corps de la méthode,

Exemple :

```
static void AfficheMessage ()
{
    // Corps de la méthode
}
```

10.2. Utilisation des fonctions

Après avoir été définie, une fonction peut être appelée :

- A partir de la même classe

```
static void Main()
{
    Console.WriteLine("Programme principal");
    // ....
    AfficheMessage();
    // ....
    AfficheMessage();
    // ....
    bool btrouve = ChercheEtTrouve()
}
```

- A partir d'une autre classe

Le nom de la méthode doit être précédé du nom de la classe qui contient la méthode qui aura été déclarée avec le mot clé **public**.

Une méthode non déclarée **public** est une méthode privée (**private**) pour la classe ; cette méthode ne pourra être appelée que de la classe où elle est définie.

```
static void AfficheMessage ()
private static void AfficheMessage ()
```

Les 2 notations sont équivalentes.

```
namespace Fonctions
{
    class Test1
    {
        // programme principal
        static void Main()
        {
            Console.WriteLine("Programme principal");
            // ....
            Test2.AfficheMessage();
            // ....
            Test2.AfficheMessage();
        }
    }
    class Test2
    {
        // définition de la fonction
        public static void AfficheMessage()
        {
            Console.WriteLine("Ceci est un message");
        }
    }
}
```

```
}
```

- A partir d'une autre méthode (méthodes imbriquées)

Il est possible également d'appeler une méthode à partir d'une autre méthode.

```
namespace Fonctions
{
    class Test1
    {

        // définition des fonctions
        static void AfficheMessage()
        {
            Console.WriteLine("Ceci est un message");
        }
        static bool ChercheEtTrouve()
        {
            bool btrouve;
            // .....
            AfficheMessage();
            // .....
            return btrouve;
        }

        // programme principal
        static void Main()
        {
            Console.WriteLine("Programme principal");
            // .....
            AfficheMessage();
            // .....
            AfficheMessage();
            // .....
            bool btrouve = ChercheEtTrouve()
        }
    }
}
```

Affichemessage est appelée

10.3. Le corps de méthode

10.3.1. Les variables

Des variables peuvent être déclarées dans le corps de méthode : elles sont locales, créées au début de la méthode et détruites à la sortie

```
static void ChercheEtTrouve ()
{
    bool btrouve = false;
    // .....
}
```

Pour partager une des informations entre méthodes, on peut utiliser une variable de classe.

```
namespace Fonctions
{
    class Test1
```

```
{
    // définition de la variable de classe
    static string strMess;
    // définition les fonctions
    static void EtablirMessage()
    {
        strMess = "Ceci est un message";
    }

    static void AfficherMessage()
    {
        Console.WriteLine(strMess);
    }
    // programme principal
    static void Main()
    {
        Console.WriteLine("Programme principal");
        // ....
        EtablirMessage();
        // ....
        AfficherMessage();
        // ....
    }
}
```

10.3.2. L'instruction return

L'instruction **return** dans une fonction permet le retour immédiat à l'appelant. Dans le cas de l'exemple précédent :

```
static void Main()
{
    Console.WriteLine("Programme principal");
    // ....
    EtablirMessage();
    return;
    AfficherMessage();
    // ....
}
```

L'ajout de l'instruction **return** entre l'appel des 2 fonctions provoque un avertissement du compilateur « impossible d'atteindre le code détecté » ; le programme n'exécute jamais la fonction `AfficherMessage`.

Il est plus courant d'utiliser l'instruction **return** dans une expression conditionnelle : le retour au niveau supérieur se fait que si la condition est vraie.

Dans le cas de méthode renvoyant un résultat, l'instruction **return** suivie d'une expression termine la méthode immédiatement en retournant l'expression comme valeur de retour de la méthode.

```
static bool ChercheEtTrouve ()
{
    bool btrouve = false;
    // ....
    return btrouve;
}
```

L'instruction **return** est obligatoire dans ce cas.

10.4. Le passage de paramètres

Les paramètres permettent de passer des informations à l'intérieur et à l'extérieur d'une méthode.

10.4.1. . Déclaration des paramètres

Chaque paramètre se caractérise par un type et un nom. Chaque paramètre est séparé de l'autre par une virgule.

```
static void AfficheMessage(string strMess, int nbfois)
{
    for (int i = 1; i <= nbfois; i++)
    {
        Console.WriteLine(strMess);
    }
}
```

10.4.2. . Appel de la méthode

Le code appelant doit fournir les valeurs des paramètres lors de l'appel de la méthode, dans l'ordre de définition.

Dans le cas de l'exemple précédent :

```
static void Main()
{
    Console.WriteLine("Programme principal");
    // ....
    string strMess = "Ceci le premier message affiché 2 fois";
    AfficheMessage(strMess, 2);
    // ....
    strMess = "Ceci est le 2eme message";
    AfficheMessage(strMess, 1);
    // ....
}
```

C# utilise par défaut le mécanisme de passage des paramètres par valeur : les données sont transférées de l'extérieur de la méthode vers l'intérieur : ce sont des **paramètres d'entrée**.

10.5. Méthode de passage de paramètres

Il existe trois façons de passer des paramètres :

- Par valeur (paramètre d'entrée)
- Par référence (paramètre d'entrée-sortie)
- Par paramètre de sortie

10.5.1. Par valeur (paramètres d'entrée)

Les données sont transférées de l'extérieur vers l'intérieur de la méthode.

La valeur du paramètre est copiée : la variable peut être modifiée à l'intérieur de la méthode sans influence sur sa valeur à l'extérieur.

Langage C# Partie 1

```
namespace Fonctions
{
    class Test2
    {
        // définition de la fonction
        static void Incrmente(int n)
        {
            n++ ;
            Console.WriteLine("n= {0}",n); // Affiche 2
        }

        // programme principal
        static void Main()
        {
            Console.WriteLine("Programme principal");
            // ....
            int x = 1;
            Incrmente(x);
            Console.WriteLine("x= {0}", x); // Affiche 1

            Console.ReadLine();
        }
    }
}
```

Le paramètre doit être initialisé avant l'appel de la fonction ; Après l'appel de la fonction, la valeur affichée est toujours 1 : Il n'y a pas de retour de la valeur de la fonction vers l'appelant.

10.5.2. Par référence (paramètres d'entrée/sortie)

Les données peuvent être transférées de l'extérieur vers l'intérieur, puis vers l'extérieur de la méthode.

On utilise le mot clé **ref** par paramètre, pour spécifier un appel par référence, dans la méthode et lors de son appel.

```
namespace Fonctions
{
    class Test2
    {
        // définition de la fonction
        static void Incrmente (ref int n)
        {
            n++ ;
            Console.WriteLine("n= {0}",n); // Affiche 3
        }

        // programme principal
        static void Main()
        {
            Console.WriteLine("Programme principal");
            // ....
            int x = 1;
            Incrmente (ref x);
            Console.WriteLine("x= {0}", x); // Affiche 2

            Console.ReadLine();
        }
    }
}
```

Le paramètre doit être initialisé avant l'appel de la fonction ; Après l'appel de la fonction, la valeur affichée est maintenant 2 : Il y a retour de la valeur de la fonction vers l'appelant.

10.5.3. Paramètres de sortie

Les données ne peuvent être transférées que de l'intérieur vers l'extérieur de la méthode.

On utilise le mot clé **out** par paramètre, pour spécifier un appel avec paramètre de sortie, dans la méthode et lors de son appel.

```
namespace Fonctions
{
    class Test3
    {
        // définition de la fonction
        static void Incremente (out int n)
        {
            n=3 ;
            Console.WriteLine("n= {0}",n); // Affiche 2
        }

        // programme principal
        static void Main()
        {
            Console.WriteLine("Programme principal");
            // ....
            int x ;
            Incremente (out x);
            Console.WriteLine("x= {0}", x); // Affiche 3

            Console.ReadLine();
        }
    }
}
```

Dans la fonction, le paramètre doit être initialisé avant d'être utilisé ; Après l'appel de la fonction, la valeur affichée est 3 : Il y a retour de la valeur de la fonction vers l'appelant.

10.6. La récursivité

Le langage C# permet les fonctions récursives. C'est-à-dire que l'une des instructions d'une fonction peut être un appel à la fonction elle-même. C'est très pratique pour coder certains algorithmes comme la factorielle :

$\text{factorielle}(n) = n * \text{factorielle}(n - 1)$

ou l'algorithme d'Euclide :

si $n_2 > n_1$, $\text{pgcd}(n_1, n_2) = \text{pgcd}(n_1, n_2 - n_1)$

Ce principe est basé sur une notion mathématique : la *réurrence* :

Pour démontrer qu'une propriété est vraie quelle que soit la valeur de n , on démontre que :

- La propriété est vraie pour $n=1$.
- Si la propriété est vraie pour $n-1$, elle est vraie pour n .

Ainsi, si les deux théorèmes précédents sont démontrés, on saura que la propriété est vraie pour $n=1$ (1er théorème). Si elle est vraie pour $n=1$ elle est vraie pour $n=2$ (2ème théorème). Si elle est vraie pour $n=2$, elle est vraie pour $n=3$... Et ainsi de suite.

La création d'une fonction récursive risque d'engendrer un phénomène sans fin. C'est pourquoi, on prévoira toujours une fin de récursivité. Cette fin correspond en fait au codage du premier théorème :

```
static long SommeDesNombres(long n)
{
    if (n == 1)
    {
        return 1; // 1er théorème
    }
    else
    {
        return n + SommeDesNombres(n - 1); // 2ème théorème
    }
}
```

10.7. La surcharge des méthodes

Le langage C# permet que deux fonctions différentes aient le même nom à condition que les paramètres de celles-ci soient différents soit dans leur nombre, soit dans leur type, soit dans le mode de passage du paramètre (out ou ref).

Par exemple, les trois fonctions prototypées ci-dessous sont trois fonctions différentes. Cette propriété du langage C# s'appelle la *surcharge*.

```
double maFonction(double par1);
double maFonction(double par1, double par2);
double maFonction(int par1);
```

Contrairement aux autres langages de programmation, ce qui permet au compilateur d'identifier et de distinguer les sous-programmes, ce n'est pas le nom seul de la fonction, mais c'est la **signature**. Deux fonctions sont identiques si elles ont la même signature, c'est-à-dire le même **nom**, le même **nombre de paramètres** de même **type** et de même **mode** de passage.

Le nom du paramètre et le type de retour de la fonction n'affectent pas la signature par exemple, ces 2 fonctions sont identiques, et ne pourront donc pas être déclarées dans la même classe.

```
double maFonction(double par1);
int maFonction(double parx);
```

Exemple :

```
class Exemple
{
    static int Add(int a, int b)
    {
        return a + b;
    }
    static int Add(int a, int b, int c)
    {
        return Add(a,b)+ c;
    }
    static void Main()
    {
        Console.WriteLine(Add(1, 2) + Add(1, 2, 3));
    }
}
```

La méthode **Add** a été surchargée pour pouvoir traiter l'addition de 3 entiers. On notera que dans le corps de la méthode surchargée, on utilise la méthode initiale, pour éviter la redondance de code.

Il est évident que l'abus de cette possibilité peut amener à écrire des programmes difficiles à maintenir.

Cependant, cela permet de donner le même nom à des fonctions qui jouent le même rôle mais dont le nombre et le type de paramètres doivent être différents. C'est le cas, par exemple, de la fonction **IndexOf** de la classe **String** qui peut recevoir 1, 2 voire 3 paramètres.

Dans tous les cas, le premier paramètre est la portion de chaîne (ou le caractère) que l'on veut repérer dans la chaîne globale (voir paragraphe II.3.b du présent support).

11. Les tableaux

11.1. Définition

Imaginons que dans un programme, nous ayons besoin simultanément de 12 valeurs (par exemple, des notes pour calculer une moyenne). Evidemment, la seule solution dont nous disposons à l'heure actuelle consiste à déclarer douze variables, appelées par exemple Notea, Noteb, Notec, etc.

Bien sûr, on peut opter pour une notation un peu simplifiée, par exemple N1, N2, N3, etc. Mais cela ne change pas fondamentalement notre problème, car arrivé au calcul, et après une succession de douze instructions de lecture distinctes, cela donnera obligatoirement quelque chose comme :

Moy = (N1 + N2 + N3 + N4 + N5 + N6 + N7 + N8 + N9 + N10 + N11 + N12) / 12

Ce qui est laborieux. Et pour un peu que nous soyons dans un programme de gestion avec quelques centaines ou quelques milliers de valeurs à traiter, cela se révèle fortement problématique.

Si, de plus, on est dans une situation où l'on ne peut pas savoir d'avance combien il y aura de valeurs à traiter, on se retrouve là face à un mur.

C'est pourquoi la programmation nous permet de rassembler toutes ces variables en une seule, au sein de laquelle chaque valeur sera désignée par un numéro. En bon français, cela donnerait donc quelque chose du genre « la note numéro 1 », « la note numéro 2 », « la note numéro 8 ».

Un tableau est une suite d'éléments de même type. On peut accéder à un élément d'un tableau en utilisant sa position : l'index.

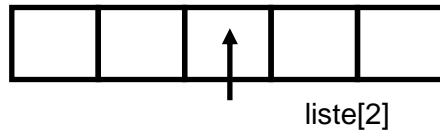
En C#, le premier élément se trouve à l'index 0.

11.2. Déclaration et création

Un tableau est déclaré en spécifiant son type, sa dimension (ou rang) et son nom.

```
type [ ] nom ;
```

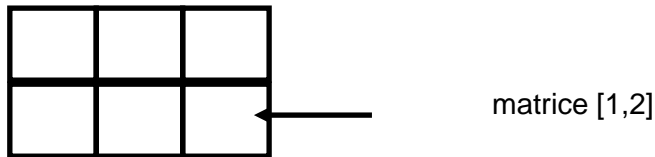
```
int [ ] liste; // tableau à une dimension
```



liste[2] désignera le 3ème élément du tableau liste.

Un seul index est nécessaire pour repérer un élément de tableau.

```
string [, ] matrice; // tableau à deux dimensions
```



matrice [1,2] désignera l'élément situé sur la 2ème ligne, 3ème colonne du tableau matrice.

Deux index sont nécessaires pour repérer un élément de tableau, l'indice des lignes et l'indice des colonnes

La déclaration d'une variable tableau n'entraîne pas sa création: il faut obligatoirement utiliser new pour créer explicitement l'instance du tableau et spécifier la taille de toutes les dimensions.

```
int[] liste; // déclaration
liste = new int[5]; // création d'une instance
```

que l'on peut résumer en :

```
int[] liste = new int[5]; // création d'une instance
int [, ] matrice = new string [2,3] ;
```

Le compilateur C# initialise implicitement chaque élément du tableau à sa valeur par défaut (en fonction du type) que l'on peut initialiser comme suit :

```
int[] liste = new int[5]{4,2,3,1,5};
int [, ] matrice = new int[2,3] {
    {1,2,3},
    {10,20,30}};
```

Chaque élément doit obligatoirement être initialisé.

La taille d'un tableau n'est pas nécessairement une constante; elle peut également être spécifiée par une valeur entière au moment de la compilation.

```
int taille = int.Parse(Console.ReadLine()); // lecture
int []liste= new int [taille]; // création
int [,]matrice= new int [2,taille]; // création
```

En C#, lors d'une tentative d'accès au tableau, l'index est automatiquement contrôlé de manière à garantir sa validité. Un index hors limite – inférieur à 0 ou supérieur ou égal à sa taille- renvoie une exception **IndexOutOfRangeException**.

11.3. Utilisation

Soit 2 tableaux ainsi déclarés :

```
int[] liste = new int[5] {4,2,3,1,5}; // création d'une instance
string[,] matrice= new string [2,3] {
                                {1,2,3},
                                {10,20,30}};
```

11.3.1. Propriétés

La propriété **Length** renvoie le nombre total de postes d'un tableau.

liste.Length rend la valeur 5

matrice.Length rend la valeur 6 (2 * 3)

La propriété Rank renvoie le nombre de dimensions du tableau

liste.Rank rend la valeur 1

matrice.Rank rend la valeur 2.

11.3.2. Méthodes

La méthode **Sort** permet de trier un tableau.

System.Array.Sort(liste); rend dans liste un tableau trié.

La méthode **Clear** permet de réinitialiser un tableau à sa valeur par défaut.

```
// initialise p éléments à partir de la position i
System.Array.Clear(liste, i, p);
```

La méthode **GetLength** permet de renvoyer la longueur d'une dimension.

```
matrice.GetLength(0) ; //rend la valeur 2.
matrice.GetLength(1) ; //rend la valeur 3.
```

La méthode **Clone** crée une nouvelle instance de tableau dont les éléments sont des copies des éléments du tableau cloné.

```
int[] listeCopie = (int [])liste.Clone();
```

alors que l'instruction suivante ne copie pas l'instance du tableau, mais se contente de référencer la même instance de tableau.

```
int[] listeCopie = liste ;
```

11.3.3. Utilisations particulières

Un tableau peut être paramètre de retour de méthode, ou passé en paramètre dans une méthode.

```
namespace Tableaux
{
    class Test1
    {
        static void Main()
        {
            // création nouveau tableau
            int[] nouveauT = CreeTableau(5);
            Console.WriteLine("Nb de postes ={0}", nouveauT.Length);
            // Affichage du tableau
            AfficheTableau(nouveauT);
            //
            Console.ReadLine();
        }
    }
}
```

Tableau en retour de méthode

```
static int[] CreeTableau(int taille)
```

```
{
    return new int[taille];
}
```

Tableau passé en paramètre de méthode

```
static void AfficheTableau(int [] unTableau )
```

```
{
    for (int i = 0; i < unTableau.Length; i++)
    {
        Console.WriteLine("Poste {0}: {1}", i, unTableau[i]);
    }
}
```

C'est ainsi que Main peut accepter un tableau de chaînes comme paramètre, réceptionnant les arguments de la ligne de commande.

```
static void Main(string [] args)
{
    // ....
}
```

11.3.4. L'instruction foreach

L'instruction foreach simplifie le parcours des éléments d'un tableau.

```
foreach (int entier in unTableau)
{
    Console.WriteLine(entier);
}
```

Les éléments d'index (initialisation, contrôle et incrémentation) sont inutiles.

11.3.5. Opérations de tri sur un tableau

Sur une structure de données de type tableau, il est possible de faire plusieurs types de traitement, comme par exemple la recherche du minimum ou du maximum, la somme ou le produit ou la moyenne des postes du tableau.

On peut également vouloir rechercher un élément donné dans un tableau.

Par exemple, sur un tableau de 35 postes numériques, on désire calculer la somme des postes, et rechercher le plus petit élément.

Pour le calcul de la somme, on ajoutera le contenu des cases une à une, depuis la première jusqu'à la trente cinquième.

Pour la recherche du minimum :

- On va supposer que la première case contient le minimum relatif
- On va comparer le contenu de la deuxième case avec le minimum relatif : si celui-ci est inférieur, il deviendra le minimum relatif.
- On recommencera l'opération avec les postes restants

Un tableau est ordonné lorsqu'il existe une relation d'ordre entre les différentes cases : On parle de :

- tri croissant si le contenu de la case d'indice i est inférieur ou égal au contenu de la case d'indice $i + 1$
- tri décroissant si le contenu de la case d'indice i est supérieur ou égal au contenu de la case d'indice $i + 1$

11.3.6. Recherche d'un élément sur un tableau trié

Une première manière de vérifier si un mot se trouve dans le dictionnaire consiste à examiner successivement tous les mots du dictionnaire, du premier au dernier, et à les comparer avec le mot à vérifier.

Ca marche, mais cela risque d'être long : si le mot ne se trouve pas dans le dictionnaire, le programme ne le saura qu'après 40 000 tours de boucle ! Et même si le mot figure dans le dictionnaire, la réponse exigera tout de même en moyenne 20000 tours de boucle. C'est beaucoup, même pour un ordinateur.

Or, il y a une autre manière de chercher, bien plus intelligente pourrait-on dire, et qui met à profit le fait que dans un dictionnaire, les mots sont triés par ordre alphabétique. D'ailleurs, un être humain qui cherche un mot dans le dictionnaire ne lit jamais tous les mots, du premier au dernier : il utilise lui aussi le fait que les mots soient triés.

Pour une machine, quelle est la manière la plus rationnelle de chercher dans un dictionnaire ? C'est de comparer le mot à vérifier avec le mot qui se trouve pile poil au milieu du dictionnaire. Si le mot à vérifier est antérieur dans l'ordre alphabétique, on sait qu'on devra le chercher dorénavant dans la première moitié du dico. Sinon, on sait maintenant qu'on devra le chercher dans la deuxième moitié.

A partir de là, on prend la moitié de dictionnaire qui nous reste, et on recommence : on compare le mot à chercher avec celui qui se trouve au milieu du morceau de dictionnaire restant. On écarte la mauvaise moitié, et on recommence, et ainsi de suite.

A force de couper notre dictionnaire en deux, puis encore en deux, etc. on va finir par se retrouver avec des morceaux qui ne contiennent plus qu'un seul mot. Et si on n'est pas tombé sur le bon mot à un moment ou à un autre, c'est que le mot à vérifier ne fait pas partie du dictionnaire.

Regardons ce que cela donne en termes de nombre d'opérations à effectuer, en choisissant le pire cas : celui où le mot est absent du dictionnaire :

- Au départ, on cherche le mot parmi 40 000.
- Après le test n°1, on ne le cherche plus que parmi 20 000.
- Après le test n°2, on ne le cherche plus que parmi 10 000.
- Après le test n°3, on ne le cherche plus que parmi 5 000.
- etc.
- Après le test n°15, on ne le cherche plus que parmi 2.
- Après le test n°16, on ne le cherche plus que parmi 1.

Et là, on sait que le mot n'existe pas. Moralité : on a obtenu notre réponse en 16 opérations contre 40 000 précédemment ! Il n'y a pas photo sur l'écart de performances entre la technique barbare et la technique fûtée. Attention, toutefois, même si c'est évident, **la recherche dichotomique ne peut s'effectuer que sur des éléments préalablement triés.**

12. Autres types de données

12.1. Le type enum

Les énumérateurs sont utiles lorsqu'une variable ne peut prendre qu'un ensemble spécifique de valeurs. Leur but est de rendre le code source plus compréhensible.

Un type énuméré est un entier défini par le programmeur à l'aide du mot `enum` ; il s'agit d'une liste de constantes symboliques qui possèdent un identificateur et formant un ensemble de valeurs appartenant à un nouveau type.

Définition d'un type **enum** :

```
enum EtatCivil {Celibataire, Marie, Divorcé, Veuf} ;
```

12.1.1. Utilisation d'un type enum :

On pourra déclarer une variable `ec` de type `EtatCivil`, en utilisant la syntaxe suivante :

```
EtatCivil ec ;
```

On pourra l'affecter en codant :

```
ec = EtatCivil.Marie ;
```

Ou

```
ec = (EtatCivil)1; // cast d'un type int en EtatCivil
```

On pourra l'afficher en codant :

```
Console.WriteLine("Statut :{0}",ec);
```

Par défaut, le compilateur associe la valeur 0 à célibataire, la valeur 1 à marié, la valeur 2 à divorcé et la valeur 3 à Veuf.

La déclaration peut être placée dans la classe, mais en dehors d'une fonction, ou le plus souvent accessible depuis tout point et donc en dehors de la classe.

Nous reviendrons sur ces principes de visibilité et portée.

Nous pouvons aussi définir des valeurs spécifiques pour chaque valeur de l'énumération :

```
enum EtatCivil
{
    Celibataire=10,
    Marie =20,
    Divorcé= Marie + 1,
    Veuf= 40
}
```

Il est cependant prudent de prévoir une valeur zéro pour la plus commune des valeurs pour des problèmes d'initialisation.

On peut également convertir une chaîne de caractères en l'une des valeurs possibles de l'énumération :

```
string strStatut = "Veuf";
ec = (EtatCivil)Enum.Parse(typeof(EtatCivil), strStatut, false);
```

Le 3eme argument signale que la casse est ignorée.

Lorsque le 2eme argument ne correspond à aucune des valeurs possibles, une erreur de type `ArgumentException` est générée.

12.2. Le type struct

Dans sa forme la plus simple, une structure comprend plusieurs informations regroupées dans une même entité.

Les objets créés à partir de structures se comportent comme des types prédéfinis.

Exemple : Simulation d'un point en géométrie

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

La structure contient 2 membres de type int spécifiant la position horizontale et verticale d'un point.

12.2.1. Utilisation d'un type struct :

On pourra déclarer une variable p de type Point, en utilisant la syntaxe suivante :
Point point ;

Les champs de p seront accessibles par point.x et point.y .et ne sont pas initialisés.

Exemple:

```
static void Main()
{
    Point point;
    point.x = 0; Point.y = 0;
    Console.WriteLine("Coordonnées : {0},{1}",point.x, point.y);
}
```

12.2.2. Les différences entre classe et structure.

Structures et classes sont syntaxiquement similaires mais il existe des différences importantes.

Nous reviendrons sur ces différences après avoir découvert la programmation objet.
Une structure est notamment un type valeur alors qu'une classe est un type référence.

12.3. Dates et heures

Le type valeur **DateTime** représente des dates et des heures dont la valeur est comprise entre 12:00:00 (minuit), le 1er janvier de l'année 0001 de l'ère commune et 11:59:59 (onze heures du soir), le 31 décembre de l'année 9999 après J.C. (ère commune).

Il permet d'effectuer diverses opérations sur les dates : déterminer le jour de la semaine, déterminer si une date est inférieure à une autre, jouter un nombre de jours à une date.

12.3.1. Déclaration d'une date :

```
// crée un objet DateTime
DateTime d1= new DateTime();
//crée un objet DateTime pour le 01/01/2008
DateTime d2 = new DateTime(2008, 01, 01);
// crée un objet DateTime pour le 01/01/2008 12h 0mn 0s 0 ms
DateTime d3 = new DateTime(2008, 01, 01, 12, 0, 0, 0);
```

12.3.2. Utilisation d'un type DateTime :

La structure DateTime possède quelques propriétés intéressantes :

Date permet de rendre un nouveau DateTime avec la partie heure initialisée à 0

```
Console.WriteLine(d3.Date); // rend 2008/01/01 à 0h
```

Day, Month, Year permettent de rendre respectivement jour, mois année.

Hour, Minute, Second, Millisecond permettent de rendre respectivement heures, minutes, secondes et millisecondes.

```
Console.WriteLine("{0},{1},{2}", d2.Day, d2.Month, d2.Year);
// rend 1,1,2008
```

DayOfWeek permet de rendre le jour de la semaine

```
Console.WriteLine("{0}", d2.DayOfWeek); // rend Tuesday
```

DayOfYear permet de rendre le quantième correspondant à la date.

```
Console.WriteLine("{0}", d2.DayOfYear); // rend 1
```

Now donne la date et l'heure du jour

```
// crée un objet DateTime initialisé avec date et heure du
jour
DateTime d1 = DateTime.Now;
```

Today donne la date du jour : l'heure est initialisée à 0

```
// crée un objet DateTime initialisé avec date du jour
DateTime d1 = DateTime.Today;
```

Et possède également un certain nombre de méthodes :

AddDays, AddMonths, AddYears permettent d'ajouter respectivement jour, mois, et année.

AddHours, AddMinutes, AddSeconds, AddMilliseconds permettent d'ajouter respectivement heures, minutes, secondes et millisecondes.

Compare, CompareTo permettent de comparer deux dates

```
int res1 = DateTime.Compare(d2, d3);
int res2 = d2.CompareTo(d3);
et renvoient 0 si d2=d3, 1 si d2> d3, -1 sinon.
```

Les opérateurs de comparaison peuvent également être employés.

ToString permet de mettre en forme une date. En utilisant des formats généraux

```
// Format date longue
Console.WriteLine(d2.ToString("D")); // mardi 1 janvier 2008
// Format date courte
Console.WriteLine(d2.ToString("d")); // 01/01/2008
// Jour + Mois en clair
Console.WriteLine(d2.ToString("M")); // 1 janvier
```

En utilisant des formats personnalisés, par exemple :

d, M	Jour (0 à 31), mois (01 à 12)
dd,MM	Jour (01 et 31), mois (01 à 12)
ddd, MMM	Abréviation du jour/ mois
dddd, MMMM	Nom complet du jour / mois
y	Année (de 1 à 99)
yy	Année (de 01 à 99)
yyyy	Année (de 1 à 9999)

```
Console.WriteLine(d2.ToString("d-MMMM-yy")); // 1-janvier-08
```

(Voir l'aide en ligne pour les autres méthodes, et tous les formats).

12.3.3. Utilisation de TimeSpan :

Les types valeur `DateTime` et `TimeSpan` se distinguent par le fait que `DateTime` représente un instant, tandis que `TimeSpan` représente un intervalle de temps. Cela signifie, par exemple, que vous pouvez soustraire une instance `DateTime` d'une autre pour obtenir l'intervalle de temps les séparant. De la même façon, vous pouvez ajouter un `TimeSpan` positif au `DateTime` en cours pour calculer une date ultérieure.

Vous pouvez ajouter ou soustraire un intervalle de temps d'un objet `DateTime`. Les intervalles de temps peuvent être négatifs ou positifs ; ils peuvent être exprimés en unités telles que les graduations ou les secondes ou comme un objet `TimeSpan`.

Déclaration d'un `TimeSpan` :

```
// crée un nouveau TimeSpan
TimeSpan ts1 = new TimeSpan();
//crée un nouveau TimeSpan en heure, mn, secondes
TimeSpan ts2 = new TimeSpan(10,20,30);
```

La structure `TimeSpan` possède des propriétés:

`Days`, `Minutes`, `Seconds`, `Milliseconds` permettent de rendre respectivement le nombre de jour, minutes, secondes et millisecondes de l'intervalle de temps.

Et aussi des méthodes :

`Add`, `Subtract` permettent d'ajouter ou de soustraire un intervalle de temps passé en argument.

Exemple : Calcul du nombre de jours écoulés depuis le début du 21eme siècle

```
DateTime dj = DateTime.Today; // aujourd'hui
DateTime ds = new DateTime(2001, 01, 01); // début du 21eme
siècle
TimeSpan ts = dj - d;
Console.WriteLine("Nb de jours : {0}", ts.Days);
```

13. Le traitement des erreurs

13.1. Les objectifs

Repérer les instructions qui sont susceptibles de générer des exceptions
Capturer des exceptions pour afficher les messages d'erreur adéquats.
Générer des exceptions

13.2. Vue d'ensemble

Il est important de proposer des standards de gestion d'erreurs dans les applications.

Cette approche standardisée aura pour but d'éviter les arrêts brutaux de vos applications liés à des déficiences de prises en compte des erreurs et les pertes de données qui s'en suivraient.

Des erreurs d'exécution peuvent se produire lors de l'exécution d'une instruction : Si aucun traitement particulier n'est effectué, l'application s'arrête brutalement, ce qui en général, n'est guère apprécié des utilisateurs, et peut provoquer des pertes de données.

Il faut donc identifier les parties de programme où des erreurs d'exécution peuvent se produire, et écrire du code spécifique pour les traiter.

Les techniques de traitement d'erreurs permettent en cours d'exécution de programme de détecter et de réagir à :

Des erreurs générées par le système, comme les divisions par 0, l'utilisation d'un index de tableau invalide ou l'absence de correspondance d'une valeur d'une énumération.

Une bonne gestion des erreurs permet aussi d'améliorer la productivité des développeurs et de faciliter la maintenance des applications.

Le modèle retenu est celui qui existait préalablement en C++.

Le gestionnaire d'erreurs utilise des blocs Try...Catch...Finally.

Les classes relatives à la gestion des exceptions se situent dans l'espace de nom `System.Exception`.

La structure de base est de type :

Try : Début de la prise en compte des erreurs

Catch : Interception des erreurs survenues entre Try et Catch

Finally : bloc d'instructions facultatif.

Ce dernier est systématiquement parcouru, qu'une exception ait eu lieu ou non.

Il est possible de définir plusieurs blocs catch pour un seul Try afin de filtrer les erreurs par nature. Les blocs Catch fonctionnent à la manière des groupes case.

Il est possible d'imbriquer les structures Try.

L'instruction `Throw` permet de générer une exception nouvelle. Attention à son utilisation toutefois car elle demande pas mal de ressources au système. Vous avez d'ailleurs certainement pu le constater lors des plantages de vos programmes ...

Un des principaux intérêts de cette approche réside dans la centralisation possible des procédures prenant en charge la gestion des exceptions.

Il est ainsi possible de les isoler dans une classe particulière et de rendre ces procédures accessibles depuis tout point de votre application.

Des erreurs générées par une fonction du programme, par exemple une valeur négative ou trop élevée pour l'âge d'une personne.

Exemple :

```
static void Main(string[] args)
{
    int a = 10, b = 0, c;
    Console.WriteLine("La division");
    c= a/b;
    Console.WriteLine(c);
}
```

Le programme est interrompu par une erreur,

L'exception **DivideByZeroException** n'a pas été gérée.

Le programme s'arrête brutalement, et le deuxième affichage n'est jamais exécuté.

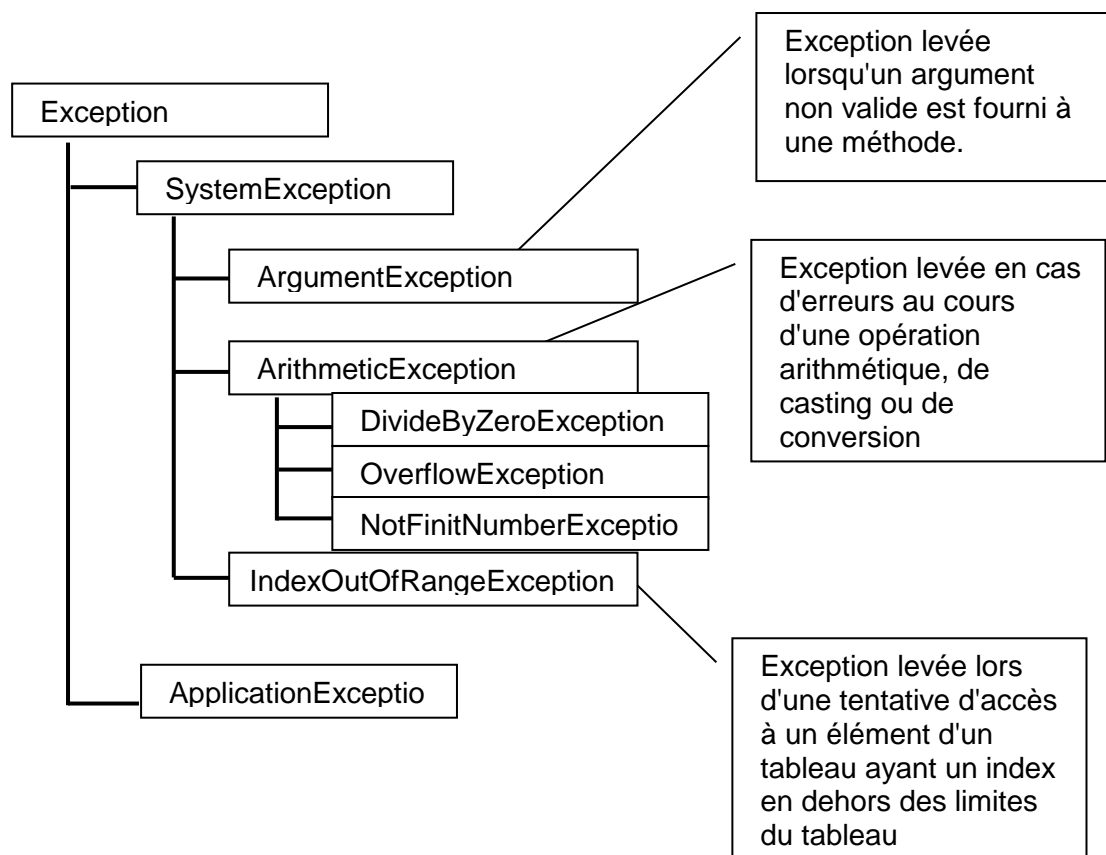
13.3. Les objets Exception

Le framework définit une hiérarchie de classes d'exception, dont la classe de base est la classe **Exception**.

Il existe deux catégories d'exceptions sous la classe de base Exception :

- les classes d'exceptions prédéfinies du Common Language Runtime, dérivées de **SystemException**.
- les classes d'exceptions de l'application définies par l'utilisateur, dérivées de **ApplicationException**.

Chaque classe d'exception décrit clairement l'erreur qu'elle représente



13.4. Les clauses try et catch

L'idée est de séparer les instructions essentielles du programme, des instructions de gestion d'erreur.

Les parties de code susceptibles de lever des exceptions sont placées dans un bloc **try**, et le code de traitement de ces exceptions est placé dans un bloc **catch**.

A l'intérieur du groupe **try**, les instructions sont écrites, sans se soucier du traitement d'erreurs.

Dans notre exemple :

```
static void Main(string[] args)
{
    int a = 10, b = 0, c;
    try
    {
        Console.WriteLine("La division");
        c = a/b;
        Console.WriteLine(c);
    }
    catch (Exception)
    {
        Console.WriteLine("Erreur arithmétique");
    }
}
```

Le programme rentre dans le bloc try, comme si l'instruction try n'existait pas, et les instructions sont exécutées en séquence.

Dès qu'une erreur est détectée par le système, un objet de la classe Exception, ou classe dérivée est automatiquement créé, et le programme se débranche à la première instruction du bloc catch correspondant.

A l'intérieur des parenthèses de la clause catch, un nom d'objet est souvent spécifié, de manière à obtenir des informations supplémentaires au sujet de l'erreur, par exemple le champ Message de la classe Exception, qui délivre en clair le message de l'erreur.

```
catch (Exception e)
{
    Console.WriteLine("Erreur arithmétique \n" + e.Message);
}
```

Le message affiché sera :

Erreur arithmétique

Tentative de division par zéro.

Nous nous sommes servis de la classe générale d'exception pour traiter l'erreur ; de ce fait, toutes les erreurs détectées par n'importe quelle instruction afficheront ce même message... ce qui paraît absurde !

Il est possible, et recommandé, d'enchaîner les blocs catch chacun correspondant à un type d'exception traité.

L'exemple page suivante illustre ce principe.

Langage C# Partie 1

```
static void Main(string[] args)
{
    int a = 10, b = 0, c;
    try
    {
        Console.WriteLine("La division");
        c= a/b;
        Console.WriteLine(c);
    }
    catch (ArithmeticException ae)
    {
        Console.WriteLine("Erreur arithmétique\n" + ae.Message);
    }
    catch (Exception e)
    {
        Console.WriteLine("Erreur générale\n" + e.Message);
    }
}
```

De ce fait, si aucun bloc catch spécifique n'existe, l'exception est interceptée par un bloc catch général, le cas échéant.

Il faut veiller à toujours classer les exceptions dans les blocs catch de la plus spécifique à la plus générale.

Cette technique permet de gérer l'exception spécifique avant qu'elle ne passe à un bloc catch plus général.

Attention : vous vous apercevrez rapidement que la mise en place d'une gestion d'erreurs affecte les performances de manière considérable Réservez ces traitements aux erreurs imprévisibles (vous préférerez ainsi contrôler préventivement la saisie d'un nombre en mémoire plutôt que de mettre en place un bloc try ... catch).

13.5. Le groupe finally

C# fournit la clause finally pour entourer les instructions qui doivent être exécutées, quoi qu'il arrive : ces instructions seront donc exécutées, si le programme se termine normalement, à l'issue du bloc try, ou anormalement, à l'issue d'un catch.

Le bloc finally est utile dans deux cas :

- éviter la duplication d'instructions
- des instructions devant s'exécuter à la fois en cas de fin normale et anormale trouveront leur place dans un bloc finally
- libérer des ressources après la levée d'une exception.

De manière générale, et sur un exemple :

```
using System;
...
try
{
    // Séquence d'instructions correspondant à la saisie de
    // données
    // numériques, à la gestion de tableau et à l'exécution
    // d'opérations arithmétiques.
}
catch (FormatException fe)
{
    Console.WriteLine("Erreur de saisie \n" + fe.Message);
}
```



```

}
catch (ArithmeticException ae)
{
    Console.WriteLine("Erreur arithmétique \n" + ae.Message);
}
catch (IndexOutOfRangeException ioore)
{
    Console.WriteLine("Index de tableau non valide \n" +
        ioore.Message);
}
catch (Exception e)
{
    // Ce message est affiché pour toutes les autres
    // exceptions susceptibles d'être générées.
    Console.WriteLine("Erreur générale \n" + e.Message);
}
finally
{
    // Instructions à exécuter quelle que soit l'issue
    // du programme
}

```

13.6. L'instruction throw

Dans le traitement d'une fonction, il peut être intéressant de générer une exception. Par exemple, dans l'utilisation d'un tableau, on peut générer exception lorsque l'index champ est inférieur à 1 ou supérieur au nombre de postes contenus dans le tableau.

Pour générer une exception, on utilise le mot-clé **throw** :

```

Console.WriteLine("Entrez un chiffre compris entre 1 et 9:");
if (i < 0 || i >= 9)
    throw new IndexOutOfRangeException(i + "n'est pas un choix
valide");

```

L'exécution séquentielle normale du programme est interrompue, et le contrôle d'exécution est transféré sur le premier bloc catch capable de gérer l'exception, en fonction de sa classe.

Les instructions qui suivent le **throw** ne seront jamais exécutées.

Le choix de l'exception dépend bien sûr du type de problème à traiter.

14. Les fichiers

14.1. Les objectifs

Créer, accéder et mettre à jour un fichier texte.
Gérer les fichiers sur disque.

14.2. Vue d'ensemble

Un fichier est une collection ordonnée et nommée d'une séquence particulière d'octets ayant un stockage persistant.

Bien que l'emploi des bases de données se soit généralisé, il est encore nécessaire de savoir accéder aux fichiers « classiques », dits fichiers « plats » (fichiers à organisations séquentielle et relative). Le cas se présente, par exemple, pour récupérer des anciens fichiers ou bien pour assurer l'interface entre deux applications.

Un fichier séquentiel est typiquement un fichier Texte, constitué de lignes de texte séparées par des paires de caractères Chr(13) + Chr(10) (Retour chariot - changement de ligne ou CR-LF) qui indiquent un changement de ligne.

L'accès aléatoire (ou accès direct) à un fichier n'étant pas pris en charge au niveau du Framework nous ne verrons pas ce type d'accès.

Les techniques de manipulation de flux binaires seront revues lors des techniques de sérialisation des objets.

Les classes du Framework relatives à la manipulation de fichiers de l'espace de nom System.IO

L'utilisation d'un fichier dans un programme comportera trois phases :

1. Ouverture du fichier avant la première utilisation ;

Lors de l'ouverture, on spécifiera le mode d'utilisation du fichier :

- Si on ouvre un fichier **pour lecture**, on ne pourra que récupérer les informations qu'il contient, sans les modifier en aucune manière.
- Si on ouvre un fichier **pour écriture**, on pourra mettre dedans toutes les informations que l'on veut. Mais les informations précédentes, si elles existent, seront **intégralement écrasées**.
- Si on ouvre un fichier **pour ajout**, on ne peut ni lire, ni modifier les informations existantes. Mais on pourra ajouter de nouveaux **enregistrements**)

2. Traitement du fichier (Effacement total, positionnement selon un index, lecture, écriture, re-écriture ...)

- Une instruction de lecture copie du disque vers la mémoire, les informations d'un enregistrement.
- Une instruction d'écriture copie les informations en mémoire sur une place libre du support.
- Une instruction de réécriture recopie les informations en mémoire à la place des dernières informations lues.
- Une instruction de suppression supprime une ligne du support.

3. Fermeture du fichier après la dernière utilisation

En C#, toutes les entrées/sorties de données sont vues comme des **flux** (ou flot de données). Un flux permet d'effectuer des lecture/écriture d'octets à d'un **fichier**, de la **mémoire**, du **réseau** ou de la **console**.

L'espace de noms **System.IO** contient un certain nombre de classes permettant d'effectuer des opérations sur les fichiers. Ces diverses classes peuvent être regroupées en différentes catégories :

- Les classes **Directory** et **DirectoryInfo** pour manipuler les répertoires (créer un répertoire, en afficher le contenu...),
- Les classes **File** et **FileInfo** qui fournissent des informations sur les fichiers et permettent diverses manipulations (suppression, changement de nom, copie) mais sans permettre de lire ou d'écrire des enregistrements,
- Les classes **Stream** qui permettent de lire et d'écrire dans les fichiers,
- Des classes qui fournissent des informations sur les répertoires.

14.3. Les classes de flux

La classe abstraite de base **Stream** prend en charge la lecture et l'écriture d'octets en modes synchrone et asynchrone.

Toutes les classes qui représentent des flux héritent de la classe **Stream**. La classe **Stream** et ses classes dérivées donnent une vue générique des sources de données et des référentiels, isolant ainsi le programmeur des détails propres au système d'exploitation et aux périphériques sous-jacents.

Les flux impliquent trois opérations fondamentales :

- Il est possible de **lire les flux**. La lecture est le transfert de données d'un flux vers une structure de données tel qu'un tableau d'octets
- Il est possible d'**écrire les flux**. L'écriture désigne le transfert de données d'une source de données vers un flux
- Les flux peuvent prendre en charge la **recherche**. La recherche consiste à envoyer une requête concernant la position actuelle dans un flux et à modifier cette dernière.

Selon la source de données sous-jacente ou le référentiel, les flux peuvent prendre en charge certaines de ces fonctionnalités. Par exemple, **NetworkStreams** ne prend pas en charge la recherche. Les propriétés **CanRead**, **CanWrite** et **CanSeek** de **Stream** et de ses classes dérivées déterminent les opérations prises en charge par les différents flux

Classes de Flux

Classes	Types de flux
Stream	Classe de base abstraite des flux
BufferedStream	Flux bufférisé (données mises en cache)
FileStream	Classe à tout faire pour les fichiers : - Prend en charge les opérations de lecture et d'écriture synchrones et asynchrones. - Prend en charge l'accès aléatoire via la méthode Seek. Comme Stream ne travaille que sur des tableaux d'octets (byte), le développeur a souvent recours à d'autres classes spécialisées sur les traitements de flux : StreamReader/StreamWriter, BinaryReader/BinaryWriter
MemoryStream	Flux non bufférisé, directement accessible en mémoire (Ce flux n'a pas de magasin de sauvegarde et peut servir de mémoire tampon temporaire)
TextReader	Classes de base abstraites de StreamReader/ StringReader
StreamReader	Lecture de fichier texte
StringReader	Lecture de chaînes de caractères
TextWriter	Classes de base abstraites de StreamWriter/ StringWriter
StreamWriter	Ecriture de fichier texte
StringWriter	Ecriture de chaînes de caractères
BinaryReader	Lecture de flux binaires
BinaryWriter	Ecriture de flux binaires

14.4. Utilisation d'un fichier texte

14.4.1. Ouverture d'un fichier

La classe `FileStream` (dérivée de `Stream`) permet d'ouvrir ou de créer un fichier en spécifiant

- le mode d'ouverture (les intentions du programmeur) au moyen de l'énumération **FileMode**
- le mode d'accès (lecture uniquement, écriture uniquement ou lecture/écriture) au moyen de l'énumération **FileAccess**
- le mode de partage (ce que peuvent faire les autres utilisateurs de ce fichier) au moyen de l'énumération **FileShare**

FileMode : énumération des modes d'ouverture de fichier.

Les paramètres **FileMode** vérifient si un fichier est remplacé, créé ou ouvert ou une combinaison de ces actions.

Utilisez **Open** pour ouvrir un fichier existant. Pour ajouter à un fichier, utilisez **Append**. Pour tronquer un fichier ou le créer s'il n'existe pas, utilisez **Create**.

Nom de membre	Description
Append	Ouvre le fichier s'il existe et accède à la fin du fichier, ou crée un nouveau fichier. FileMode.Append peut seulement être utilisé conjointement avec FileAccess.Write . Toute tentative de lecture échoue et lève un ArgumentException .
Create	Spécifie que le système d'exploitation doit créer un fichier. Si le fichier existe, il est remplacé. Cela nécessite FileIOPermissionAccess.Write et FileIOPermissionAccess.Append . System.IO.FileMode.Create équivaut à demander que si le fichier n'existe pas, utilisez CreateNew ; sinon, utilisez Truncate .
CreateNew	Spécifie que le système d'exploitation doit créer un fichier. Cela nécessite FileIOPermissionAccess.Write . Si le fichier existe, un IOException est levé.
Open	Spécifie que le système d'exploitation doit ouvrir un fichier existant. La possibilité d'ouvrir le fichier dépend de la valeur spécifiée par FileAccess . System.IO.FileNotFoundException est levé si ce fichier n'existe pas.
OpenOrCreate	Spécifie que le système d'exploitation doit ouvrir un fichier s'il existe ; sinon, un nouveau fichier doit être créé. Si le fichier est ouvert avec FileAccess.Read , FileIOPermissionAccess.Read est requis. Si l'accès au fichier est FileAccess.ReadWrite et si le fichier existe, FileIOPermissionAccess.Write est requis. Si l'accès au fichier est FileAccess.ReadWrite et si le fichier n'existe pas, FileIOPermissionAccess.Append est requis en plus de Read et Write .
Truncate	Spécifie que le système d'exploitation doit ouvrir un fichier existant. Une fois ouvert, le fichier doit être tronqué de manière à ce que sa taille soit égale à zéro octet. Ceci nécessite FileIOPermissionAccess.Write . Toute tentative de lecture d'un fichier ouvert avec Truncate entraîne une exception.

FileAccess : énumération d'autorisation de lecture/écriture.

Nom de membre	Description	Valeur
Read	Accès en lecture au fichier. Les données peuvent être lues à partir de ce fichier. Combinez avec Write pour l'accès en lecture/écriture.	1
ReadWrite	Accès en lecture et en écriture au fichier. Les données peuvent être écrites dans le fichier et lues à partir de celui-ci.	3
Write	Accès en écriture au fichier. Les données peuvent être écrites dans le fichier. Combinez avec Read pour l'accès en lecture/écriture.	2

FileShare : énumération d'autorisation de partage.

Cette énumération est généralement utilisée pour définir si deux processus peuvent simultanément lire à partir du même fichier. Par exemple, si un fichier est ouvert et si FileShare.Read est spécifié, les autres utilisateurs peuvent ouvrir le fichier en lecture mais pas en écriture.

Nom de membre	Description	Valeur
Delete	Autorise la suppression ultérieure d'un fichier.	
Inheritable	Crée le handle de fichier hérité par les processus enfants. Ceci n'est pas pris en charge par Win32.	16
None	Refuse le partage du fichier en cours. Toute demande d'ouverture du fichier (par ce processus ou un autre) échouera jusqu'à la fermeture du fichier.	0
Read	Permet l'ouverture ultérieure du fichier pour la lecture. Si cet indicateur n'est pas spécifié, toute demande d'ouverture du fichier pour la lecture (par ce processus ou un autre) échouera jusqu'à la fermeture du fichier. Cependant, si cet indicateur est spécifié, des autorisations supplémentaires peuvent toujours être nécessaires pour accéder au fichier.	1
ReadWrite	Permet l'ouverture ultérieure du fichier pour la lecture ou l'écriture. Si cet indicateur n'est pas spécifié, toute demande d'ouverture du fichier pour la lecture ou l'écriture (par ce processus ou un autre) échouera jusqu'à la fermeture du fichier. Cependant, si cet indicateur est spécifié, des autorisations supplémentaires peuvent toujours être nécessaires pour accéder au fichier.	3
Write	Permet l'ouverture ultérieure du fichier pour l'écriture. Si cet indicateur n'est pas spécifié, toute demande d'ouverture du fichier pour l'écriture (par ce processus ou un autre) échouera jusqu'à la fermeture du fichier. Cependant, si cet indicateur est spécifié, des autorisations supplémentaires peuvent toujours être nécessaires pour accéder au fichier.	2

On peut ouvrir un fichier :

- En spécifiant son nom et son mode d'ouverture

```
FileStream fs = new FileStream("Exemple.txt", FileMode.Open);
```

Ouverture du fichier Exemple.txt existant (FileMode.Open)

- En spécifiant son nom, son mode d'ouverture et un mode d'accès

```
FileStream fs = new FileStream("Exemple.txt", FileMode.Open, FileAccess.Read);
```

Ouverture du fichier Exemple.txt existant (FileMode.Open), en lecture seule (FileAccess.Read).

- En spécifiant son nom, son mode d'ouverture, un mode d'accès et un mode de partage

```
FileStream fs = new FileStream("Exemple.txt", FileMode.Open, FileAccess.Read, FileShare.Read);
```

Ouverture du fichier Exemple.txt existant (FileMode.Open), en lecture seule (FileAccess.Read) partagée avec les autres utilisateurs (FileShare.Read).

14.4.2. Traitement des enregistrements du fichier

Les classes **StreamReader** (spécialisée dans la lecture de fichier texte) et **StreamWriter** (spécialisée dans l'écriture de fichier texte) qui dérivent des classes abstraites de base **TextReader** et **TextWriter** prennent en charge respectivement la lecture et l'écriture des flux de caractères – encodés **UTF-8*** par défaut.

La classe **StreamReader** permet de lire un fichier texte, ligne par ligne. Elle permet également de spécifier le type d'encodage, important pour nous, qui utilisons des caractères accentués.

UTF-8 (UCS Transformation Format 8 bits). UTF-8 est un codage Unicode multi-octet des caractères. Il est compatible ASCII. Chaque caractère est codé sur une suite de un à 4 mots de 8 bits (il n'existe pas actuellement de caractères codés avec plus de 4 mots).

Pour lire le contenu du fichier texte Exemple.txt (supposé existant et sans lettres accentuées), on codera :

```
FileStream fs = new FileStream("Exemple.txt",
FileMode.Open); ❶
StreamReader sr = new StreamReader (fs); ❷

string strLine = sr.ReadLine(); ❸
while (strLine != null) ❹
{
    // traitement de la ligne lue
    // ...
    strLine = sr.ReadLine(); ❺
}

sr.Close(); ❻
fs.Close(); ❼
```

- ❶ Ouverture du fichier **Exemple.txt** existant
 - ❷ Déclaration d'un objet **StreamReader** à partir de l'objet **FileStream**
 - ❸ Lecture de la première ligne du fichier
 - ❹ Boucle de lecture de tous les enregistrements du fichier
- Lire un fichier séquentiel de bout en bout suppose de programmer une **boucle**. ; Comme on sait rarement à l'avance combien d'enregistrements comporte le fichier, il faut tester la fin de fichier : si aucun enregistrement n'est lu, la méthode **ReadLine** renvoie **null**
- ❺ Lecture de l'enregistrement suivant
 - ❻ Fermeture du lecteur en cours.
 - ❼ Fermeture du fichier

Membres de StreamReader

Méthodes publiques	Description
Close	Ferme le lecteur en cours et le flux sous-jacent.
DiscardBufferedData	Permet à StreamReader d'ignorer ses données en cours.
Peek	Substitué. Retourne le prochain caractère disponible, mais ne le consomme pas.
Read	Surchargé. Substitué. Lit le caractère ou le jeu de caractères suivant dans le flux d'entrée.
Read(char [] buffer, int index, int count)	Lecture d'un jeu de caractères suivant dans le flux d'entrée.
ReadBlock	Lit un maximum de caractères à partir du flux en cours et écrit les données dans <i>buffer</i> , en commençant par <i>index</i> .
ReadLine	Substitué. Lit une ligne de caractères à partir du flux en cours et retourne les données sous forme de chaîne. Retourne null en fin de fichier.
ReadToEnd	Substitué. Lit le flux entre la position actuelle et la fin du flux.

On aurait pu aussi écrire :

```
StreamReader sr = new StreamReader("Exemple.txt");

string strLine = sr.ReadLine();
while (strLine != null)
{
    // traitement de la ligne lue
    strLine = sr.ReadLine();
}
sr.Close();
```

Le lecteur peut être créé directement en spécifiant le nom du fichier. Les caractères sont supposés être encodés UTF-8 par défaut.

Ou encore:

```
StreamReader sr = new StreamReader("Exemple.txt");

string strLine;
while ((strLine = sr.ReadLine()) != null)
{ // traitement de la ligne lue }
sr.Close();
```


Pour lire correctement un fichier contenant des lettres accentuées, cas des fichiers créés par les éditeurs évolués, il faut créer l'objet **StreamReader** de la manière suivante :

```
StreamReader sr = new StreamReader("Exemple.txt",  
ASCIIEncoding.Default);
```

Ou

```
FileStream fs = new FileStream("Exemple.txt", FileMode.Open);  
StreamReader sr = new StreamReader (fs,  
ASCIIEncoding.Default);
```

La classe StreamWriter permet d'écrire dans un flot ; elle est symétrique de StreamReader.

Exemple : lecture du fichier texte Exemple.txt et écriture des lignes lues dans un nouveau fichier Out.txt.

```
FileStream fsI = new FileStream("Exemple.txt", FileMode.Open);  
FileStream fsO = new FileStream("Out.txt",  
FileMode.CreateNew); ❶  
  
StreamReader sr = new StreamReader (fsI);  
StreamWriter sw = new StreamWriter(fsO); ❷  
  
string strLine = sr.ReadLine();  
while (strLine != null)  
{  
    // traitement de la ligne lue  
    sw.WriteLine(strLine); ❸  
    strLine = sr.ReadLine();  
}  
  
sr.Close();  
sw.Close(); ❹  
fsI.Close();  
fsO.Close();
```

❶ Création et ouverture du fichier Out.txt

❷ Déclaration d'un objet StreamWriter sw à partir de l'objet FileStream de sortie.

❸ Ecriture de la ligne lue dans le flux de sortie ;

❹ Fermeture du lecteur.

Membres de StreamWriter

Méthodes publiques	Description
Close	Ferme le lecteur en cours et le flux sous-jacent.
Flush	Force l'écriture sur le disque.
Write	Surchargé. Écrit dans le flux.
WriteLine	Surchargé. Écrit des données de la manière spécifiée par les paramètres surchargés, suivies d'un terminateur de ligne.

14.4.3. Fermeture du fichier

On a vu dans les exemples précédents l'appel systématique après traitement de la méthode Close sur les objets FileStream, StreamReader ou StreamWriter.

14.4.4. Encodage

A la différence des fichiers binaires qui stockent des octets, les fichiers texte stockent des caractères exprimés dans un encodage spécifique qui peut être l'ASCII ou plusieurs déclinaisons de l'Unicode. Vous avez eu l'occasion de découvrir ces principes lors de l'étude sur la représentation numérique des informations.

Plusieurs déclinaisons de ces encodages existent. Voici quelques exemples de ce que vous trouverez dans le framework Dot Net.

Les différentes représentations de l'encodage sont spécifiées sous la forme d'une classe System.Text.Encoding et déclinées en différentes versions comme par exemple:

- System.Text.ASCIIEncoding
- System.Text.UTF7Encoding
- System.Text.UTF8Encoding

Pour rappel, un caractère ASCII est stocké sur 1 octet et sa représentation est fonction de la culture définie au niveau de l'application. Ainsi, un caractère de même valeur pourra avoir une représentation différente fonction de la page de code utilisée.

La norme Unicode intègre la première partie de la table Ascii qui est commune à l'ensemble des langues.

Il existe aujourd'hui des mécanismes d'extension à la norme Unicode qui se trouvait initialement définie sur deux caractères ce qui permet d'étendre le nombre de caractères pouvant être représentés à plus d'un million.

Un substitut ou paire de substitution est une paire de valeurs de codage Unicode 16 bits qui représentent à elles deux un seul caractère. Ce qu'il faut garder à l'esprit est que chacune de ces paires correspond en fait à un caractère 32 bits. L'utilisation de ces caractères est toutefois restreinte. Nous verrons que la norme XML l'intègre en bonne partie mais c'est une autre histoire...

Pour préciser l'encodage d'un fichier, créer un objet de type Encoding.

Les encodages sont définis dans l'espace de nom System.Text car ils ne sont seulement utilisés pour le traitement de fichier mais pour tout traitement de texte.

```
// Je précise l'encodage du document avant de le lire
Encoding encodage = new System.Text.UTF8Encoding();

// L'ajout du symbole @ avant les guillemets permet de spécifier que
// les séquences
// d'échappement ne doivent pas être traitées, ce qui facilite
// l'écriture d'un nom de fichier qualifié complet
StreamReader sR = new StreamReader(@"C:\monFichier.txt", encodage);
```

Pour l'écriture, il faudrait de même associer un objet Encoding à un flux en écriture.

14.5. Gestion des fichiers sur disque

14.5.1. Gestion des répertoires

La classe **Directory** fournit des méthodes de classe pour la création, le déplacement et l'énumération dans les répertoires et les sous répertoires.

La classe **DirectoryInfo** fournit des informations semblables à celles de la classe **Directory**, mais les méthodes sont des méthodes d'instance : pour les utiliser, il faut d'abord créer un objet **DirectoryInfo**.

Membres de Directory (extrait)

Méthodes	Description
directory CreateDirectory (string path) *	Crée un répertoire. L'exception IOException est générée si ce nom (répertoire ou fichier) existe déjà
void Delete (string rep) *	Supprime le répertoire de nom <i>rep</i>
bool Exists (string rep) *	Renvoie true si le répertoire de nom <i>rep</i> existe
string GetCurrentDirectory ()	Renvoie le nom du répertoire courant.
string[] GetDirectories (string rep) *	Renvoie les noms des sous répertoires du répertoire <i>rep</i>
string[] GetFiles (string rep) *	Renvoie les noms des fichiers du répertoire <i>rep</i>
string[] GetLogicalDrives () *	Renvoie les noms des unités du système
void Move (string repS, string repD)	Déplace le répertoire <i>repS</i> vers <i>repD</i> L'exception IOException est générée si ce nom existe déjà
void SetCurrentDirectory (string repS)	Définit le répertoire de travail <i>repS</i>

Exemple 1: Affichage de tous les fichiers du répertoire courant (utilisation de **Directory**)

```
// affichage de tous les fichiers du répertoire courant
string curDir = Directory.GetCurrentDirectory() ; ❶
foreach (string f in Directory.GetFiles(curDir)) ❷
{
    Console.WriteLine(f);
}
```

❶ Recherche du répertoire courant

❷ Boucle d'affichage du nom des fichiers

L'avantage d'utiliser la classe **DirectoryInfo** est d'obtenir des informations supplémentaires grâce aux propriétés fournies par la classe.

Propriétés de DirectoryInfo

Propriétés	Description
Attributes	Obtient ou définit des attributs de répertoire(valeur de l'énumération FileAttributes)
CreationTime	Date de création du répertoire.
Exists*	Indique si le répertoire existe.
FullName	Nom complet du répertoire(y compris l'unité).
LastAccessTime	Date de dernier accès au répertoire.
LastWriteTime	Date de dernière écriture dans le répertoire.
Name	Nom du répertoire.
Parent	Répertoire père.

Exemple 2: Affichage de tous les sous répertoires du répertoire courant avec leur date de création.

```
// affichage de tous les fichiers du répertoire courant
string curDir = Directory.GetCurrentDirectory() ; ❶
DirectoryInfo cdi = new DirectoryInfo(curDir); ❷
foreach (DirectoryInfo di in cdi.GetDirectories()) ❸
{
    Console.WriteLine(fi.Name + fi.CreationTime.ToString("d")) ❹;
}
```

- ❶ Recherche du répertoire courant
- ❷ Création d'une instance DirectoryInfo
- ❸ Parcours des sous répertoires
- ❹ Affichage des informations du sous-répertoire.

14.5.2. Gestion des fichiers

La classe File fournit des méthodes de classe pour créer, copier, supprimer, déplacer et ouvrir des fichiers et facilite la création d'objets FileStream. La classe FileInfo fournit des méthodes d'instance identiques.

Membres de File (extrait)

Méthodes	Description
void Copy (string f1, string f2, bool bRep) *	Copie le fichier <i>f1</i> vers le fichier <i>f2</i> : si <i>bRep</i> = true , <i>f2</i> est remplacé.
void Move (string f1, string f2) ;*	Déplace le fichier <i>f1</i> vers le fichier <i>f2</i> : (le renomme dans le même répertoire)
void Delete (string f) ;	Supprime le fichier <i>f</i> .
bool Exists (string f) ;	Renvoie true si le fichier existe.
string GetCurrentDirectory () ;	Renvoie le nom du répertoire courant.
FileAttributes GetAttributes (string f) ;*	Renvoie les attributs du fichier (Archive, Directory, Hidden, Normal, ReadOnly, System...)
void SetAttributes (string f, FileAttributes) ;	Modifie les attributs du fichier.

Exemple 1: Copie d'un fichier sans vouloir écraser le fichier de destination s'il existe déjà.

```
try
{
File.Copy("ancien.txt ", "nouveau.txt", false) ;
}
catch (FileNotFoundException)
{
Console.WriteLine("Le fichier source n'existe pas ");
}
catch (IOException)
{
Console.WriteLine(" Le fichier de destination existe déjà");
}
```

L'avantage d'utiliser la classe FileInfo est semblable à l'utilisation de DirectoryInfo par rapport à Directory.

Propriétés de FileInfo

Propriétés	Description
Attributes	Obtient ou définit des attributs de répertoire(valeur de l'énumération FileAttributes)
CreationTime	Date de création du fichier.
Exists*	Indique si le fichier existe.
FullName	Nom complet du fichier(unité, répertoire,nom, extension).
LastAccessTime	Date de dernier accès.
LastWriteTime	Date de dernière écriture.
Length	Taille du fichier en octets.
Name	Nom du fichier(sans le répertoire mais avec l'extension).
DirectoryName	Nom du répertoire du fichier

Exemple 2: Renommer un fichier.

```
FileInfo fi = new FileInfo("Exemple.txt ");
if (fi.Exists) fi.MoveTo("ExempleAnc.txt");
```

L'exception IOException est générée si le fichier de destination existe déjà.

15. Annexe 1 : identificateurs du langage C#

Identificateurs prédéfinis réservés par le langage C#			
abstract	event	new	struct
as	explicit	null	switch
base	extern	object	this
bool	false	operator	throw
break	finally	out	true
byte	fixed	override	try
case	float	params	typeof
catch	for	private	uint
char	foreach	protected	ulong
checked	goto	public	unchecked
class	if	readonly	unsafe
const	implicit	ref	ushort
continue	in	return	using
decimal	int	sbyte	virtual
default	interface	sealed	volatile
delegate	internal	short	void
do	is	sizeof	while
double	lock	stackalloc	
else	long	static	
enum	namespace	string	
Mots clés contextuels à ne pas utiliser			
from	get	group	into
join	let	orderby	partial (type)
partial (méthode)	select	set	value
where (contrainte)	where (clause)	yield	