

Applications financières en C#

Semestre 1



Organisation de la formation



- Module de 18 heures

Date	Heure
To define	Rendu des projets finaux

- **Laurent DAVOUST, FRM**

- **Diplôme:**

- GARP – **Financial Risk Manager (FRM ®)** – 2023
- Master Université Paris Dauphine –
Ingénierie Economique et Financière – 2015

- **Expériences professionnelles:**

- EDF – **Risk Manager** – 2014/2015
- APTimum – **Consultant Risque de Marché** – 2015/2018
- Société Générale – **Ingénieur Modélisateur Marché & IT Data Manager** – 2018/2023
- Phit Formation & Solutions – **Fondateur** – 2023/aujourd'hui
- Université Paris-Dauphine – **Intervenant** – 2020/aujourd'hui



- **Laurent DAVOUST, FRM**

- **Compétences clefs:**

- **Développement informatique:** Python, C#, VBA, PHP, SQL
- **Connaissances financières:** Gestion d'actifs, gestion des risques de marché, gestion patrimoniale, valorisations comptables et prudentielles (RP, PVA, IPV), et modèles de valorisation financiers
- **Entreprenariat:** Lancement d'un projet, étapes dans la création d'entreprise, structuration d'un organisme de formation



- **Me contacter:**

- **Mail:** laurent.davoust@phit-formation.com
- **LinkedIn:** <https://www.linkedin.com/in/laurent-davoust-frm-439149ab/>

1. Bases de C#

1. **Comprendre les fondamentaux du langage C#** : Revoir les concepts de base tels que les variables, les types de données, les opérateurs, les structures conditionnelles et les boucles appliqués à ce langage.
2. **Maîtriser la gestion des exceptions** : Gérer les erreurs de manière efficace en utilisant les blocs try, catch, et finally.
3. **Manipuler les collections** : Découvrir les collections de base (tableaux, listes, dictionnaires) et savoir quand les utiliser.

2. Programmation Orientée Objet (POO)

1. **Maîtriser la gestion des exceptions** : Apprendre à gérer les erreurs de manière efficace en utilisant les blocs `try`, `catch`, et `finally`.
2. **Appliquer les principes de la POO en C#** : Implémenter les concepts fondamentaux de la POO tels que l'encapsulation, l'héritage, le polymorphisme, et l'abstraction.
3. **Créer des classes et des objets** : Concevoir et implémenter des classes, créer des objets, et manipuler leurs propriétés et méthodes.
4. **Utiliser des interfaces et des classes abstraites** : Les intégrer dans des architectures logicielles orientées objet.

3. Notions avancées

1. **Découvrir et implémenter les principaux design patterns** : Apprendre à reconnaître et appliquer les design patterns couramment utilisés dans les applications financières (ex : Singleton, Factory, Observer).
2. **Maîtriser LINQ pour manipuler les données** : Comprendre et utiliser LINQ pour effectuer des requêtes sur des collections d'objets, en optimisant l'accès et la manipulation des données.
3. **Gérer l'asynchronisme en C#** : Apprendre à utiliser les mots-clés `async` et `await`, et à implémenter des méthodes asynchrones pour améliorer la réactivité des applications.
4. **Applications financières** : Implémentation d'algorithmes financiers en prenant en compte les spécificités du C#

1. Les bases du langage

1. Introduction au C# et à l'environnement de développement
2. Variables, Types de Données, et Opérateurs
3. Structures de Contrôle
4. Gestion des dates : DateTime

2. Structures de données « complexes »

1. Listes
2. Dictionnaires
3. Ensembles
4. Vecteurs et matrices
5. L'extension LINQ

3. Développement de classes C#

1. Introduction à la Programmation Orientée Objet
2. Héritage et Polymorphisme
3. Classes Abstraites et Interfaces
4. Type enum

4. Pour aller plus loin

1. Conventions de nom en C#
2. Gestion des Exceptions
3. Design Patterns
4. Principe de substitution de Liskov
5. Multiple héritage via Interfaces
6. Méthodes d'extension
7. Dépendances / Injection de Dépendances
8. Gestion des dépendances
9. Manipulation de fichier CSV
10. Manipulation de fichier Excel (COM)
11. Manipulation de fichier JSON
12. Tests unitaires

- **Connaissances**

- Compréhension du système d'exploitation Windows et de sa gestion des fichiers
- Logique de programmation

- **Equipements requis**

- Ordinateur avec Visual Studio 2022 ou + installé

- **Projet final (20 points)**

- Ce projet est défini dans le document regroupant à la fois les exercices et les projets vus en cours, mais aussi le projet d'évaluation.
- Dans ce document, y sont inscrits toutes les étapes à respecter pour le projet
- Ces documents sont accessibles directement sur le github.

Les bases du C#

Module 1



1.1. Introduction au C# et à l'environnement de développement

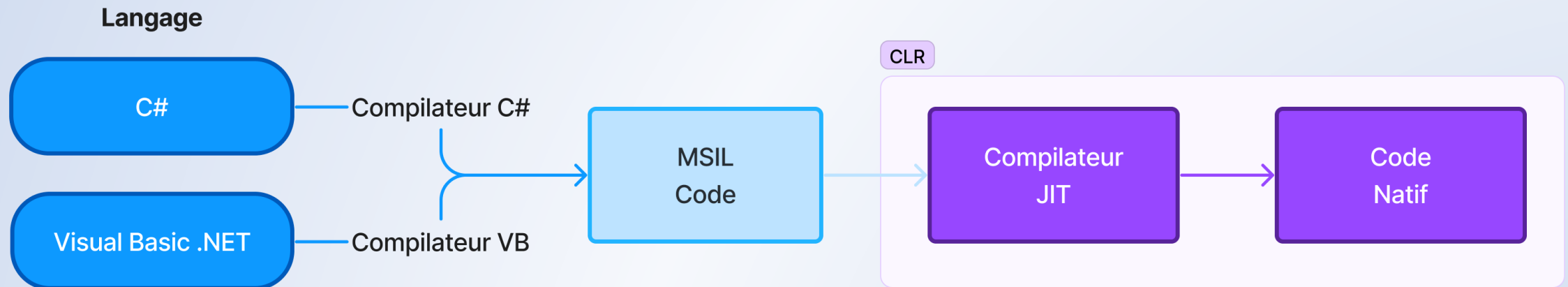


- **Genèse du C#**

- Créateur : Microsoft
- Première annonce : 2000 (dans le projet .NET Framework)
- Objectif : langage moderne, orienté objet et simple à utiliser.
- Inspiration : C, C++ et essentiellement Java.

- **.NET Framework ?**

- Fournir un environnement complet pour le développement et l'exécution d'applications Windows
- Runtime *Common Language Runtime (CLR)* utilisé par les langages de la plateforme .NET (C#, VB.NET, ...) : lancement & exécution de programmes, ramasse-miettes et gestion d'exceptions.

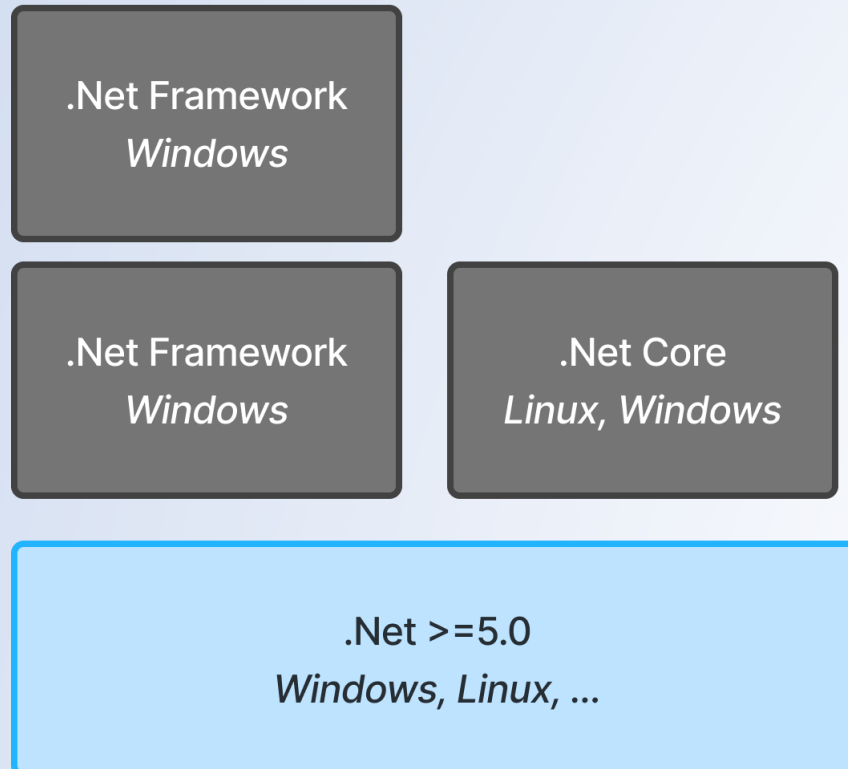


Evolution du C#

Version	Année de sortie	Description	Librairie associée
1.0	2002	Caractéristiques initiales du langage	.Net Framework 1.0
2.0	2005	Introduction des génériques, des méthodes anonymes	.Net Framework 2.0
3.0	2008	Introduction de LINQ et des expressions lambda.	.Net Framework 3.5
4.0	2010	Introduction des types dynamiques	.Net Framework 4.0
5.0	2012	Introduction des méthodes asynchrones	.Net Framework 4.5
6.0	2015	Améliorations de syntaxe	.Net Framework 4.6
7.0	2016	Introduction des tuples et des fonctions locales	.Net Framework 4.7
8.0	2019	Introduction des membres read only	.Net Standard 2.1 .Net Core 3.0
9.0	2020	Enregistrements (records), types de référence non nullables par défaut	.NET 5.0

Pour plus d'informations :
<https://learn.microsoft.com/fr-fr/dotnet/csharp/whats-new/csharp-version-history>







Evolution du .Net



^ Versions prises en charge

Version	Type de publication	Phase d'accompagnement	Dernière mise en production	Date de publication la plus récente	Fin du support
.NET 9.0	Assistance à terme standard ⓘ	Aperçu ⓘ	9.0.0-preview.7	13 août 2024	
.NET 8.0 (dernière)	Prise en charge à long terme ⓘ	Actif ⓘ	8.0.8	15 août 2024	10 novembre 2026
.NET 6.0	Prise en charge à long terme ⓘ	Maintenance ⓘ	6.0.33	13 août 2024	12 novembre 2024

Source : <https://dotnet.microsoft.com/fr-fr/download/dotnet>

Jan 2024	Jan 2023	Change	Programming Language		Ratings	Change
1	1			Python	13.97%	-2.39%
2	2			C	11.44%	-4.81%
3	3			C++	9.96%	-2.95%
4	4			Java	7.87%	-4.34%
5	5			C#	7.16%	+1.43%
6	7	▲		JavaScript	2.77%	-0.11%
7	10	▲		PHP	1.79%	+0.40%

Source : <https://www.blogdumoderateur.com/classement-langage-programmation-annee-2023/>

- **Développement Web**

- **ASP.NET Core:**

- Développement de sites web, applications web modernes, et services web RESTful.

- **Applications de bureau**

- **Windows Forms et WPF :**

- Pour la création d'applications de bureau traditionnelles sous Windows, C# est utilisé avec Windows Forms (WinForms) et Windows Presentation Foundation (WPF), ce dernier offrant plus de capacités graphiques avancées et une meilleure séparation du code et de l'interface utilisateur.

- **Développement de Jeux**

- **Unity:**

- C# est le langage principal pour le développement de jeux avec le moteur Unity, l'un des moteurs de jeu les plus populaires pour le développement de jeux sur plusieurs plateformes, y compris PC, consoles, mobiles, et réalité virtuelle.

- **Développement d'Applications Mobiles**

- **Xamarin :**

- C# est utilisé pour développer des applications mobiles natives pour Android et iOS

- **Développement d'Applications Cloud et Microservices**

- **Azure :**

- Avec l'avènement du cloud computing, C# est devenu un choix populaire pour le développement d'applications cloud et de microservices, notamment grâce à l'intégration avec Microsoft Azure et les capacités multiplateformes de .NET Core.

- **Faire un tour sur l'application**
 - Création de nouveaux projets
 - Barre des menus, barre d'outils
 - Gestionnaire de package NUGET
 - Explorateur de solution
 - Fenêtre de code
 - Ajustement des thèmes
 - Fenêtre de débogage
 - Fenêtre de tests
 - IntelliSense

Variables, Types de données, Opérateurs



- **Fonctionnalités du C#**

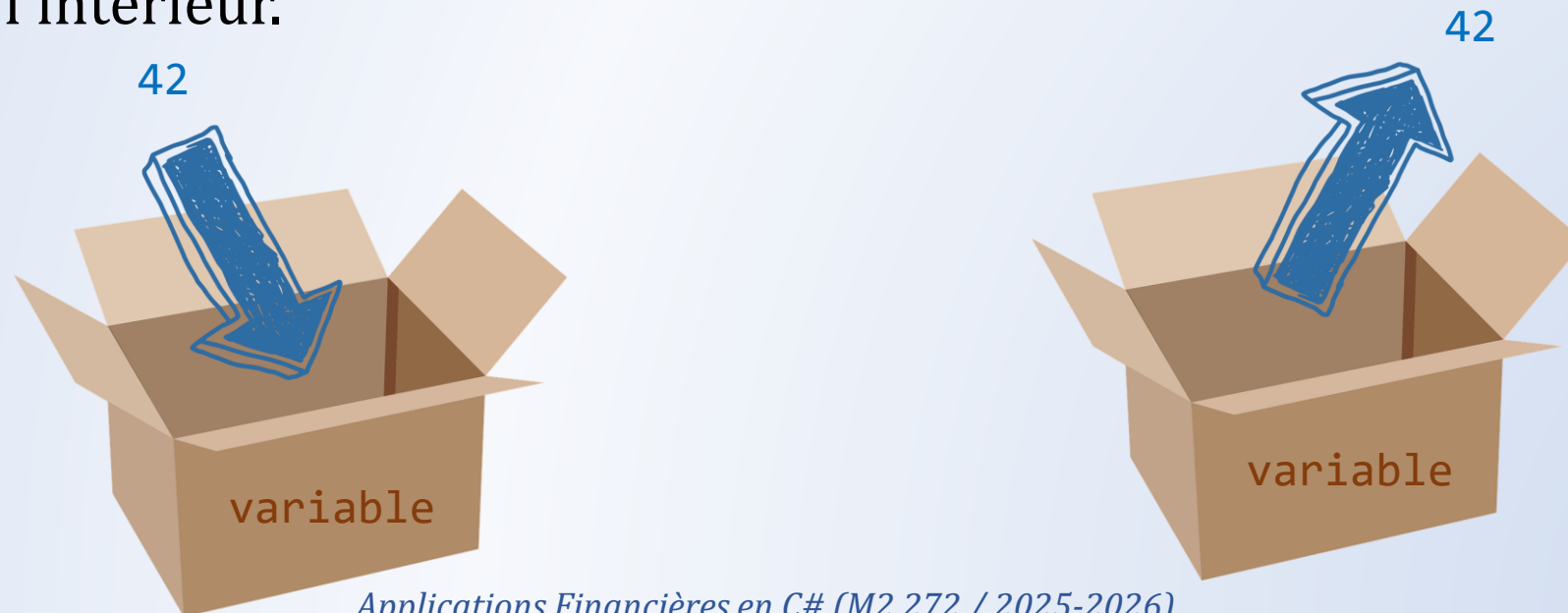
- C# est un langage de programmation orienté objet, **orienté composant** = définition des types et de leur comportement
- **Garbage Collection** = récupération automatique de la mémoire occupée par des objets inutilisés inaccessibles
- **Type nullable** = protection contre les variables qui ne font pas référence à des objets alloués
- **Gestion des exceptions** = approche structurée et extensible de la détection et de la récupération des erreurs.
- **Système de type unifié** = tous les types C#, y compris les types primitifs (ex int/double) héritent d'un seul type object racine. Tous les types partagent un ensemble d'opérations communes.

```
using System;

class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

• Définition

- Espace de stockage nommé dans la mémoire de l'ordinateur qui est utilisé pour conserver une valeur qui peut être modifiée pendant l'exécution du programme.
- On peut imaginer une variable comme une boîte dans laquelle vous pouvez stocker une information.
- Cette boîte a un nom, et vous pouvez utiliser ce nom pour accéder à l'information stockée à l'intérieur.



- **Type**

- Le type de la variable détermine quel genre de données elle peut stocker (par exemple, des nombres, des textes, ou des valeurs booléennes) et comment ces données sont stockées en mémoire.

- **Nom**

- Le nom de la variable est l'identifiant unique que vous utilisez pour faire référence à la variable dans votre code.

- **Valeur**

- La valeur de la variable est l'information actuellement stockée dans la variable. La valeur peut être assignée lors de la déclaration de la variable ou modifiée au cours de l'exécution du programme.

- **Portée**

- La portée d'une variable détermine où dans le code la variable est accessible.

- **Durée de vie**

- La durée de vie d'une variable fait référence à la période pendant laquelle la variable existe en mémoire.

- **Entiers**

- **int** : Représente un entier signé 32 bits. Il est utilisé pour stocker des valeurs numériques sans partie décimale. Plage de valeurs : -2,147,483,648 à 2,147,483,647.
- **long** : Représente un entier signé 64 bits. Utilisé pour des valeurs entières plus grandes. Plage de valeurs : -9,223,372,036,854,775,808 à 9,223,372,036,854,775,807.
- **byte** : Représente un entier non signé 8 bits. Utilisé pour des valeurs entre 0 et 255.

- **Nombres à Virgule Flottante**

- **float** : Représente un nombre à virgule flottante 32 bits. Utilisé pour des valeurs numériques avec des fractions, nécessitant moins de précision. Plage de valeurs : environ $\pm 1.5 \times 10^{-45}$ à $\pm 3.4 \times 10^{38}$, avec 7 chiffres significatifs.
- **double** : Représente un nombre à virgule flottante 64 bits. Utilisé pour des valeurs numériques avec des fractions, nécessitant plus de précision. Plage de valeurs : environ $\pm 5.0 \times 10^{-324}$ à $\pm 1.7 \times 10^{308}$, avec 15-16 chiffres significatifs.

- **Booléen**

- **bool** : Représente une valeur booléenne, true ou false. Utilisé pour les opérations logiques et les conditions

- **Caractère**

- **char** : Représente un caractère unique Unicode 16 bits. Utilisé pour stocker des caractères, comme les lettres et les chiffres.
- **string** : Représente une chaîne de caractères.

```
// Affectation d'une valeur à une variable de type int (entier)
int age = 25;

// Affectation d'une valeur à une variable de type double
double temperature = 36.6;

// Affectation d'une valeur à une variable de type float
float height = 1.75;

// Affectation d'une valeur à une variable de type long
long distance = 1234567890123;

// Affectation d'une valeur à une variable de type string
string name = "John Doe";

// Affectation d'une valeur à une variable de type char
char initial = 'J';
```

- **Pour Entiers (int et long) et Nombres à virgule flottante (double et float)**
 - **Opérations arithmétiques :**
 - addition (+),
 - soustraction (-),
 - multiplication (*),
 - division (/),
 - modulo (%).
 - **Comparaisons :**
 - égalité (==),
 - inégalité (!=),
 - supérieur (>),
 - inférieur (<),
 - supérieur ou égal (>=),
 - inférieur ou égal (<=).
 - **Opérations logiques:**
 - ET (&),
 - OU (|)

- **Pour Caractère (char)**

- **Comparaisons :**

- égalité (==),
 - inégalité (!=),
 - supérieur (>),
 - inférieur (<),
 - supérieur ou égal (>=),
 - inférieur ou égal (<=).

- **Opérations de conversion :**

- conversion en majuscule ou minuscule (Char.ToUpper, Char.ToLower).

- **Tests :**

- vérifier si le caractère est une lettre, un chiffre, etc. (Char.IsLetter, Char.IsDigit, etc.).

- **Pour Chaînes de Caractères (string)**

- **Concaténation :**

- assembler deux chaînes (+ ou String.Concat).

- **Comparaison :**

- égalité (==),
 - comparaison lexicographique (String.Compare).

- **Recherche :**

- trouver un sous-ensemble (String.Contains, String.IndexOf).

- **Modification :**

- remplacement de texte (String.Replace),
 - Enlever les espaces en début et fin (String.Trim),
 - conversion en majuscules ou minuscules (String.ToUpper, String.ToLower).

- **Division :**

- découper une chaîne en sous-chaînes (String.Split).

Consigne :

Tester les éléments suivants pour une variable de type int :

Addition (+) : $5 + 3 = 8$

Soustraction (-) : $5 - 3 = 2$

Multiplication () : $5 * 3 = 15$*

Division (/) : $5 / 3 = 1$ si int

Modulo (%) : $5 \% 3 = 2$

Consigne :

Tester les éléments suivants pour une variable de type int :

Égalité ($==$) : $5 == 3$ donne False

Inégalité ($!=$) : $5 != 3$ donne True

Supérieur ($>$) : $5 > 3$ donne True

Inférieur ($<$) : $5 < 3$ donne False

Supérieur ou égal ($>=$) : $5 >= 3$ donne True

Inférieur ou égal ($<=$) : $5 <= 3$ donne False

Consigne :

Tester les éléments suivants pour une variable de type int :

ET (&) : 5 & 3 donne 1 (en considérant les représentations binaires de 5 (101) et 3 (011), le résultat est 001)

OU (|) : 5 | 3 donne 7 (en considérant les représentations binaires de 5 (101) et 3 (011), le résultat est 111)

```
// Opérations arithmétiques
Console.WriteLine("5 + 3 = " + (5 + 3));
Console.WriteLine("5 - 3 = " + (5 - 3));
Console.WriteLine("5 * 3 = " + (5 * 3));
Console.WriteLine("5 / 3 = " + (5 / 3.0)); // Utilisation de 3.0 pour obtenir un résultat en virgule flottante
Console.WriteLine("5 % 3 = " + (5 % 3));

// Comparaisons
Console.WriteLine("5 == 3 donne " + (5 == 3));
Console.WriteLine("5 != 3 donne " + (5 != 3));
Console.WriteLine("5 > 3 donne " + (5 > 3));
Console.WriteLine("5 < 3 donne " + (5 < 3));
Console.WriteLine("5 >= 3 donne " + (5 >= 3));
Console.WriteLine("5 <= 3 donne " + (5 <= 3));

// Opérations logiques (bit à bit)
Console.WriteLine("ET (&) : 5 & 3 donne " + (5 & 3));
Console.WriteLine("OU (|) : 5 | 3 donne " + (5 | 3));
```

Consigne :

Tester les éléments suivants pour une variable de type char :

Égalité (==) : 'a' == 'b' donne False

Inégalité (!=) : 'a' != 'b' donne True

Supérieur (>) : 'b' > 'a' donne True

Inférieur (<) : 'a' < 'b' donne True

Supérieur ou égal (>=) : 'b' >= 'a' donne True

Inférieur ou égal (<=) : 'a' <= 'b' donne True

Consigne :

Tester les éléments suivants pour une variable de type char :

Conversion en majuscule : 'a'.ToUpper() donne 'A'

Conversion en minuscule : 'A'.ToLower() donne 'a'

Vérifier si le caractère est une lettre : 'a'.IsLetter donne True

Vérifier si le caractère est un chiffre : '1'.IsDigit donne True

```
// Comparaisons
Console.WriteLine("'a' == 'b': " + ('a' == 'b')); // False
Console.WriteLine("'a' != 'b': " + ('a' != 'b')); // True
Console.WriteLine("'b' > 'a': " + ('b' > 'a')); // True
Console.WriteLine("'a' < 'b': " + ('a' < 'b')); // True
Console.WriteLine("'b' >= 'a': " + ('b' >= 'a')); // True
Console.WriteLine("'a' <= 'b': " + ('a' <= 'b')); // True

// Conversion en majuscule et en minuscule
Console.WriteLine("'a'.ToUpper(): " + 'a'.ToString().ToUpper()); // 'A'
Console.WriteLine("'A'.ToLower(): " + 'A'.ToString().ToLower()); // 'a'

// Vérifier si le caractère est une lettre
Console.WriteLine("'a'.IsLetter: " + Char.IsLetter('a')); // True

// Vérifier si le caractère est un chiffre
Console.WriteLine("'1'.IsDigit: " + Char.IsDigit('1')); // True
```

Consigne :

Tester les éléments suivants pour une variable de type string :

Concatène « Hello » et « world ! » avec le + et le String.Concat

Compare « Apple » et « apple »

Recherche « world » dans « Hello world »

Remplace « world » par « C# » dans « Hello world ! »

Découpe la chaîne de caractères au niveau des virgules

« apple,orange,banana »

Exemple pour les strings

44

```
// Concaténation
string hello = "Hello, ";
string world = "world!";
string concatenated = hello + world;
string concatenatedMethod = String.Concat(hello, world);

// Comparaison
string firstString = "apple";
string secondString = "Apple";
bool areEqual = firstString == secondString;
int lexicographicalComparison = String.Compare(firstString, secondString, StringComparison.OrdinalIgnoreCase);
```


Exemple pour les strings

45

```
// Recherche
string sentence = "Hello, world!";
bool containsWord = sentence.Contains("world"); // Vrai
int indexOfWord = sentence.IndexOf("world");

// Modification
string messyString = " Hello, world! ";
string replaced = messyString.Replace("world", "C#"); // " Hello, C#! "
string trimmed = messyString.Trim(); // "Hello, world!"
string upperCased = messyString.ToUpper(); // " HELLO, WORLD! "
string lowerCased = messyString.ToLower(); // " hello, world! "

//Division
string fruits = "apple,orange,banana";
string[] splitFruits = fruits.Split(',');
```

- **CamelCase (camelCase)**

- **Utilisation** : principalement pour les variables locales et les paramètres de méthode.
- **Description** : La première lettre du nom est en minuscule, et la première lettre de chaque mot suivant est en majuscule. Exemple : `localVariable`, `methodParameter`.

- **PascalCase (PascalCase)**

- **Utilisation** : pour les noms de classe, méthode, propriétés et constantes publiques.
- **Description** : Chaque mot commence par une lettre majuscule, y compris le premier mot. Exemple : `ClassName`, `MethodName`, `PublicProperty`.

- **Clarté** : Le nom doit indiquer clairement l'usage de la variable. Préférer `customerAddress` à `addr`.
- **Concision** : Le nom doit être assez court tout en restant descriptif. Éviter les noms trop longs et complexes.
- **Éviter les abréviations** : sauf celles universellement reconnues. Par exemple, `Http` plutôt que `HypertextTransferProtocol`.
- **Éviter les chiffres** : aux débuts et fins de noms, sauf si le chiffre fait partie intégrante de la signification (ex : `md5Hash`).
- **Pas de caractères spéciaux** : Éviter l'utilisation de caractères spéciaux comme `$`, `%`, `#`, etc.

- **Types explicites** : Utilisez-les lorsque le type n'est pas évident au moment de la déclaration. Ils améliorent la lisibilité du code. Exemple : `int customerCount = 10;` .
- **Mot-clé var** : Utilisez var lorsque le type de la variable est évident à partir de l'expression de droite. var peut rendre le code plus propre, surtout avec des types complexes, mais doit être utilisé lorsque le type est clairement identifiable. Exemple : `var customerList = new List<Customer>();` .

- **Déclarez les variables le plus près possible de leur premier usage** pour réduire la portée des variables et améliorer la lisibilité.
- **Initialisez les variables au moment de la déclaration** si possible. Cela aide à éviter les erreurs liées à l'utilisation de variables non initialisées.

- **Définition:**

- Convertir une variable d'un type de données à un autre (par exemple string → double)

- **Conversion Implicite:**

- Se fait automatiquement par le compilateur lorsque la conversion est sûre, c'est-à-dire lorsqu'il n'y a aucun risque de perte de données.

- **Conversion Explicite :**

- Nécessite une indication explicite de l'intention du programmeur.

```
// Conversion implicite
int a = 10;
double b = a;

// Conversion Explicite
double a = 9.78;
int b = (int)a;
```


- **Convert.ToDouble()** :

- La méthode Convert.ToDouble() tente de convertir un objet en double. Si la conversion échoue (par exemple, si la chaîne n'est pas un nombre valide), une exception sera levée.

- **Double.Parse()** :

- La méthode Double.Parse() est utilisée pour convertir une chaîne en double. Comme Convert.ToDouble(), elle lève une exception si la chaîne n'est pas un nombre valide.

- **Double.TryParse()**

- La méthode Double.TryParse() tente de convertir une chaîne en double, mais au lieu de lever une exception en cas d'échec, elle renvoie false et place 0 dans la variable de sortie.

```
// Convert.ToDouble()
string str = "123.45";
double result = Convert.ToDouble(str);

// Double.Parse()
string str = "123.45";
double result = Double.Parse(str);

// Double.TryParse()
string str = "123.45";
double result;
bool success = Double.TryParse(str, out result);
if (success) {
    Console.WriteLine("Conversion réussie : " +
result);
} else {
    Console.WriteLine("Conversion échouée.");
}
```

- **<type>.ToString()** :
 - La méthode ToString() permet de transformer une variable en string. Pour les types complexes, il est possible de définir son comportement.

```
// int
int number = 123;
string str = number.ToString();

// double
double number = 123.45;
string str = number.ToString();
```


Structures de contrôle



- **Définition**

- Les structures de contrôle sont des blocs de construction fondamentaux dans n'importe quel langage de programmation, y compris C#.
- Elles permettent à un programme de **prendre des décisions** (structures conditionnelles) ou de **répéter une action plusieurs fois** (boucles).
- Sans les structures de contrôle, un programme ne pourrait exécuter des instructions que de manière séquentielle, de haut en bas, sans pouvoir réagir aux différentes conditions ou répéter des opérations, ce qui limiterait considérablement sa fonctionnalité et son efficacité.

- **Définition**

- Les instructions conditionnelles permettent à votre programme de prendre des décisions et d'exécuter des instructions différentes en fonction de conditions spécifiques.
- Les instructions conditionnelles se ramènent toujours à des tests de **booléen** → c'est vrai ou c'est faux !

- **2 types d'instructions conditionnelles**

- **If et If-Else :**

- plus flexible pour tester des conditions qui ne sont pas basées sur une seule variable ou pour des comparaisons plus complexes.

- **Switch :**

- Plus efficace lorsque vous avez de nombreuses conditions à comparer avec la même variable ou expression.

```
if (condition)
{
    // Instructions à exécuter si la condition est vraie
}
else
{
    // Instructions à exécuter si la condition est fausse
}
```

Consigne :

Ecrire un programme qui va vérifier si à partir d'un âge saisi dans une variable « age », si on est majeur ou non

```
int age = 20;

if (age >= 18)
{
    Console.WriteLine("Vous êtes majeur.");
}
else
{
    Console.WriteLine("Vous êtes mineur.");
}
```

```
switch (expression)
{
    case valeur1:
        // Instructions pour valeur1
        break;
    case valeur2:
        // Instructions pour valeur2
        break;
    ...
    default:
        // Instructions si aucune correspondance n'est trouvée
        break;
}
```

Consigne :

Ecrire un programme qui va écrire le mot du mois pour un nombre saisi entre 1 et 12 (par exemple 1 → janvier)


```
int month = 4;

switch (month)
{
    case 1:
        Console.WriteLine("Janvier");
        break;
    case 2:
        Console.WriteLine("Février");
        break;
    // Ajoutez d'autres mois ici...
    default:
        Console.WriteLine("Mois inconnu");
        break;
}
```

- **Définition**

- Les boucles sont des structures de contrôle qui répètent une section de code un nombre déterminé ou indéterminé de fois, tant qu'une condition est respectée.

- **4 types d'instructions**

- **For**: Utilisation lorsque le nombre de répétitions est connu à l'avance
- **Foreach** : La boucle foreach est utilisée pour parcourir les éléments d'une collection ou d'un tableau. Elle est particulièrement utile quand on souhaite exécuter une opération sur chaque élément d'une collection sans se soucier de l'indexation.
- **While** : La boucle while est utilisée lorsque le nombre de répétitions n'est pas connu à l'avance. Elle continue de s'exécuter tant que la condition spécifiée est vraie. La principale différence entre while et for est que while est préféré dans les cas où le nombre d'itérations n'est pas déterminé avant l'entrée dans la boucle.
- **Do-While** : La boucle do-while est similaire à la boucle while, à ceci près qu'elle garantit que le bloc de code est exécuté au moins une fois avant de vérifier la condition.

```
for (initialisation; condition; itération)
{
    // Bloc de code à exécuter
}
```

- **Initialisation** : Déclare et initialise le compteur de boucle.
- **Condition** : La boucle continue tant que cette condition est vraie.
- **Itération** : Incrémente ou décrémente le compteur de boucle.

Consigne :

Ecrire un programme qui va afficher les nombres de 1 à 10.

```
for (int i = 1; i <= 10; i++)  
{  
    Console.WriteLine(i);  
}
```

```
while (condition)
{
    // Bloc de code à exécuter
}
```

- La principale différence entre while et for est que while est préféré dans les cas où le nombre d'itérations n'est pas déterminé avant l'entrée dans la boucle.

Consigne :

Ecrire les termes de la suite suivante $u(n) = u(n - 1) + n$, $u(0) = 0$ jusqu'à que $u(n)$ ait atteint 1000.

```
int u = 0; // Initialisation de u(0)
int n = 0; // Le premier terme de la suite

while (u < 1000) // Continuer tant que u(n) < 1000
{
    Console.WriteLine($"u({n}) = {u}"); // Affiche le terme actuel de la suite
    n++; // Incrémenter n pour passer au terme suivant
    u = u + n; // Calculer u(n) selon la formule
}

// Afficher le premier terme de la suite pour lequel u(n) >= 1000
Console.WriteLine($"Le premier terme de la suite pour lequel u(n) atteint ou dépasse 1000
est u({n}) = {u}.");
```



```
do
{
    // Bloc de code à exécuter
} while (condition);
```

- La boucle do-while est similaire à la boucle while, à ceci près qu'elle garantit que le bloc de code est exécuté au moins une fois avant de vérifier la condition.

Consigne :

Demander à l'utilisateur de saisir un mot de passe jusqu'à ce qu'il saisisse le bon mot de passe.

```
string password;  
do  
{  
    Console.WriteLine("Entrez le mot de passe :");  
    password = Console.ReadLine();  
} while (password != "secret");  
Console.WriteLine("Mot de passe correct !");
```

```
foreach (type variable in collection)
{
    // Bloc de code à exécuter
}
```

- **Avantages par rapport aux autres types de boucles**
 - Plus simple à écrire et à lire pour les opérations sur les collections.
 - Moins de risque d'erreurs, comme dépasser les limites d'un tableau.

Consigne :

Afficher tous les éléments d'un tableau de chaînes.

```
string[] names = { "Alice", "Bob", "Charlie" };  
foreach (string name in names)  
{  
    Console.WriteLine(name);  
}
```

- **Définition**

- Les instructions de saut en C# sont des outils puissants qui permettent de modifier le flux d'exécution d'un programme.

- **2 types d'instructions de saut**

- **Break:** termine immédiatement l'exécution de la boucle (for, foreach, while, do-while) ou d'un bloc switch dans lequel elle se trouve, transférant le contrôle à l'instruction suivant la boucle ou le switch.
- **Continue:** interrompt l'itération en cours dans une boucle (for, foreach, while, do-while) et déclenche immédiatement l'itération suivante de la boucle. Contrairement à break, continue ne termine pas la boucle; elle passe simplement à l'itération suivante.

Exemple – break

76

```
for (int i = 0; i < 10; i++) {  
    if (i == 5) {  
        break; // Interrompt la boucle lorsque i atteint 5  
    }  
    Console.WriteLine(i);  
}
```

```
switch (variable) {  
    case 1:  
        // Instructions pour le cas 1  
        break; // Termine ce cas  
    case 2:  
        // Instructions pour le cas 2  
        break; // Termine ce cas  
}
```

```
for (int i = 0; i < 10; i++) {  
    if (i % 2 == 0) {  
        continue; // Passe à l'itération suivante si i est pair  
    }  
    Console.WriteLine(i); // Affiche seulement les nombres impairs  
}
```

- Contrairement à `break`, `continue` ne termine pas la boucle; elle passe simplement à l'itération suivante.

- **Définition**

- L'opérateur ternaire est une alternative compacte aux instructions if-else, permettant d'assigner une valeur à une variable ou de choisir entre deux opérations en une seule ligne de code.

- **Avantages de l'Opérateur Ternaire**

- **Concision** : Réduit le nombre de lignes de code pour des décisions simples.
- **Clarté** : Peut rendre le code plus lisible en éliminant les besoins des blocs if-else pour des opérations simples.
- **Inline** : Permet des affectations conditionnelles directes dans les déclarations de variables.

```
condition ? expressionSiVrai : expressionSiFaux;
```

- **condition** : une expression qui est évaluée. Le résultat doit être de type booléen (true ou false).
- **expressionSiVrai** : l'expression qui est évaluée et renvoyée si la condition est true.
- **expressionSiFaux** : l'expression qui est évaluée et renvoyée si la condition est false.

```
int score = 85;
string grade;
if (score >= 60) {
    grade = "Pass";
} else {
    grade = "Fail";
}
```

```
int score = 85;
string grade = score >= 60 ?
    "Pass" : "Fail";
```

- **Comparaison avec les Instructions If-Else**

- Les instructions if-else sont plus adaptées pour gérer des décisions complexes avec plusieurs branches de conditions ou nécessitant des blocs de code substantiels. L'opérateur ternaire, en revanche, est idéal pour des décisions simples, surtout quand il s'agit d'assigner une valeur à une variable basée sur une condition.

Gestion des dates : DateTime



- **Définition**

- La classe DateTime en C# est utilisée pour représenter des dates et des heures. Elle est essentielle pour toute application qui doit gérer des informations temporelles, comme des horodatages, des échéances, ou des calculs basés sur le temps.

- **Initialisation à partir de l'année, du mois et du jour**
 - Vous pouvez créer un DateTime en spécifiant l'année, le mois et le jour. Vous pouvez aussi spécifier l'heure, les minutes, les secondes et les millisecondes.
- **Utilisation de la date et l'heure actuelles:**
 - La propriété DateTime.Now renvoie la date et l'heure actuelles.
 - La propriété DateTime.Today renvoie la date actuelle avec l'heure définie à 00:00:00.

```
// Initialisation en spécifiant année, mois,
// jour
DateTime date1 = new DateTime(2024, 9, 3);
// 3 septembre 2024
DateTime date2 = new DateTime(2024, 9, 3, 14,
30, 0);
// 3 septembre 2024, 14:30:00

// Date actuelle
DateTime now = DateTime.Now;
// Date et heure actuelles
DateTime today = DateTime.Today;
// Date actuelle, 00:00:00
```

- Convertir une chaîne de caractères en DateTime est une opération courante, notamment pour traiter des entrées utilisateur ou des données provenant de fichiers.
- **Méthodes**
 - DateTime.Parse()
 - DateTime.TryParse()
 - DateTime.ParseExact() : convertit une chaîne de caractères en DateTime, mais elle nécessite que la chaîne corresponde exactement à un format de date spécifié. Si la chaîne n'est pas dans ce format exact, une exception est levée.

```
// Transformation à l'aide de Parse()
string dateString = "2024-09-03";
DateTime date = DateTime.Parse(dateString);

// Transformation à l'aide de TryParse()
string dateString = "2024-09-03";
DateTime date;
bool success = DateTime.TryParse(dateString,
out date);

// Transformation à l'aide de ParseExact()
string dateString = "03/09/2024";
string format = "dd/MM/yyyy";
DateTime date =
DateTime.ParseExact(dateString, format, null);
```

```
// Accès aux composants de la date et de l'heure
DateTime date = new DateTime(2024, 9, 3, 14, 30, 0);
int year = date.Year; // 2024
int month = date.Month; // 9
int day = date.Day; // 3
int hour = date.Hour; // 14
int minute = date.Minute; // 30
int second = date.Second; // 0

// Addition et soustraction de temps
DateTime date = DateTime.Now;
DateTime tomorrow = date.AddDays(1); // Ajoute un jour
DateTime nextHour = date.AddHours(1); // Ajoute une heure
DateTime lastWeek = date.AddDays(-7); // Soustrait une semaine
```

```
// Comparaison de dates
/*
Vous pouvez comparer deux objets DateTime en utilisant les opérateurs de comparaison
(<, >, <=, >=, ==, !=) ou les méthodes comme CompareTo
*/
DateTime date1 = new DateTime(2024, 9, 3);
DateTime date2 = DateTime.Today;
if (date1 > date2) {
    Console.WriteLine("date1 est dans le futur par rapport à date2");
}
int comparison = date1.CompareTo(date2);
// Renvoie -1 si date1 < date2, 0 si égaux, 1 si date1 > date2

// Formatage de DateTime en string
// Format disponible :
// https://learn.microsoft.com/fr-fr/dotnet/standard/base-types/custom-date-and-time-format-strings
DateTime date = DateTime.Now;
string dateString = date.ToString("dd/MM/yyyy HH:mm:ss"); // Format personnalisé
```

- Ecrire la fonction factorielle (dit $n!$) qui se calcule comme suit :

$$n! = \begin{cases} 1 \times 2 \times \dots \times (n-1) \times (n), & \text{si } n \geq 1 \\ 1, & \text{si } n = 0 \end{cases}$$

- Ecrire un programme qui affiche à la console la valeur du nombre de Neper e avec la plus grande précision informatique possible.

$$e = \sum_{i=0}^{+\infty} \frac{1}{i!}$$

- Ecrire la suite de Fibonacci qui se calcule comme suit :

$$\begin{aligned} u(0) &= 0, u(1) = 1, \\ u(n) &= u(n-1) + u(n-2) \quad \forall n \geq 2 \end{aligned}$$

Ecrire un programme qui simule un échantillon de taille N de nombres aléatoires uniformes compris entre 0 et 1. Hypothèse : Les N nombres de l'échantillon sont distincts deux à deux.

1. Le nombre N est choisi par un utilisateur à la console.
2. Afficher les valeurs de l'échantillon
3. Calculer et afficher les valeurs max et min de l'échantillon
4. Calculer et afficher les statistiques élémentaires de cet échantillon : Le min, le max, la moyenne, l'écart type, la médiane ainsi que le quantile d'ordre p

Ecrire un programme qui simule le tirage d'un échantillon de taille N de nombres aléatoires Gaussiens centrés réduits (moyenne 0 et volatilité 1).

Pour simuler une Gaussienne centrée, réduite on utilise l'algorithme suivant :

On considère U_1 et U_2 deux variables aléatoires uniformes qui prennent leurs valeurs dans l'intervalle $[0,1]$:

Tirer deux valeurs indépendantes uniforme u_1 et u_2 entre 0 et 1, puis calculer $\sqrt{-2 \times \log(u_1) \times \cos(2\pi \times u_2)}$. Vérifier les résultats par rapport à une loi normale (faire un graphique par bucket et afficher dans la console une visualisation)

On se propose de trouver le zéro d'une fonction régulière continue dérivable dans \mathbb{R} du type $f(y)=K$ à l'aide de l'algorithme itératif suivant :

1/ Choisir $K \in [-1, +\infty]$

2/ Choisir une précision de l'erreur ϵ

3/ Calculer pour chaque itération n , $y_n = y_{n-1} + \frac{K - f(y_{n-1})}{f'(y_{n-1})}$ jusqu'à que $|y_n - y_{n-1}| \leq \epsilon$

Tester l'algorithme pour $f(x) = \sin(x) - x^4$ (rappel $f'(x) = 4x^3 + \cos(x)$)