

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 (70%)**

- 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.

- **Contrôle sur table (30%)**

- 40% QCM
- 60% Questions ouvertes (écriture de code, débogage, ...)

Les bases du C#

Module 1



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

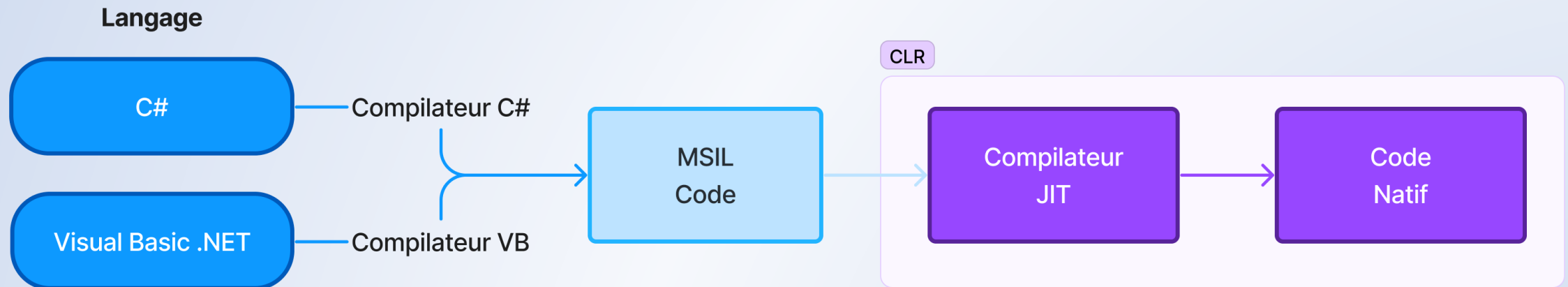


- **Genèse du C#**

- 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.

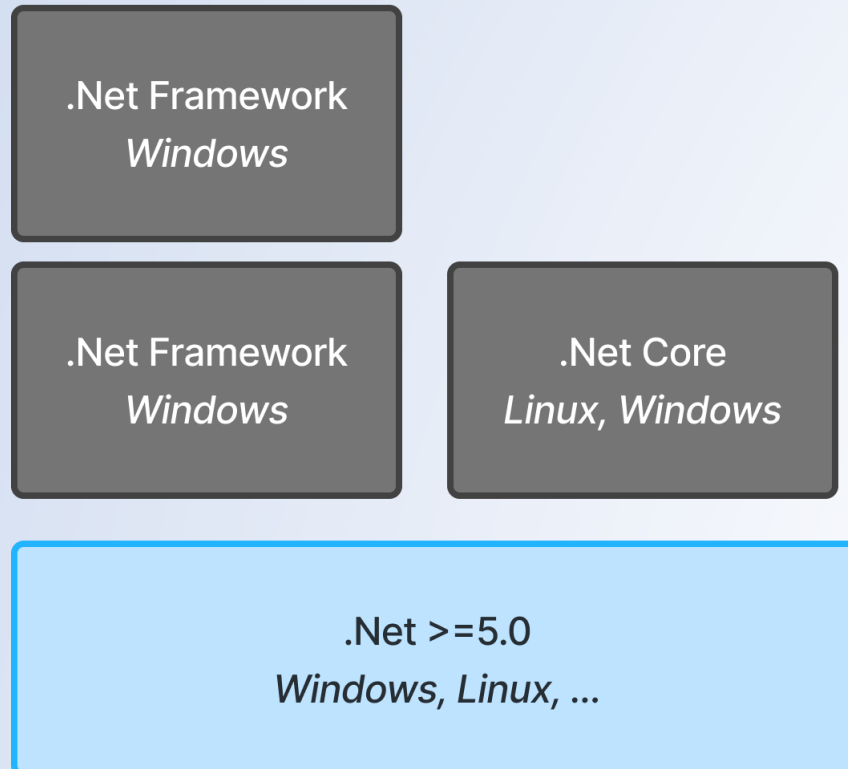


Evolution du C#

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#**

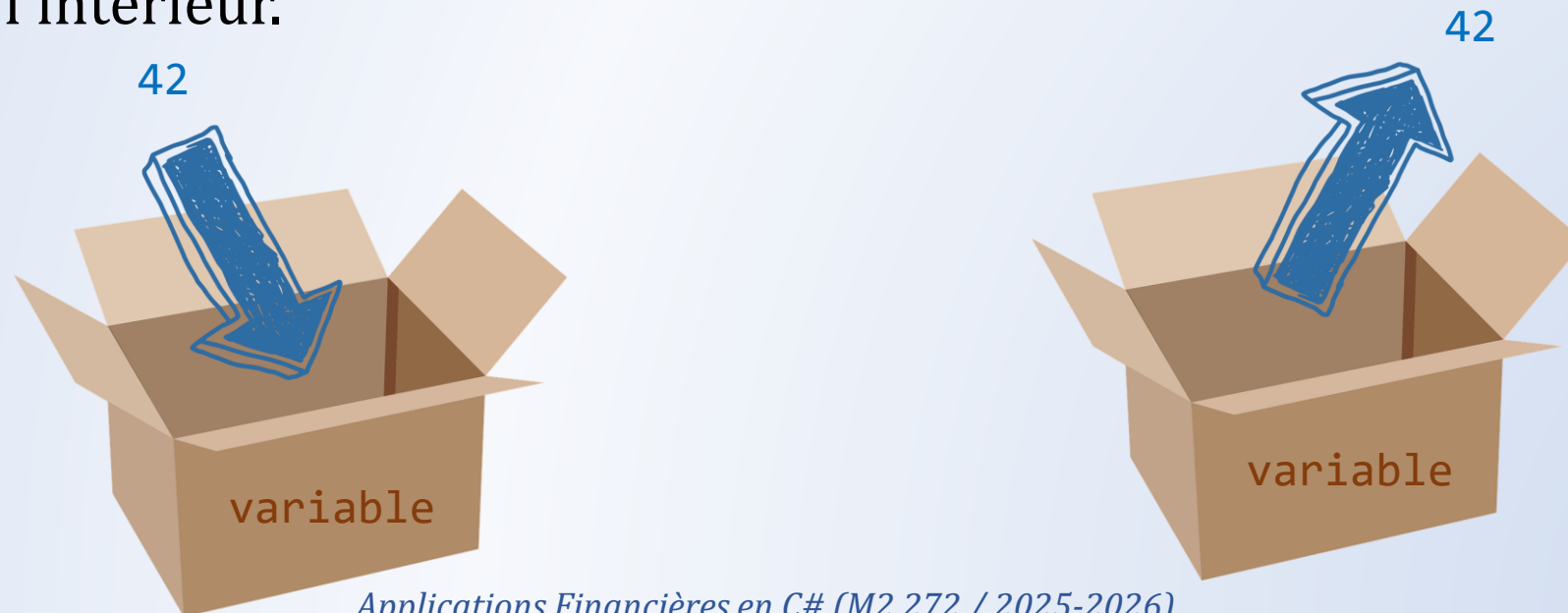
- C# est un langage de programmation orienté objet, **orienté composant** = définition des types et de leur comportement
- **Garbage Collection** = récupération automatique de la mémoire occupée par des objets inutilisés inaccessibles
- **Type nullable** = protection contre les variables qui ne font 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).

Exercice 1

36

- Voir la feuille d'exercice.

- **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
}
```

```
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;
}
```

```
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.


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

- Ecrire dans une console les résultats de la valeur actualisée pour 1€ pour les maturités allant de 1% à 10%, pour les maturités allant de 1 an à 20 ans.
- La valeur associée sera donc $(1 + r)^{-T}$.
- On s'attend à un format du type :

Taux en % ; Maturité en année ; Valeur actualisée

- La suite de Fibonacci s'écrit comme suit :
- $u(0)=0, u(1)=1,$
- $u(n)=u(n-1)+u(n-2) \forall n \geq 2$

Calculer à l'aide d'une boucle $u(30)$.

Attention: au-delà de 48, le `int` n'est plus suffisant (les nombres deviennent négatif à cause du grand nombre, il faut passer en `long`)

- La suite de Fibonacci s'écrit comme suit :
- $u(0)=0, u(1)=1,$
- $u(n)=u(n-1)+u(n-2) \forall n \geq 2$

Calculer à l'aide d'une boucle $u(30)$ à l'aide d'une fonction récursive.

Attention: au-delà de 48, le `int` n'est plus suffisant (les nombres deviennent négatif à cause du grand nombre, il faut passer en `long`)

```
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.

```
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.

```
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.
 - Nous verrons un exemple pendant la création de Liste / Tableau

- **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 (*déjà vu !*)
- **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

62

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

```
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
```

Structures de données « complexes »

Module 2



- **Définition:**

- Les collections sont des structures de données dynamiques qui peuvent contenir un ensemble d'objets.

- **Pourquoi les utiliser ?**

- Contrairement aux tableaux, les collections peuvent grandir et rétrécir dynamiquement, offrant plus de flexibilité dans la gestion des données.

Listes



- **Description :**

- Permet de stocker une liste d'objets, accessible par index.

- **Utilisation typique :**

- Quand vous avez besoin d'une collection ordonnée, avec la possibilité d'ajouter ou de supprimer des éléments facilement.


```
// Création de la liste
List<string> fruits = new List<string>();

// Ajout d'éléments à la liste
fruits.Add("Pomme");
fruits.Add("Banane");
fruits.Add("Orange");

// Parcours et affichage des éléments de la liste
Console.WriteLine("Liste de fruits:");
foreach (var fruit in fruits)
{
    Console.WriteLine(fruit);
}
```

Dictionnaires



- **Description :**

- Associe des clés uniques à des valeurs, permettant une recherche rapide des éléments.

- **Utilisation typique :**

- Idéal pour des recherches rapides et l'accès à des éléments via une clé.
- Permet de vérifier les doublons

```
// Création du dictionnaire
Dictionary<string, int> ages = new Dictionary<string, int>();

// Ajout d'éléments au dictionnaire
ages.Add("Alice", 30);
ages.Add("Bob", 25);
ages.Add("Charlie", 28);

// Accès à une valeur du dictionnaire en utilisant la clé
Console.WriteLine($"L'âge de Bob est: {ages["Bob"]} ans");

// Modification d'une valeur associée à une clé
ages["Alice"] = 31;
```

```
// Parcours du dictionnaire avec une boucle foreach
foreach (KeyValuePair<string, int> pair in ages)
{
    Console.WriteLine($"Nom: {pair.Key}, Âge: {pair.Value}");
}

// Vérification de l'existence d'une clé dans le dictionnaire
if (ages.ContainsKey("Diana"))
{
    Console.WriteLine("Diana est dans le dictionnaire.");
}

// Suppression d'une paire clé-valeur
ages.Remove("Charlie");
```

- Ecrire dans une console les résultats de la valeur actualisée pour 1€ pour les maturités allant de 1% à 10%, pour les maturités allant de 1 an à 20 ans.
- La valeur associée sera donc $(1 + r)^{-T}$.
- *On s'attend maintenant à ce que le résultat soit restitué dans des dictionnaires imbriqués de taux -> maturité -> valeur*

Ensembles



- **Description :**

- Collection qui ne permet pas les doublons et fournit des opérations rapides de recherche, d'ajout et de suppression.

- **Utilisation typique :**

- Quand vous avez besoin d'assurer l'unicité des éléments et d'effectuer des opérations de set efficaces.


```
// Création d'un HashSet de strings
HashSet<string> fruits = new HashSet<string>();

// Ajout d'éléments
fruits.Add("Pomme");
fruits.Add("Banane");
fruits.Add("Fraise");
fruits.Add("Pomme"); // Cet élément ne sera pas ajouté car il est déjà présent

// Vérification de la présence d'un élément
if (fruits.Contains("Banane"))
{
    Console.WriteLine("La banane est présente dans l'ensemble.");
}

// Affichage de tous les éléments
Console.WriteLine("Liste des fruits dans l'ensemble :");
foreach (var fruit in fruits)
{
    Console.WriteLine(fruit);
}
```

Vecteurs et matrices



- **Description :**

- Les vecteurs et les matrices sont des concepts fondamentaux en mathématiques, qui trouvent également une large application en informatique et en programmation.
- Ils sont essentiels dans de nombreux domaines, tels que le traitement de données, l'intelligence artificielle, la modélisation 3D, et la résolution de systèmes d'équations linéaires.

- **Description :**

- Un vecteur est simplement une collection ordonnée de nombres (ou d'autres types de données). En C#, un vecteur peut être représenté par un tableau unidimensionnel (array) ou une liste (List<T>).

- **Déclaration d'un vecteur**

```
// Vecteur de 3 éléments
double[] vecteur = new double[3];

// Initialisation avec des valeurs
double[] vecteur = { 1.0, 2.0, 3.0 };
```

```
// Accès aux Éléments d'un Vecteur
// Les éléments d'un vecteur sont accessibles par leur index, commençant à 0.
double firstElement = vecteur[0]; // Accès au premier élément
vecteur[1] = 5.0; // Modification du deuxième élément

// Exemple : Calcul du Produit Scalaire
double[] vecteurA = { 1.0, 2.0, 3.0 };
double[] vecteurB = { 4.0, 5.0, 6.0 };
double produitScalaire = 0;
for (int i = 0; i < vecteurA.Length; i++)
{
    produitScalaire += vecteurA[i] * vecteurB[i];
}
Console.WriteLine("Produit Scalaire: " + produitScalaire); // Affiche 32.0
```

- **Description :**

- Une matrice est une collection bidimensionnelle de nombres.
- En C#, une matrice est typiquement représentée par un tableau à deux dimensions (2D array).

- **Déclaration d'une matrice**

```
// Matrice 3x3 de double
double[,] matrice = new double[3, 3];

// Initialisation avec des valeurs
double[,] matrice = {
    { 1.0, 2.0, 3.0 },
    { 4.0, 5.0, 6.0 },
    { 7.0, 8.0, 9.0 }
};
```

Exemple – Boucle for

91

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

Définition d'une matrice

92

```
// Accès aux Éléments d'une Matrice
// Les éléments d'une matrice sont accessibles par deux indices : un pour la ligne et un pour la colonne
double element = matrice[0, 1]; // Accès à l'élément de la première ligne, deuxième colonne
matrice[2, 2] = 10.0; // Modification de l'élément de la troisième ligne, troisième colonne

// Exemple : Multiplication de Matrices
double[,] matriceA = {
    { 1, 2 },
    { 3, 4 }
};
double[,] matriceB = {
    { 5, 6 },
    { 7, 8 }
};
double[,] produit = new double[2, 2];
for (int i = 0; i < 2; i++)
{
    for (int j = 0; j < 2; j++)
    {
        produit[i, j] = 0;
        for (int k = 0; k < 2; k++)
        {
            produit[i, j] += matriceA[i, k] * matriceB[k, j];
        }
    }
}
```


- **Description :**

- Site : <https://numerics.mathdotnet.com/>
- Math.NET Numerics vise à fournir des méthodes et des algorithmes pour les calculs numériques dans les domaines de la science, de l'ingénierie et de l'utilisation quotidienne. Les sujets couverts incluent les fonctions spéciales, l'algèbre linéaire, les modèles de probabilité, les nombres aléatoires, l'interpolation, l'intégration, la régression, les problèmes d'optimisation et bien plus encore.
- Découvrons comment installer un NUGET

LINQ (*Language Integrated Query*)



- **Description :**

- LINQ (Language Integrated Query) est une puissante fonctionnalité de C# qui permet de formuler des requêtes complexes directement dans le code, en utilisant des expressions semblables à SQL.
- LINQ peut être utilisé pour interroger diverses sources de données telles que des collections en mémoire, des bases de données, des fichiers XML, et bien plus encore.
- L'un des principaux avantages de LINQ est qu'il offre une syntaxe cohérente et unifiée pour accéder aux données, quel que soit leur type de source.

- **Source de données LINQ :**

- LINQ permet de faire des requêtes sur des collections d'objets (par exemple, des tableaux, des listes) ou des bases de données en utilisant des méthodes de haut niveau, tout en restant dans le contexte du langage C#. Les principales sources de données supportées par LINQ incluent :
 - **LINQ to Objects** : Pour interroger des collections en mémoire (List, Array, etc.).
 - **LINQ to SQL** : Pour interroger des bases de données SQL Server.
 - **LINQ to XML** : Pour manipuler des données XML.
 - **LINQ to Entities** : Pour travailler avec Entity Framework.

• Syntaxe de Requête

- La syntaxe de requête ressemble à une requête SQL. C'est la syntaxe la plus lisible et intuitive, surtout pour les développeurs ayant une expérience en SQL.

• Syntaxe de Méthode

- La syntaxe de méthode utilise des méthodes d'extension pour effectuer les opérations de requête. Cette syntaxe est plus compacte et souvent plus flexible, permettant l'utilisation de fonctions lambda.

```
// Vecteur initial
int[] numbers = { 2, 4, 6, 8, 10 };

// Requête LINQ en syntaxe de requête
var evenNumbers = from num in numbers
                  where num % 2 == 0
                  select num;

// Requête LINQ en syntaxe de méthode
var evenNumbers = numbers.Where(num => num % 2 == 0);
```

- **Where : Filtrer les données**

- Where est utilisé pour filtrer une collection en fonction d'une condition donnée.

```
string[] names = { "Alice", "Bob", "Charlie", "David" };

// Filtrer les noms commençant par 'A'
var filteredNames = names.Where(name => name.StartsWith("A"));

foreach (var name in filteredNames)
{
    Console.WriteLine(name); // Affiche : Alice
}
```

- **Select : Transformer les données**

- Select est utilisé pour projeter les éléments d'une collection dans une nouvelle forme.

```
int[] numbers = { 1, 2, 3, 4, 5 };

// Doubler chaque nombre
var doubledNumbers = numbers.Select(num => num * 2);

foreach (var n in doubledNumbers)
{
    Console.WriteLine(n); // Affiche : 2, 4, 6, 8, 10
}
```


- **OrderBy et OrderByDescending : Trier les données**

- **OrderBy** est utilisé pour trier les éléments d'une collection dans l'ordre croissant.
- **OrderByDescending** trie dans l'ordre décroissant.

```
string[] names = { "Alice", "Charlie", "Bob", "David" };

// Trier les noms par ordre alphabétique
var sortedNames = names.OrderBy(name => name);

foreach (var name in sortedNames)
{
    Console.WriteLine(name); // Affiche : Alice, Bob, Charlie, David
}
```


- **GroupBy : Grouper les données**

- GroupBy regroupe les éléments d'une collection en fonction d'une clé donnée.

```
string[] names = { "Alice", "Bob", "Charlie", "David", "Daniel" };

// Grouper les noms par leur première lettre
var groupedNames = names.GroupBy(name => name[0]);

foreach (var group in groupedNames)
{
    Console.WriteLine("Noms commençant par " + group.Key + ":");
    foreach (var name in group)
    {
        Console.WriteLine(name); // Affiche les noms groupés par leur première lettre
    }
}
```

- A partir de la liste suivante [10, 9, 8, 7, 6], récupérez une nouvelle liste en utilisant LINQ renvoyant uniquement pour les multiples de 2, leur valeur au carré.

- A partir de cette même liste [10, 9, 8, 7, 6], regroupez par si élément pair ou non, et renvoyez sous la forme d'un dictionnaire : true/false → $\text{moyenne}(x*x)$

Le développement de classes en C#

Module 3

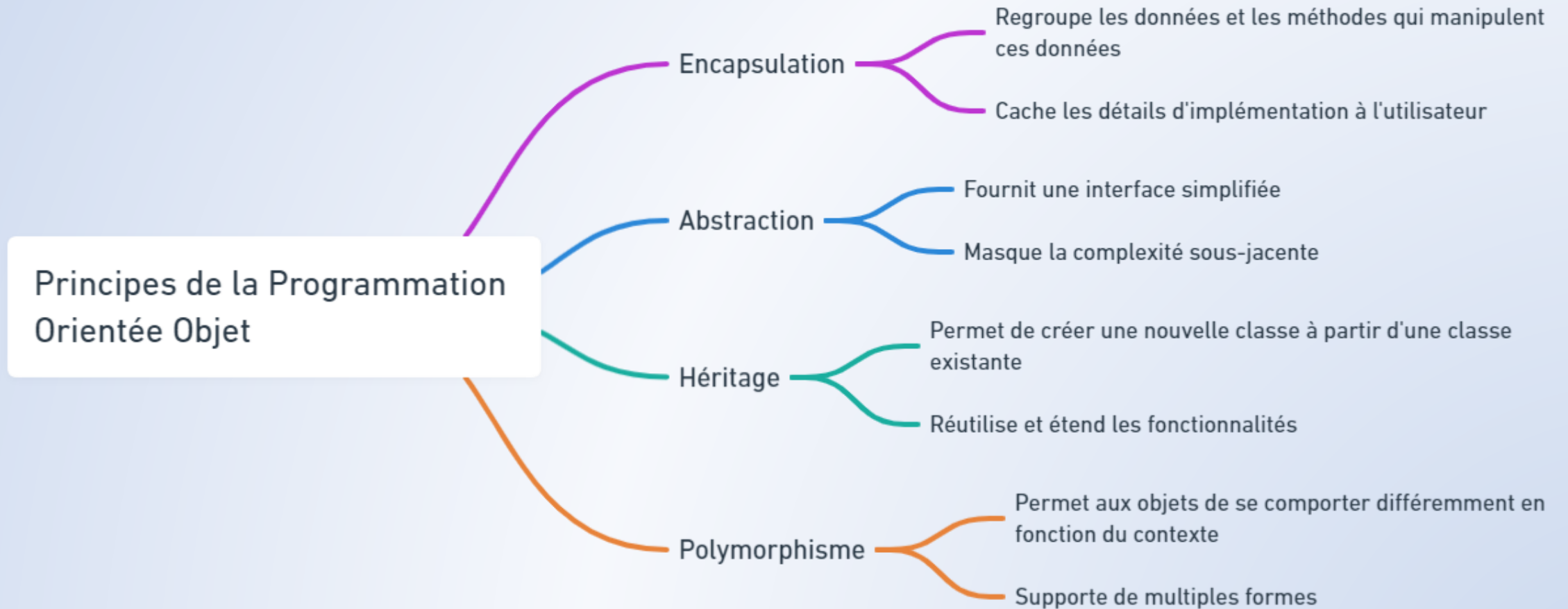


Introduction à la Programmation Orientée Objet (P00)



- **Définition**

- **Paradigme de programmation:** Programmation orientée objet (POO)
- **Basée sur:** Le concept d'"objets"
- **Contenu des objets:**
 - **Données:** Sous forme d'attributs ou de propriétés
 - **Code:** Sous forme de procédures, souvent appelées méthodes
- **Objectif de la POO:** Structurer les programmes
- **Méthode:** Regrouper les données et les opérations qui les manipulent



- **Principes**

- **Encapsulation:**

- Regroupe données (attributs) et méthodes dans une "classe".
 - Les données sont accessibles uniquement via les méthodes de l'objet.
 - Préviend les accès non autorisés et modifications accidentelles.

- **Abstraction:**

- Crée des modèles simplifiés pour représenter des concepts complexes.
 - Utilisation de classes abstraites et interfaces pour définir des modèles de base.
 - Spécifie les méthodes à implémenter par les classes dérivées.

- **Principes**

- **Héritage:**

- Permet à une classe "fille" d'hériter attributs et méthodes d'une classe "mère".
 - Possibilité d'ajouter ou modifier attributs et méthodes hérités.
 - Facilite la réutilisation, extension, et modification du comportement des classes existantes.

- **Polymorphisme:**

- Permet aux entités de se comporter différemment selon le contexte.
 - Une méthode peut avoir plusieurs formes d'implémentation.
 - Facilite la redéfinition de méthodes dans des classes dérivées.
 - Permet un traitement uniforme d'objets de types différents.

- **Avantages**

- **Modularité:**

- Création de modules indépendants pour la construction de systèmes complexes.
 - Facilite la maintenance et la mise à jour des systèmes.

- **Réutilisabilité:**

- Utilisation de l'héritage et du polymorphisme pour réutiliser des classes existantes.
 - Réduction du temps de développement et augmentation de la fiabilité du code.

- **Scalabilité:**

- Facilité d'extension des systèmes par l'ajout de nouvelles classes ou par héritage.
 - Pas besoin de modifier significativement le code existant pour étendre les fonctionnalités.

- **Maintenabilité:**

- Code plus facile à tester, débbugger, et maintenir grâce à une structure mieux organisée.
 - L'encapsulation prévient les effets de bord, améliorant la qualité du code.

- **Classe :**

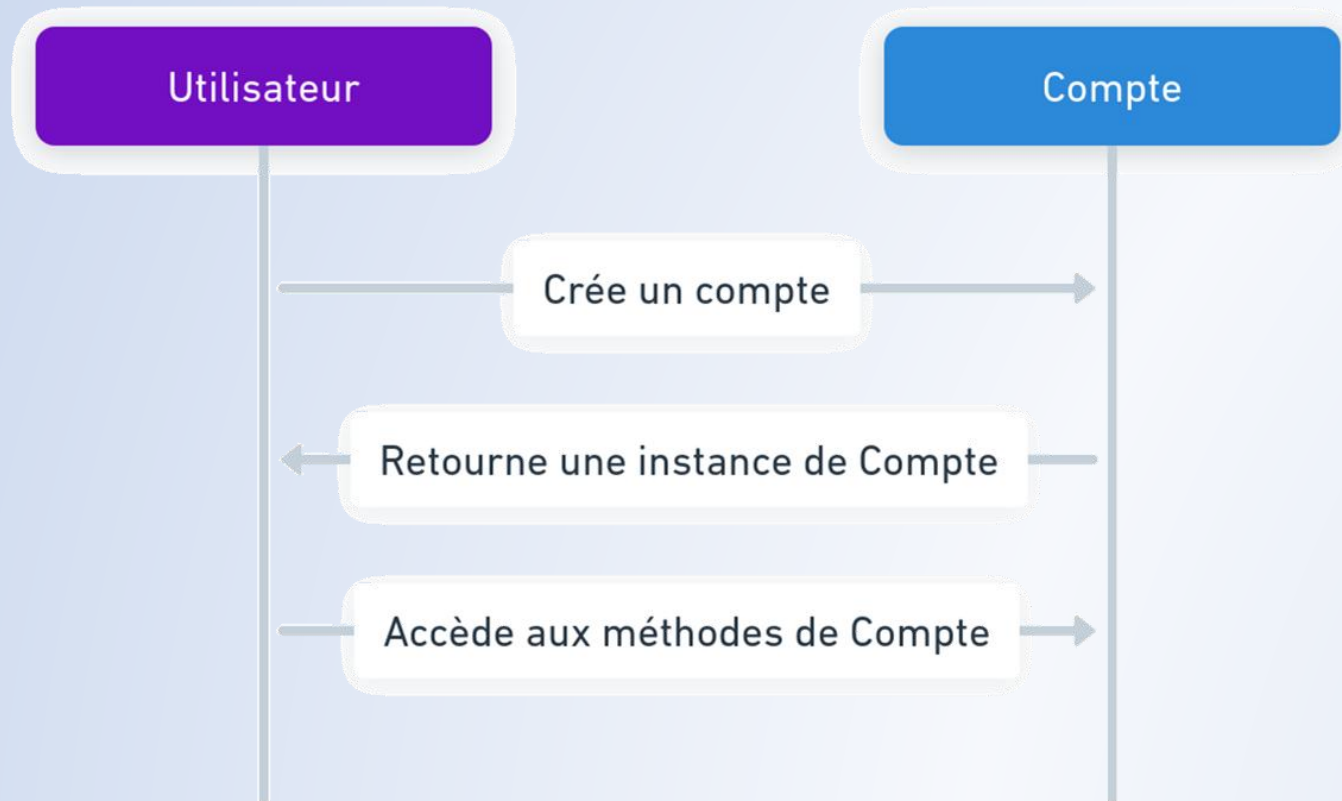
- Une classe est un "plan" ou un "modèle" qui définit les caractéristiques et les comportements (sous forme de propriétés et de méthodes) que posséderont les objets créés à partir de ce modèle. En somme, c'est comme une recette qui décrit comment faire un gâteau, quels ingrédients utiliser et les étapes à suivre.

- **Objet :**

- Un objet est une instance d'une classe. Si la classe est la recette, alors l'objet est le gâteau que vous avez fait en suivant cette recette. Chaque objet a son propre état et peut comporter des valeurs différentes pour ses propriétés, bien qu'il partage la même structure et les mêmes comportements (méthodes) que définis dans la classe.

- **Création d'objets :**

- Pour créer un objet en C#, on utilise le mot-clé `new` suivi du nom de la classe et de parenthèses. Par exemple, `Voiture maVoiture = new Voiture();` crée un nouvel objet `maVoiture` de la classe `Voiture`.



Voici le diagramme de séquence illustrant l'interaction entre une classe **Utilisateur** et une classe **Compte** dans un langage orienté objet comme C#.

Ce diagramme montre comment une instance de **Utilisateur** peut créer un objet **Compte**, recevoir une instance de ce compte, et ensuite accéder à ses méthodes.

```
public class Compte{}

class Program
{
    static void Main()
    {
        // Création d'un nouvel objet Compte
        Compte monCompte = new Compte();
    }
}
```

- **Champs :**

- Les champs sont les variables internes de la classe qui contiennent l'état de l'objet. Pour un compte bancaire, les champs typiques pourraient inclure le solde du compte, le nom du titulaire, et peut-être un identifiant de compte. Ces champs sont généralement privés pour préserver l'encapsulation.

- **Propriétés :**

- Les propriétés fournissent un accès contrôlé aux champs. Elles permettent de lire ou de modifier les valeurs tout en implémentant des contrôles ou des validations spécifiques. Par exemple, on pourrait vouloir s'assurer que le solde du compte ne peut pas être défini à une valeur négative directement.

- **Méthodes :**

- Les méthodes définissent les actions que l'objet peut exécuter. Pour un compte, cela pourrait inclure déposer de l'argent, retirer de l'argent, ou afficher le solde actuel.


```
public class Compte
{
    // Champs
    private string titulaire;
    private decimal solde;
    private int compteId;

    // Propriétés
    public string Titulaire
    {
        get { return titulaire; }
        set { titulaire = value; }
    }

    public decimal Solde
    {
        get { return solde; }
        private set { solde = value; } // Le solde ne peut être modifié que par les méthodes
    }
}
```

```
public class Compte
{
    // (...)
    public int CompteId
    {
        get { return compteId; }
        set { compteId = value; }
    }

    // Méthodes
    public void Deposer(decimal montant)
    {
        if (montant > 0)
        {
            Solde += montant;
            Console.WriteLine($"Dépôt de {montant} réussi. Nouveau solde: {Solde}.");
        }
        else
        {
            Console.WriteLine("Le montant à déposer doit être positif.");
        }
    }
}
```



```
public class Compte
{
    // (...)
    public bool Retirer(decimal montant)
    {
        if (montant <= Solde)
        {
            Solde -= montant;
            Console.WriteLine($"Retrait de {montant} réussi. Nouveau solde: {Solde}.");
            return true;
        }
        else
        {
            Console.WriteLine("Fonds insuffisants pour ce retrait.");
            return false;
        }
    }

    public void AfficherSolde()
    {
        Console.WriteLine($"Le solde du compte est de {Solde}.");
    }
}
```

- **Constructeurs :**

- Un constructeur est une méthode spéciale qui est appelée automatiquement lorsqu'un objet est créé. Il sert à initialiser l'objet, en donnant des valeurs initiales aux champs par exemple. Les constructeurs ont le même nom que la classe.

- **Destructeurs :**

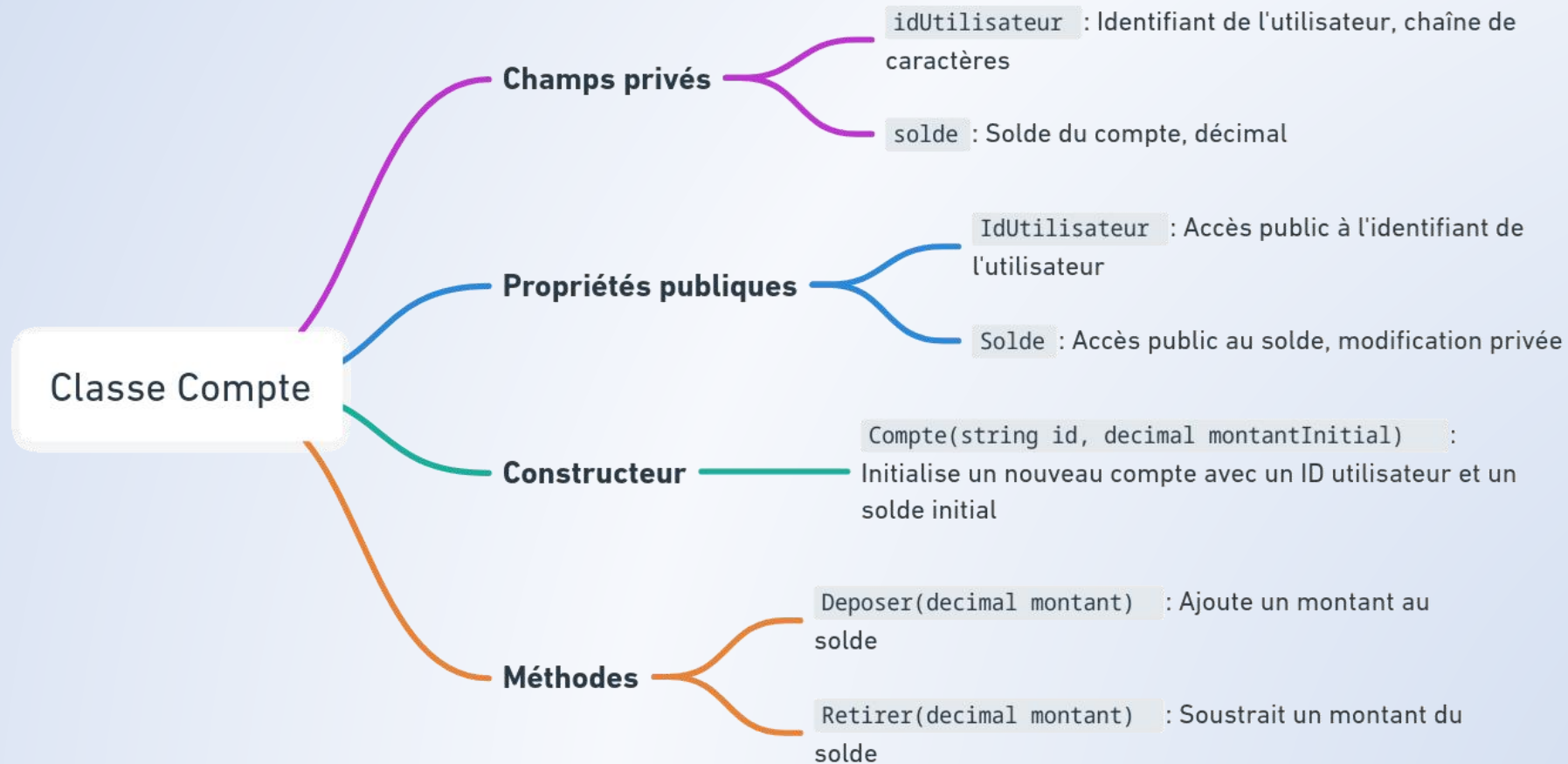
- Un destructeur est une méthode qui est appelée automatiquement lorsque l'objet est détruit ou que sa portée se termine. En C#, les destructeurs sont moins couramment utilisés car le ramasse-miettes (garbage collector) s'occupe de la gestion de la mémoire.

```
public class Compte
{
    // (...)
    // Constructeur
    public Compte(string titulaire, int compteId)
    {
        this.Titulaire = titulaire;
        this.Solde = 0; // Initialiser le solde à zéro
        this.CompteId = compteId;
    }

    // Destructeur
    ~Compte()
    {
        // Code pour nettoyer les ressources non gérées si nécessaire
        Console.WriteLine($"Le compte {CompteId} de {Titulaire} est en train d'être détruit.");
    }
}
```

Exemple complet de la classe Compte

123



- **Définition :**

- L'encapsulation est un principe fondamental de la programmation orientée objet (POO) qui consiste à restreindre l'accès aux composants d'un objet.
- Cela permet de protéger l'intégrité des données en empêchant l'accès direct aux champs internes d'une classe et en offrant une manière contrôlée de les modifier ou de les obtenir.

- **Définition :**

- Les accesseurs (getters) et mutateurs (setters) sont des méthodes spéciales qui permettent respectivement de lire et de modifier les valeurs des champs privés d'une classe.

- **Getters :**

- Permettent de récupérer la valeur d'un champ. Par exemple, `public string Titulaire { get { return titulaire; } }` permet de récupérer le nom du titulaire du compte.

- **Setters :**

- Permettent de définir la valeur d'un champ. Par exemple, `public string Titulaire { set { titulaire = value; } }` permet de modifier le nom du titulaire. Notez que pour le solde, le setter est marqué comme `private`, ce qui signifie que la valeur du solde ne peut être modifiée que par les méthodes internes de la classe `Compte`, telles que `Deposer` et `Retirer`.

- **Définition :**

- Les propriétés en C# sont une manière élégante de mettre en œuvre les accesseurs et les mutateurs. Elles ressemblent à des champs mais agissent comme des méthodes. Notre classe Compte utilise des propriétés pour exposer Titulaire, Solde et Compteld :
- Titulaire et Compteld ont des accesseurs et des mutateurs publics, ce qui signifie qu'ils peuvent être lus et modifiés de l'extérieur de la classe.
- Solde a un accesseur public mais un mutateur privé, permettant de lire le solde à l'extérieur de la classe tout en restreignant sa modification à des méthodes internes.

- **Définition :**

- Les niveaux d'accès définissent comment les membres d'une classe (champs, propriétés, méthodes) peuvent être accédés.
 - **public** : Accessible de n'importe où, aucunement restreint.
 - **private** : Accessible uniquement à l'intérieur de la classe où il est déclaré.
 - **protected** : Accessible dans la classe où il est déclaré ainsi que dans toutes les classes dérivées.
 - **internal** : Accessible uniquement à l'intérieur du même assembly (projet).

Héritage et Polymorphisme



- **Définition :**

- L'héritage est un principe fondamental de la programmation orientée objet (POO) qui permet à une classe de hériter des champs, propriétés, et méthodes d'une autre classe. Cela favorise la réutilisation du code et l'organisation hiérarchique des classes.

- **Concept :**

- L'héritage permet à une classe (dite "classe dérivée") d'hériter d'une autre classe (dite "classe de base"). En C#, l'héritage s'effectue en utilisant le symbole : suivant le nom de la classe dérivée.

```
public class ClasseDeBase
{
    // Définition de la classe de base
}

public class ClasseDerivee : ClasseDeBase
{
    // Définition de la classe dérivée qui hérite de ClasseDeBase
}
```

- **Définition :**

- L'héritage simple signifie qu'une classe dérive d'une seule classe de base. C# ne supporte que l'héritage simple directement, ce qui signifie qu'une classe ne peut hériter que d'une seule autre classe.

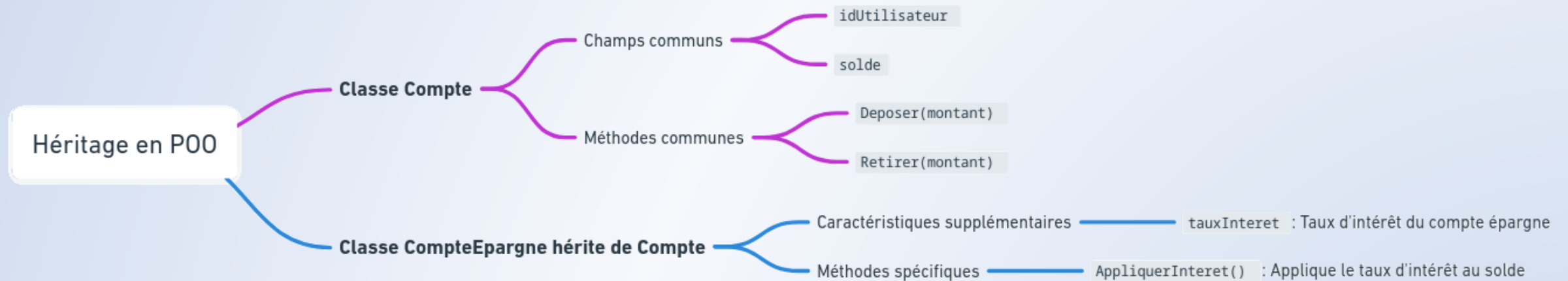
- **Exemple :**

- Par exemple, supposons que vous ayez une classe CompteEpargne qui est un type spécial de Compte. CompteEpargne hérite de Compte pour réutiliser les champs et méthodes existants tout en ajoutant ses propres fonctionnalités spécifiques.

```
public class CompteEpargne : Compte
{
    private decimal tauxInteret;

    public CompteEpargne(string titulaire, int compteId, decimal tauxInteret) : base(titulaire, compteId)
    {
        this.tauxInteret = tauxInteret;
    }

    public void AppliquerInteret()
    {
        // Calcule l'intérêt et l'ajoute au solde
        decimal interet = Solde * tauxInteret / 100;
        Deposer(interet);
        Console.WriteLine($"Intérêt de {interet} appliqué au solde.");
    }
}
```



- **Surcharge de méthodes** se produit lorsqu'on crée plusieurs méthodes avec le même nom mais des paramètres différents dans la même classe. C'est une façon d'offrir différentes variantes d'une méthode.
- **Redéfinition de méthodes** (ou polymorphisme d'héritage) permet à une classe dérivée de fournir une implémentation spécifique d'une méthode héritée de la classe de base. En C#, vous utilisez les mots-clés `virtual` dans la classe de base pour indiquer qu'une méthode peut être redéfinie et `override` dans la classe dérivée pour effectivement la redéfinir.

```
public class Compte
{
    // Existant déjà dans la classe Compte
    public virtual void AfficherSolde()
    {
        Console.WriteLine($"Le solde du compte est de {Solde}.");
    }
}

public class CompteEpargne : Compte
{
    // Redéfinition de la méthode AfficherSolde
    public override void AfficherSolde()
    {
        Console.WriteLine($"Le solde du compte épargne est de {Solde}. Intérêt: {tauxInteret}%");
    }
}
```


- **Définition :**

- Le polymorphisme est un principe de la programmation orientée objet qui permet aux objets de se comporter différemment selon leur type ou leur classe.
- En C#, il existe deux types principaux de polymorphisme : statique et dynamique.

- **Polymorphisme statique**

- se manifeste principalement par la surcharge de méthodes. Cela signifie que plusieurs méthodes peuvent avoir le même nom mais différer par leur signature, c'est-à-dire le nombre ou le type de leurs paramètres. Cela permet d'utiliser la même méthode de différentes manières en fonction des arguments fournis.

- **Polymorphisme dynamique**

- est réalisé grâce à la surdéfinition de méthodes (override) et l'utilisation de méthodes virtuelles. Il permet à une méthode d'une classe de base d'être redéfinie dans une classe dérivée pour adopter un comportement spécifique à cette classe.

Polymorphisme en POO

Polymorphisme Statique

Exemple: Surcharge de méthodes

`Deposer(decimal montant)`

`Deposer(decimal montant, string devise)`

Se produit au moment de la compilation

Polymorphisme Dynamique

Exemple: Redéfinition de méthodes

Classe de base `Compte`

Méthode `AfficherInfos()`

Classe dérivée `CompteEpargne`

Redéfinition de `AfficherInfos()`

Utilisation de références de classe de base pour accéder à des objets de classes dérivées

Se produit au moment de l'exécution

```
public class Cheque
{
    // Champs privés
    private decimal montant;

    // Propriétés publiques
    public decimal Montant
    {
        get { return montant; }
        set { montant = value; }
    }

    // Constructeur
    public Cheque(decimal montant)
    {
        this.Montant = montant;
    }
}
```

```
public void Deposer(Cheque cheque)
{
    decimal montant = cheque.Montant;
    // Logique pour déposer un chèque
    Console.WriteLine($"Dépôt de chèque de
{montant} réussi.");
}
```

Imaginons que dans notre classe Compte, nous voulions ajouter une fonctionnalité pour déposer des chèques, en plus des dépôts en espèces. Nous pourrions surcharger la méthode Deposer pour accepter un type différent de paramètre pour les chèques.

- Dans notre exemple de Compte et CompteEpargne, nous avons déjà une illustration du polymorphisme dynamique avec la méthode AfficherSolde.
- La méthode AfficherSolde dans Compte est marquée comme virtual, ce qui signifie qu'elle peut être redéfinie dans une classe dérivée.
- Dans CompteEpargne, nous redéfinissons AfficherSolde pour afficher non seulement le solde mais aussi le taux d'intérêt. C'est un exemple classique de polymorphisme dynamique, où le comportement de la méthode varie selon le type d'objet (Compte ou CompteEpargne) qui l'invoque.
- Utilisation dans notre exemple
- Compte classique : Lorsqu'on appelle AfficherSolde sur un objet de type Compte, il affiche simplement le solde.
- Compte épargne : Lorsqu'on appelle AfficherSolde sur un objet de type CompteEpargne, il affiche le solde et le taux d'intérêt, grâce à la redéfinition de la méthode.

Classes Abstraites et Interfaces



- **Définition**

- L'abstraction est un concept fondamental en programmation orientée objet (POO) qui permet de définir la structure d'une classe tout en masquant certains détails de son implémentation.
- En C#, l'abstraction peut être réalisée à l'aide de classes abstraites.
- Une classe abstraite sert de modèle pour d'autres classes.
- Elle peut contenir des méthodes abstraites (sans corps) ainsi que des méthodes concrètes (avec un corps).

- **Classe abstraite**

- Une classe abstraite est déclarée avec le mot-clé `abstract`.
- Elle ne peut pas être instanciée directement, ce qui signifie que vous ne pouvez pas créer d'objets de cette classe.
- Son utilisation principale est de servir de base pour d'autres classes qui héritent d'elle et fournissent des implémentations pour les méthodes abstraites.


```
public abstract class CompteBase
{
    // Champs communs
    protected string titulaire;
    protected decimal solde;
    protected int compteId;

    // Propriétés communes
    public string Titulaire { get; set; }
    public decimal Solde { get; protected set; }
    public int CompteId { get; set; }

    // Constructeur
    public CompteBase(string titulaire, int compteId)
    {
        this.Titulaire = titulaire;
        this.Solde = 0;
        this.CompteId = compteId;
    }
}
```

```
// Méthodes communes
public abstract void Deposer(decimal montant);
public abstract bool Retirer(decimal montant);

// Méthode virtuelle pouvant être redéfinie
public virtual void AfficherSolde()
{
    Console.WriteLine($"Le solde du compte est de
{Solde}.");
}
}
```

Prenons l'exemple de la classe Compte. Imaginez que vous voulez définir un comportement général pour différentes sortes de comptes, mais que certains comportements, comme l'application d'intérêts, diffèrent selon le type de compte. Vous pourriez définir une classe abstraite `CompteBase` qui contiendrait les éléments communs et laisserait l'implémentation spécifique de certains comportements aux classes dérivées.

- Les **classes abstraites** diffèrent des classes concrètes (non abstraites) de plusieurs façons :
 - **Instanciation** : Les classes abstraites ne peuvent pas être instanciées directement, contrairement aux classes concrètes.
 - **Contenu** : Les classes abstraites peuvent contenir des méthodes abstraites sans implémentation. Toutes les méthodes dans une classe concrète doivent avoir une implémentation.
 - **Objectif** : Les classes abstraites sont conçues pour être des classes de base dont d'autres classes peuvent hériter. Les classes concrètes sont conçues pour créer des objets.

- **Définition**

- Une interface en C# est comme un contrat ou une promesse. Elle définit un ensemble de méthodes et de propriétés que toute classe qui implémente l'interface doit fournir.
- Contrairement à une classe, une interface ne contient pas de logique de mise en œuvre; elle spécifie simplement le "quoi" mais pas le "comment".
- Cela signifie qu'elle énonce les actions possibles sans définir comment ces actions sont réalisées.

```
public interface IOperationsBancaires
{
    void Deposer(decimal montant);
    bool Retirer(decimal montant);
    void AfficherSolde();
}
```

Imaginez que nous voulons standardiser les opérations bancaires pour différents types de comptes, y compris le compte standard que vous avez déjà et potentiellement d'autres types, comme un compte pour les entreprises.

Nous pourrions définir une interface `IOperationsBancaires` qui contiendrait les méthodes de base que tous les comptes doivent implémenter, comme `Deposer`, `Retirer`, et `AfficherSolde`.

Votre classe `Compte` et `CompteEpargne` pourraient implémenter cette interface pour garantir qu'elles fournissent des implémentations pour ces opérations.

- Les interfaces et les classes abstraites semblent similaires car elles permettent toutes deux de définir des méthodes sans implémentations. Cependant, il existe des différences clés :
 - **Classes abstraites** : Peuvent contenir une implémentation concrète (code au sein des méthodes) et des membres non publics. Une classe ne peut hériter que d'une seule classe abstraite, respectant la limitation d'héritage simple de C#.
 - **Interfaces** : Ne contiennent aucune implémentation de méthode. Tous les membres d'une interface sont implicitement publics. Une classe peut implémenter plusieurs interfaces, contournant ainsi la limitation de l'héritage simple.

- Le concept de contrat d'interface est crucial car il garantit une cohérence dans l'implémentation. Lorsqu'une classe implémente une interface, elle signe un contrat qui stipule qu'elle fournira des implémentations concrètes pour toutes les méthodes et propriétés définies dans l'interface.
- Cela assure que peu importe le type de compte bancaire (ou tout autre objet) que vous manipulez, vous pouvez vous attendre à ce qu'il ait certaines fonctionnalités de base définies par l'interface.
- **Application à notre exemple** : En faisant implémenter l'interface `IOperationsBancaires` par vos classes `Compte` et `CompteEpargne`, vous assurez que chaque type de compte peut effectuer des dépôts, des retraits, et afficher le solde. Cela rend votre code plus flexible et plus facile à maintenir, car vous pouvez introduire de nouveaux types de comptes sans changer le code qui utilise ces comptes, tant qu'ils adhèrent au contrat défini par l'interface.

- Cette application bancaire (de type console) permet de gérer les comptes d'un utilisateur.
 1. Une classe Utilisateur va exister et contenir les identifiants d'un utilisateur (id, prénom, nom, ville de naissance, adresse).
 2. Une classe abstraite Compte va exister et contenir les informations suivantes Solde, Déposer, Retirer. On va auditer les différents mouvements.
 3. Une classe CompteCourante va reprendre les éléments essentiels de Compte.

4. Une classe `CompteEpargne` va ajouter une notion d'intérêt quotidien calculé par $\text{solde} * \text{taux d'intérêt}$
5. Extension du dépôt et du retrait par devise.
6. Extension du dépôt et du retrait en ajoutant des moyens de paiement (Cash, Chèque, CB).

Type enum



• Définition

- En C#, le type enum (ou énumération) est un type de données distinct qui permet de définir un ensemble de valeurs constantes nommées.
- Les énumérations sont particulièrement utiles pour représenter des groupes de valeurs étroitement liées, telles que des jours de la semaine, des états d'une machine, ou des types de transactions.
- Elles permettent d'améliorer la lisibilité et la maintenabilité du code.

- Une énumération est déclarée à l'aide du mot-clé `enum`, suivi du nom de l'énumération et d'une liste de valeurs constantes.
- Dans cet exemple, `JourSemaine` est une énumération qui représente les jours de la semaine. Par défaut, les valeurs de l'énumération sont de type `int` et commencent à 0, puis incrémentent automatiquement.

```
enum JourSemaine
{
    Lundi,
    Mardi,
    Mercredi,
    Jeudi,
    Vendredi,
    Samedi,
    Dimanche
}
```

- Vous pouvez spécifier des valeurs personnalisées pour les éléments de l'énumération :

```
enum Mois
{
    Janvier = 1,
    Février = 2,
    Mars = 3,
    Avril = 4,
    Mai = 5
}
```

```
// Création d'une variable d'énumération
JourSemaine jour = JourSemaine.Lundi;
Console.WriteLine(jour); // Affiche : Lundi

// Utilisation dans des conditions
JourSemaine jour = JourSemaine.Jeudi;
if (jour == JourSemaine.Jeud)
{
    Console.WriteLine("C'est la meilleure journée de la semaine !");
}
```

- Soit $P(t,T)$ le prix d'une obligation payant 1 EUR en $t=T$, vu de $t=t_0$
- Mettre en place le pricing en fonction des conventions suivantes :
- (ICA) Intérêt composé annuel (aka Yield convention)

$$P(t_0, T) = \frac{1}{(1 + r(t_0, T))^{T-t_0}}$$

- (ICS) Intérêt composé avec k subdivisions de l'année

$$P(t_0, T) = \frac{1}{\left(1 + \frac{r(t_0, T)}{k}\right)^{(T-t_0) \times k}}$$

- Soit $P(t,T)$ le prix d'une obligation payant 1 EUR en $t=T$, vu de $t=t_0$
- Mettre en place le pricing en fonction des conventions suivantes : (...)
- (CC) Composition en temps continue :
$$P(t_0, T) = \exp\left(-(t_0, T) \times (T - t_0)\right)$$
- (CL) Composition linéaire : $P(t_0, T) = \frac{1}{1+r(y_0, T) \times (T - t_0)}$

- Ajoutez une définition dynamique de la maturité en années à partir de date. Les conventions à utiliser sont les suivantes :

- (MM) Money Market :

ACT/360 où ACT = nombre de jours exactes entre les deux dates

- (BB) Bond Basis :

30 / 360

$$= \frac{\max(30 - d_{1,0}) + \min(d_{2,30}) + 30 \times (m_2 - m_1 - 1) + 360 \times (\text{annee}_2 - \text{annee}_1)}{360}$$

- (DF) Discount Factors :

ACT/365

- Comparez ces résultats avec la fonction YearFrac dans Excel

Pour aller plus loin

Module 4



Conventions de nom en C#



- **Classes et Interfaces :**

- Utilisez le **PascalCase** pour les noms, c'est-à-dire que chaque mot commence par une majuscule. Par exemple, Compte, CompteEpargne, et IOperationsBancaires. Cela permet d'identifier rapidement les classes et interfaces dans le code.

- **Méthodes :**

- Comme pour les classes et interfaces, utilisez le **PascalCase**. Les noms des méthodes doivent être des verbes ou des phrases verbales, indiquant clairement ce que fait la méthode. Par exemple, Deposier, Retirer, et AfficherSolde.

- **Variables (Champs et Propriétés) :**

- Pour les champs privés, utilisez le **camelCase** précédé d'un underscore, par exemple, _titulaire, _solde, et _compteld. Cela aide à distinguer les champs internes des propriétés et méthodes publiques.
- Pour les propriétés, utilisez le **PascalCase**, reflétant la nature publique et l'accès externe à ces membres, par exemple, Titulaire, Solde, et Compteld.

- **Paramètres de Méthode :**

- Utilisez le camelCase sans underscore. Les noms doivent être descriptifs pour indiquer clairement ce qu'ils représentent, par exemple, montant dans `Deposer(decimal montant)`.

- **Interfaces :**

- Préfixez les noms avec un "I" majuscule suivi du nom en PascalCase, comme `IOperationsBancaires`. Cela indique clairement qu'il s'agit d'une interface.

Gestion des Exceptions



- **Définition**

- Les exceptions en programmation sont des conditions anormales ou des erreurs qui surviennent pendant l'exécution d'un programme. Elles peuvent être causées par des situations imprévues comme une entrée invalide de l'utilisateur, un fichier manquant, ou une erreur de logique dans le code.

- **Importance**

- L'importance de la gestion des exceptions réside dans la capacité d'un programme à réagir de manière gracieuse à des erreurs inattendues, sans s'interrompre brutalement.
- Une bonne gestion des exceptions permet de :
 - Informer l'utilisateur de ce qui ne va pas de manière compréhensible.
 - Protéger le programme contre des états potentiellement dangereux ou instables.
 - Enregistrer des informations cruciales sur l'erreur pour le débogage.
 - Permettre au programme de tenter de récupérer l'erreur, si possible.

- **Conséquences**

- **Crash de l'Application :**

- Si une exception n'est pas correctement gérée, elle peut provoquer l'arrêt inattendu de l'application, entraînant une expérience utilisateur médiocre et la perte potentielle de données.

- **Sécurité Compromise :**

- Certaines erreurs peuvent exposer des informations sensibles aux utilisateurs ou aux attaquants, ou permettre des comportements imprévus qui pourraient être exploités à des fins malveillantes.

- **Difficultés de Débogage :**

- Sans une gestion appropriée et des messages d'erreur clairs, il peut être très difficile de localiser et de corriger les causes sous-jacentes des exceptions.

- **Performance Réduite :**

- Le fait de laisser des exceptions non gérées peut également affecter la performance de l'application, notamment si le programme tente inutilement de continuer après une erreur critique.

- **Manque de Fiabilité :**

- Une application qui ne gère pas correctement les exceptions peut se comporter de manière imprévisible ou produire des résultats incorrects.

- **Définition**

- Les blocs try, catch, et finally permettent de gérer élégamment les erreurs et les exceptions qui peuvent survenir lors de l'exécution du programme.

- **Syntaxe de base**

- **Try Block :**

- Le bloc try contient le code qui peut potentiellement générer une exception. C# exécute le code à l'intérieur de ce bloc jusqu'à ce qu'une exception soit rencontrée.

- **Catch Block :**

- Le bloc catch est utilisé pour gérer l'exception. Il est placé immédiatement après le bloc try et peut spécifier le type d'exception à attraper. Si une exception se produit dans le bloc try, l'exécution passe immédiatement au bloc catch.

- **Finally Block :**

- Le bloc finally est exécuté après que les blocs try et catch ont terminé leur exécution, peu importe si une exception a été levée ou non. Ce bloc est optionnel mais utile pour nettoyer les ressources, fermer les connexions à des fichiers ou à des bases de données, etc.

```
try {  
    // Code susceptible de générer une exception  
}  
catch (ExceptionType e) {  
    // Code pour gérer l'exception  
}  
finally {  
    // Code qui s'exécute toujours, avec ou sans exception  
}
```



```
try {  
    int divisor = 0;  
    int result = 10 / divisor;  
    Console.WriteLine(result);  
}  
catch (DivideByZeroException e) {  
    Console.WriteLine("Erreur: Tentative de division par zéro.");  
}  
finally {  
    Console.WriteLine("Cette ligne s'exécute toujours, peu importe le résultat.");  
}
```

Dans cet exemple, une division par zéro génère une `DivideByZeroException`. Le bloc `catch` capture cette exception et affiche un message d'erreur. Le bloc `finally` s'exécute ensuite pour confirmer que le code à l'intérieur du bloc `finally` est toujours exécuté.

```
try {
    int[] numbers = {1, 2, 3};
    Console.WriteLine(numbers[5]); // Tentative d'accès à un index inexistant
}
catch (IndexOutOfRangeException e) {
    Console.WriteLine("Erreur: Index hors des limites du tableau.");
}
catch (Exception e) {
    Console.WriteLine($"Erreur inattendue: {e.Message}");
}
finally {
    Console.WriteLine("Nettoyage optionnel.");
}
```

Ici, une tentative d'accès à un élément de tableau hors des limites génère une `IndexOutOfRangeException`. Le premier bloc `catch` spécifique gère cette exception. Si une autre exception avait été levée, le deuxième bloc `catch` (plus générique) l'aurait gérée. Le bloc `finally` assure que le nettoyage s'exécute toujours.

- **Introduction**

- **Exception :**

- Classe de base pour toutes les exceptions. Capture tout type d'erreur qui se produit pendant l'exécution du programme.

- **SystemException:Exception :**

- Classe de base pour toutes les exceptions pré-définies par le système. Syntaxe de base

DivideByZeroException

Se produit lors d'une division par zéro dans les opérations arithmétiques.

```
try
{
    int division = 10 / 0;
}
catch (DivideByZeroException ex)
{
    Console.WriteLine("Erreur : Tentative de division par zéro.");
}
```

NullReferenceException

Se produit lors de la tentative d'accès à un membre d'un objet null.

```
string text = null;
try
{
    int length = text.Length;
}
catch (NullReferenceException ex)
{
    Console.WriteLine("Erreur : Tentative d'accès à un objet null.");
}
```

IndexOutOfRangeException

Se produit lors de l'accès à un élément d'un tableau avec un indice qui est en dehors de la plage valide.

```
int[] numbers = { 1, 2, 3 };  
try  
{  
    int number = numbers[3];  
}  
catch (IndexOutOfRangeException ex)  
{  
    Console.WriteLine("Erreur : Indice de tableau hors limites.");  
}
```

InvalidOperationException

Se produit lors d'opérations non valides dans le contexte courant.

```
List<int> list = new List<int>();
try
{
    int first = list.First();
}
catch (InvalidOperationException ex)
{
    Console.WriteLine("Erreur : Opération non valide.");
}
```

- **Objectifs**

- 1. Clarification des erreurs :

- Utiliser des exceptions génériques peut rendre le débogage difficile, car elles ne fournissent pas suffisamment de contexte sur l'erreur. Les exceptions personnalisées permettent de spécifier plus précisément la nature de l'erreur.

- 2. Gestion d'erreurs spécifique :

- Elles permettent de réagir de manière appropriée à des conditions d'erreur spécifiques à votre application.

- 3. Amélioration de la lisibilité du code :

- Elles rendent le code plus lisible et plus facile à maintenir, en indiquant clairement les types d'erreurs que votre application peut générer.

- **Comment créer vos propres exceptions?**

- **Étape 1 : Définir la classe d'exception**

- Hériter de la classe `Exception` ou de toute autre classe d'exception plus spécifique.
- Utiliser le mot-clé : `base()` pour appeler le constructeur de la classe parent.

```
public class MonExceptionPersonnalisee : Exception
{
    public MonExceptionPersonnalisee() : base() {}

    public MonExceptionPersonnalisee(string message) : base(message) {}

    public MonExceptionPersonnalisee(string message, Exception inner) :
base(message, inner) {}
}
```

- **Comment créer vos propres exceptions?**
 - Étape 2 : Ajouter des propriétés spécifiques (facultatif)
 - Vous pouvez ajouter des propriétés spécifiques à votre exception pour fournir des informations supplémentaires sur l'erreur.

```
public class ValidationException : Exception
{
    public int CodeErreur { get; }

    public ValidationException(int codeErreur, string message) :
    base(message)
    {
        CodeErreur = codeErreur;
    }
}
```

Exemple 1

Supposons que vous ayez une application qui traite des inscriptions à des cours. Vous pourriez créer une exception personnalisée pour gérer le cas où un cours est complet.

```
public class CoursCompleException : Exception
{
    public CoursCompleException() : base("Le cours est complet.") {}
}

// Utilisation
public void InscrireEtudiant(string cours)
{
    if (EstComple(cours))
    {
        throw new CoursCompleException();
    }
    // Logique d'inscription
}
```

Exemple 2

Imaginons une application de gestion de stock où vous devez vérifier la quantité disponible d'un produit.

```
public class StockInsuffisantException : Exception
{
    public int QuantiteDemandee { get; }
    public int QuantiteDisponible { get; }

    public StockInsuffisantException(int quantiteDemandee, int quantiteDisponible)
        : base($"Quantité demandée : {quantiteDemandee}, Quantité disponible : {quantiteDisponible}")
    {
        QuantiteDemandee = quantiteDemandee;
        QuantiteDisponible = quantiteDisponible;
    }
}

// Utilisation
public void VerifierStock(int produitId, int quantiteDemandee)
{
    int quantiteDisponible = ObtenirQuantiteDisponible(produitId);
    if (quantiteDemandee > quantiteDisponible)
    {
        throw new StockInsuffisantException(quantiteDemandee, quantiteDisponible);
    }
    // Logique de traitement
}
```

- **Bonnes pratiques**

- Principe de base :

- Les exceptions doivent être gérées à un niveau où une action significative peut être entreprise pour remédier à l'erreur. Cela signifie que la capture d'une exception doit avoir un but précis, comme corriger l'erreur, journaliser l'information de manière plus détaillée, ou informer l'utilisateur d'une manière qui lui permette de comprendre ce qui s'est passé.

- Exemple didactique :

- Imaginons une application qui lit des données depuis un fichier. Si le fichier n'existe pas, une `FileNotFoundException` peut être levée. Au lieu de gérer cette exception immédiatement dans la méthode qui lit le fichier, il pourrait être plus judicieux de la laisser se propager jusqu'à une couche supérieure qui saurait comment informer l'utilisateur ou tenter une action corrective, comme demander un nouveau chemin de fichier.

- Conseil pratique :

- Évitez de gérer les exceptions à un niveau trop bas si cela ne fait que masquer le problème.

- **Exceptions silencieuses : à éviter !!**

- **Définition :**

- Une exception silencieuse se produit lorsque une exception est capturée mais n'est pas traitée ou journalisée, ce qui peut rendre le débogage et le suivi des problèmes extrêmement difficiles.

- **Pourquoi c'est un problème :**

- Ignorer une exception sans prendre de mesures (comme journaliser l'erreur ou avertir l'utilisateur) peut conduire à des comportements imprévus de l'application et masquer des bugs qui pourraient autrement être identifiés et corrigés.

- **Exceptions silencieuses : à éviter !!**

- **Exemple illustratif :**

- Si une application tente de se connecter à une base de données et échoue, ignorer cette exception pourrait conduire l'application à continuer comme si de rien n'était, conduisant potentiellement à des erreurs plus critiques plus tard.

- **Stratégies pour les éviter :**

- Toujours journaliser les exceptions capturées, même si la décision est prise de ne pas interrompre l'exécution du programme.
- Considérer l'utilisation de politiques de reprise après erreur, comme des tentatives de reconnexion ou des messages d'erreur explicites pour l'utilisateur.
- Utiliser des frameworks de logging pour centraliser et faciliter l'analyse des erreurs.

- **Exemple**

- Créez une application simple qui tente de lire des données depuis plusieurs fichiers et traite diverses exceptions (comme `FileNotFoundException` et `IOException`).
- Implémentez une gestion d'exceptions qui illustre les bonnes pratiques discutées, en veillant à informer l'utilisateur de façon claire lorsque des erreurs surviennent, et à journaliser les détails pour un diagnostic ultérieur.

Design Patterns



- **Définition :**

- Un *Design Pattern* (ou motif de conception) est une solution réutilisable à un problème récurrent dans le développement logiciel.
- Il s'agit d'une sorte de "plan" ou de "modèle" qui guide les développeurs dans la conception de systèmes robustes, flexibles et évolutifs.
- Les design patterns ne sont pas des morceaux de code que l'on peut copier-coller directement, mais plutôt des concepts abstraits qui peuvent être appliqués dans divers contextes et langages de programmation.

- **Pourquoi les utiliser ?**

- **Réutilisabilité du code** : Les design patterns fournissent des solutions éprouvées et testées pour des problèmes communs, réduisant ainsi le besoin de réinventer la roue.
- **Lisibilité et maintenabilité** : L'utilisation de patterns bien connus rend le code plus lisible et compréhensible pour d'autres développeurs qui connaissent ces motifs.
- **Flexibilité et extensibilité** : Les patterns sont conçus pour encourager une conception de code qui est plus modulaire, ce qui facilite l'ajout de nouvelles fonctionnalités ou la modification de l'existant.

- **Singleton :**

- Garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à celle-ci. Par exemple, une classe BanqueManager qui gère tous les Compte pourrait être implémentée comme un singleton pour assurer un point d'accès unique à la gestion des comptes.

- **Factory :**

- Fournit une interface pour créer des objets dans une super-classe, permettant aux sous-classes de modifier le type d'objets créés. Pour créer différents types de Compte (courant, épargne), une FactoryCompte pourrait simplifier l'instanciation.

- **Observer :**

- Permet à un objet de notifier automatiquement d'autres objets de changements. Si le Solde d'un Compte change, cela pourrait automatiquement notifier des observateurs, comme une interface utilisateur qui affiche le solde.

Exemple du Pattern Singleton

188

```
public class BanqueManager
{
    private static BanqueManager instance;
    private List<Compte> comptes;

    // Constructeur privé pour empêcher l'instanciation
    // externe
    private BanqueManager()
    {
        comptes = new List<Compte>();
    }

    // Méthode publique statique pour accéder à l'instance
    public static BanqueManager Instance
    {
        get
        {
            if (instance == null)
            {
                instance = new BanqueManager();
            }
            return instance;
        }
    }
}
```

```
/ Méthode pour ajouter un compte
public void AjouterCompte(Compte compte)
{
    comptes.Add(compte);
}

// Méthode pour afficher les soldes de tous les comptes
public void AfficherSoldes()
{
    foreach (Compte compte in comptes)
    {
        compte.AfficherSolde();
    }
}

// D'autres méthodes pour gérer les comptes peuvent
// être ajoutées ici
}
```

```
// Utilisation du BanqueManager
class Program
{
    static void Main(string[] args)
    {
        // Création de quelques comptes
        Compte compte1 = new Compte("Jean Dupont", 1);
        CompteEpargne compteEpargne = new
        CompteEpargne("Marie Curie", 2, 2.5m);

        // Récupération de l'instance unique de
        BanqueManager
        BanqueManager banqueManager =
        BanqueManager.Instance;

        // Ajout des comptes au gestionnaire
        banqueManager.AjouterCompte(compte1);
        banqueManager.AjouterCompte(compteEpargne);

        // Affichage des soldes des comptes via le
        gestionnaire
        banqueManager.AfficherSoldes();
    }
}
```

Dans cet exemple, BanqueManager est un singleton, ce qui signifie qu'il ne peut exister qu'une seule instance de cette classe dans l'application. L'accès à cette instance unique se fait par la propriété statique Instance. Cela garantit que toutes les interactions avec les comptes bancaires passent par un point central, permettant une gestion cohérente et évitant les conflits d'accès ou les incohérences dans les données.

L'utilisation du constructeur privé empêche la création directe d'instances de BanqueManager depuis l'extérieur de la classe, forçant toute interaction à passer par l'instance unique accessible via Instance. Ce modèle est particulièrement utile dans les contextes où une coordination centralisée est nécessaire, comme la gestion des comptes bancaires dans cet exemple.

Design Pattern Singleton

Classe BanqueManager

Instance unique

```
private static BanqueManager instance; : Instance  
statique privée
```

Constructeur privé

```
private BanqueManager() : Empêche l'instanciation  
externe
```

Accès à l'instance

Propriété publique statique `Instance`

Crée l'instance s'il n'en existe pas

Retourne l'instance existante

Gestion des comptes

```
Méthode AjouterCompte(Compte compte) : Ajoute un  
compte à la liste
```

```
Méthode AfficherSoldes() : Affiche les soldes de tous  
les comptes
```

```
public abstract class Compte
{
    public string Titulaire { get; set; }
    public decimal Solde { get; protected set; }
    public int CompteId { get; set; }

    public Compte(string titulaire, int compteId)
    {
        Titulaire = titulaire;
        CompteId = compteId;
        Solde = 0;
    }

    public abstract void Deposer(decimal montant);
    public abstract bool Retirer(decimal montant);
    public abstract void AfficherSolde();
}
```

Pour illustrer le pattern Factory en C#, prenons l'exemple d'une FactoryCompte qui crée des instances de CompteEpargne ou CompteCourant basées sur des paramètres d'entrée. Ce modèle de conception permet de décentraliser la logique de création des objets, rendant le code plus modulaire et facilitant l'ajout de nouveaux types de comptes sans modifier le code existant.


```
public class CompteEpargne : Compte
{
    private decimal tauxInteret;

    public CompteEpargne(string titulaire, int compteId, decimal tauxInteret) : base(titulaire, compteId)
    {
        this.tauxInteret = tauxInteret;
    }

    public override void Deposer(decimal montant)
    {
        Solde += montant;
        Console.WriteLine($"Dépôt de {montant} réussi dans le compte épargne. Nouveau solde: {Solde}.");
    }

    (...)
}
```

```
public class CompteEpargne : Compte
{
    (..)
    public override bool Retirer(decimal montant)
    {
        if (montant <= Solde)
        {
            Solde -= montant;
            Console.WriteLine($"Retrait de {montant} réussi du compte épargne. Nouveau solde: {Solde}.");
            return true;
        }
        else
        {
            Console.WriteLine("Fonds insuffisants pour ce retrait du compte épargne.");
            return false;
        }
    }

    public override void AfficherSolde()
    {
        Console.WriteLine($"Le solde du compte épargne est de {Solde}. Intérêt: {tauxInteret}%");
    }
}
```

```
public class CompteCourant : Compte
{
    public CompteCourant(string titulaire, int compteId) : base(titulaire, compteId)
    {
    }

    public override void Deposer(decimal montant)
    {
        Solde += montant;
        Console.WriteLine($"Dépôt de {montant} réussi dans le compte courant. Nouveau solde: {Solde}.");
    }

    (...)
}
```

```
public class CompteCourant : Compte
{
    (..)
    public override bool Retirer(decimal montant)
    {
        if (montant <= Solde)
        {
            Solde -= montant;
            Console.WriteLine($"Retrait de {montant} réussi du compte courant. Nouveau solde: {Solde}.");
            return true;
        }
        else
        {
            Console.WriteLine("Fonds insuffisants pour ce retrait du compte courant.");
            return false;
        }
    }

    public override void AfficherSolde()
    {
        Console.WriteLine($"Le solde du compte courant est de {Solde}.");
    }
}
```

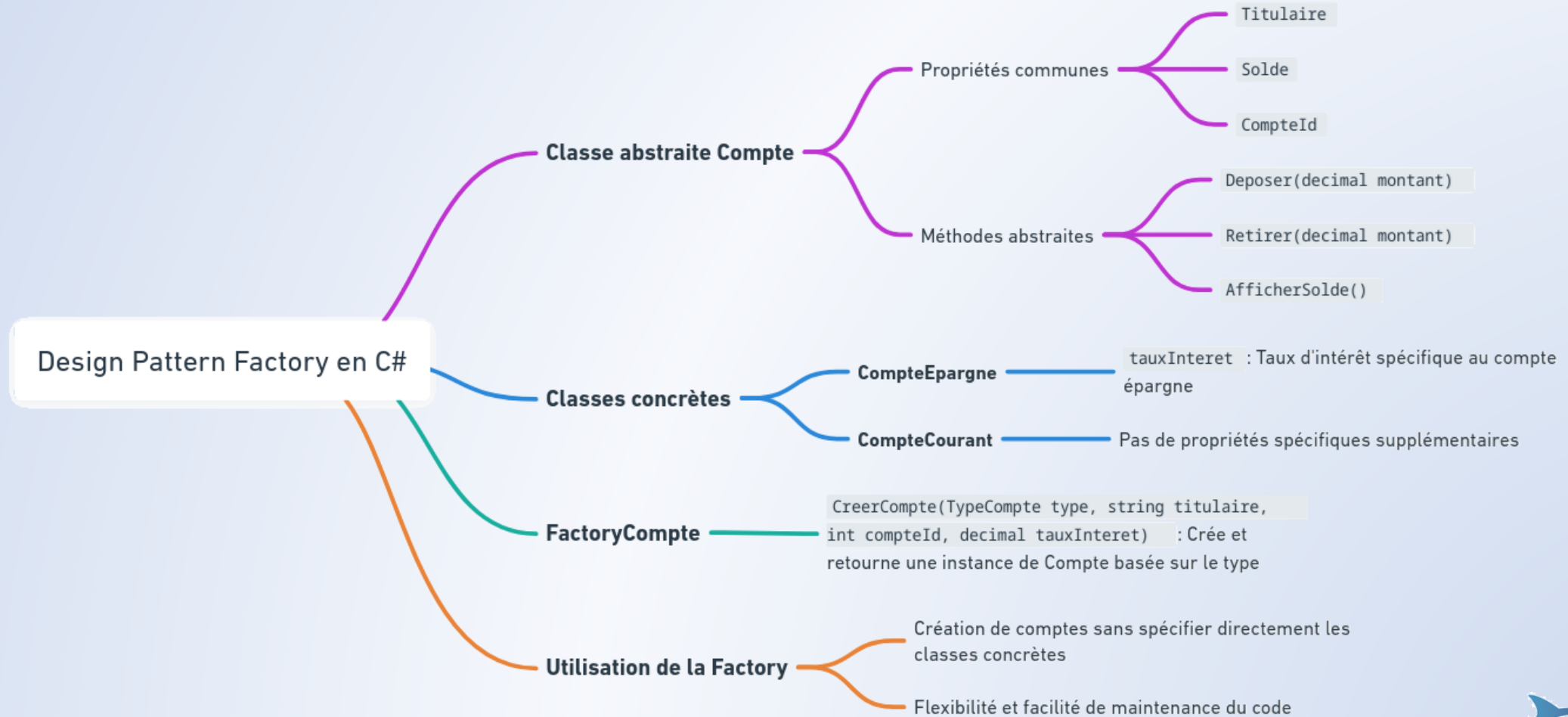
```
public class FactoryCompte
{
    public enum TypeCompte
    {
        Courant,
        Epargne
    }

    public static Compte CreerCompte(TypeCompte type, string titulaire, int compteId, decimal tauxInteret = 0)
    {
        switch (type)
        {
            case TypeCompte.Courant:
                return new CompteCourant(titulaire, compteId);
            case TypeCompte.Epargne:
                return new CompteEpargne(titulaire, compteId, tauxInteret);
            default:
                throw new ArgumentException("Type de compte non pris en charge.");
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Compte compteCourant = FactoryCompte.CreerCompte(FactoryCompte.TypeCompte.Courant, "Alice Dupont", 1);
        Compte compteEpargne = FactoryCompte.CreerCompte(FactoryCompte.TypeCompte.Epargne, "Bob Martin", 2, 0.05m);

        compteCourant.Deposer(1000);
        compteEpargne.Deposer(5000);

        compteCourant.AfficherSolde();
        compteEpargne.AfficherSolde();
    }
}
```



```
public interface IObservateur
{
    void Actualiser(decimal nouveauSolde);
}

public interface ISujet
{
    void AjouterObservateur(IObservateur observateur);
    void SupprimerObservateur(IObservateur observateur);
    void NotifierObservateurs();
}
```

Pour illustrer le pattern Observer en C#, utilisons l'exemple d'un système bancaire où un Compte peut notifier ses observateurs, par exemple des clients ou des services internes, à chaque fois que son solde change. Ce pattern est utile pour créer un système de notification ou de mise à jour automatique entre objets, où les observateurs s'abonnent à un sujet pour recevoir des mises à jour sans que le sujet ait besoin de connaître les détails de ses observateurs.

Interfaces Observer et Subject

Commençons par définir les interfaces de base pour les observateurs (IObservateur) et le sujet (ISujet): ←

Exemple du Pattern Observer

200

```
public class Compte : ISujet
{
    private decimal solde;
    private List<IObservateur> observateurs = new List<IObservateur>();

    public Compte(decimal soldeInitial)
    {
        solde = soldeInitial;
    }

    public void Deposer(decimal montant)
    {
        solde += montant;
        Console.WriteLine($"Dépôt de {montant}. Nouveau solde: {solde}.");
        NotifierObservateurs();
    }

    public void AjouterObservateur(IObservateur observateur)
    {
        observateurs.Add(observateur);
    }
}
```

```
(...)  
public void Retirer(decimal montant)  
{  
    if (montant <= solde)  
    {  
        solde -= montant;  
        Console.WriteLine($"Retrait de {montant}. Nouveau solde: {solde}.");  
        NotifierObservateurs();  
    }  
    else  
    {  
        Console.WriteLine("Solde insuffisant.");  
    }  
}  
public void SupprimerObservateur(IObservateur observateur)  
{  
    observateurs.Remove(observateur);  
}  
public void NotifierObservateurs()  
{  
    foreach (var observateur in observateurs)  
    {  
        observateur.Actualiser(solde);  
    }  
}
```

```
public class Client : IObservateur
{
    private string nom;

    public Client(string nom)
    {
        this.nom = nom;
    }

    public void Actualiser(decimal nouveauSolde)
    {
        Console.WriteLine($"{nom} a été notifié. Nouveau
solde du compte: {nouveauSolde}");
    }
}
```

Implémentation du Compte comme Sujet

Ensuite, implémentons la classe Compte qui implémente ISujet: ↑

Implémentation d'un Observateur Concret

Créons un observateur concret, par exemple Client qui implémente IObservateur: ←

```
class Program
{
    static void Main(string[] args)
    {
        Compte compte = new Compte(1000);
        Client client1 = new Client("Alice");
        Client client2 = new Client("Bob");

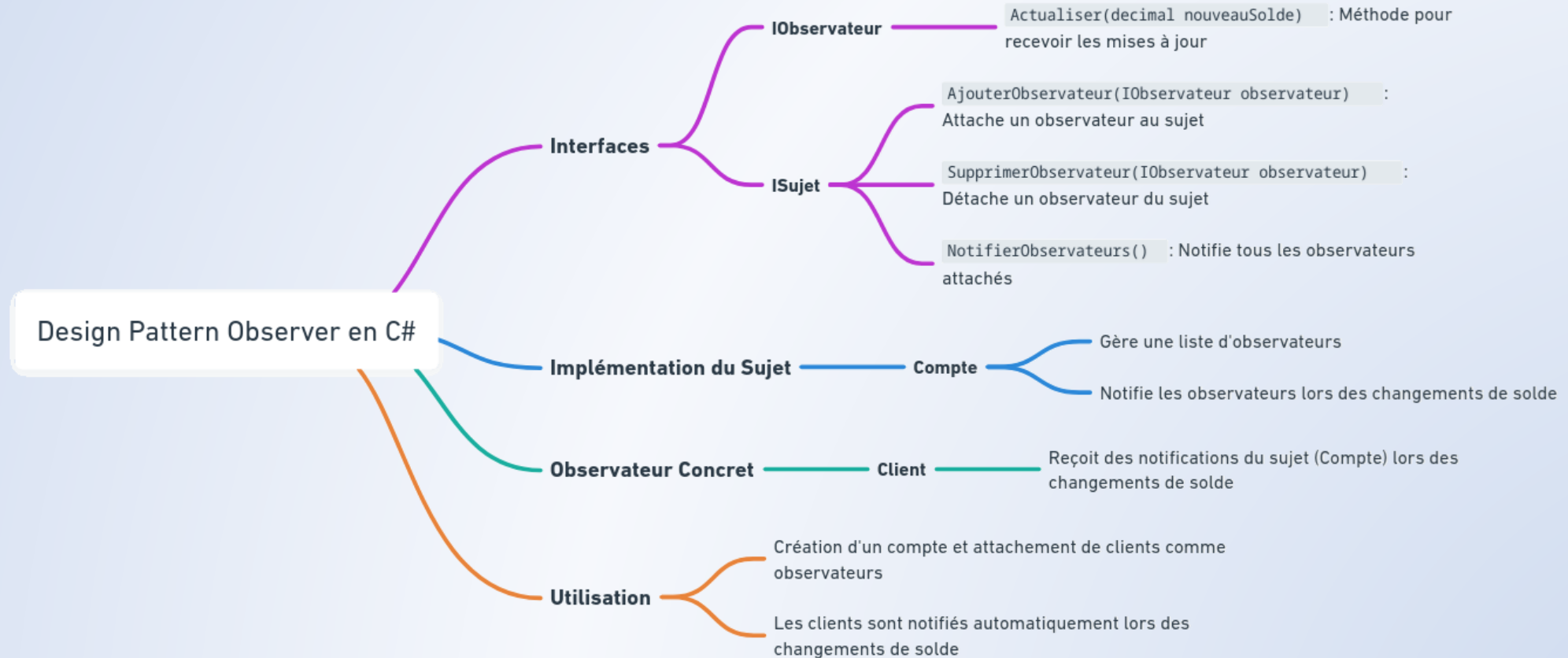
        compte.AjouterObservateur(client1);
        compte.AjouterObservateur(client2);

        compte.Deposer(500); // Cela notifiera Alice et Bob
        compte.Retirer(200); // Cela notifiera également
        Alice et Bob
    }
}
```

Utilisation du Pattern Observer

Finalement, utilisons ces classes pour créer un compte et y attacher des clients comme observateurs. Lorsque le solde du compte change, les clients sont notifiés automatiquement: ←

Dans cet exemple, lorsque Deposer ou Retirer est appelé sur compte, il notifie automatiquement tous ses observateurs (dans ce cas, client1 et client2) du changement de solde. Ce mécanisme permet une communication flexible entre le sujet et ses observateurs sans qu'ils soient fortement couplés, ce qui facilite la maintenance et l'évolution du système.



Principe de substitution de Liskov



- **Définition :**

- Ce principe stipule qu'un objet d'une classe dérivée doit être substituable par un objet de sa classe de base sans affecter le bon fonctionnement du programme. Par exemple, partout où un Compte est utilisé, un CompteEpargne devrait également pouvoir être utilisé sans problème.

```
public class Oiseau
{
    public virtual void Voler()
    {
        Console.WriteLine("L'oiseau vole.");
    }
}

public class Pigeon : Oiseau
{
    public override void Voler()
    {
        Console.WriteLine("Le pigeon vole.");
    }
}
```

```
// Dans le Main ou une méthode similaire
Oiseau oiseau = new Oiseau();
Pigeon pigeon = new Pigeon();

FaireVolerOiseau(oiseau); // Affiche "L'oiseau vole."
FaireVolerOiseau(pigeon); // Affiche "Le pigeon vole."
```

Ici, le principe est respecté car Pigeon, en tant que sous-classe de Oiseau, peut être utilisé partout où un Oiseau est attendu, sans que cela ne modifie le comportement attendu du programme.


```
public class Oiseau
{
    public virtual void Voler()
    {
        Console.WriteLine("L'oiseau vole.");
    }
}

public class Autruche : Oiseau
{
    public override void Voler()
    {
        throw new NotImplementedException("Les autruches ne
volent pas.");
    }
}
```

```
public void FaireVolerOiseau(Oiseau oiseau)
{
    oiseau.Voler();
}

// Dans le Main ou une méthode similaire
Oiseau autruche = new Autruche();

FaireVolerOiseau(autruche); // Lève une
NotImplementedException
```

Dans ce cas, bien que Autruche soit une sous-classe de Oiseau, son utilisation en tant que Oiseau entraîne une exception, car les autruches ne volent pas. L'implémentation de la méthode Voler() dans Autruche viole le LSP car elle modifie le comportement attendu de la classe de base de manière incompatible, ce qui peut entraîner des erreurs lors de l'exécution.

Multiple héritage via Interfaces



- **Définition :**

- C# ne supporte pas directement le multiple héritage pour les classes, mais il permet à une classe d'implémenter plusieurs interfaces. Par exemple, si Compte implémente une interface IOperationsBancaires et une autre interface IAuditable, cela permet à Compte d'avoir des comportements définis dans les deux interfaces.

Méthodes d'extension



- **Définition :**

- Les méthodes d'extension permettent d'ajouter de nouvelles méthodes à une classe existante sans modifier son code source.
- Par exemple, vous pourriez ajouter une méthode `AfficherDetailsCompte` à la classe `Compte` sans la modifier directement.

```
public static class ExtensionsCompte
{
    public static void AfficherDetailsCompte(this Compte compte)
    {
        Console.WriteLine($"Compte {compte.CompteId}: Titulaire = {compte.Titulaire},
Solde = {compte.Solde}");
    }
}
```

Dépendances / Injection de Dépendances



- **Définition :**

- L'injection de dépendances est une technique pour réaliser l'inversion de contrôle entre classes et leurs dépendances.
- Par exemple, si Compte dépend d'un service de notification, au lieu de l'instancier directement dans Compte, le service est fourni (injecté) via le constructeur ou une propriété.


```
public interface IServiceNotification
{
    void EnvoyerNotification(string message);
}

public class ServiceNotification : IServiceNotification
{
    public void EnvoyerNotification(string message)
    {
        Console.WriteLine($"Notification envoyée : {message}");
    }
}
```

Interfaces et classes de service

Tout d'abord, définissons une interface pour notre service de notification, ce qui facilite le découplage et améliore la testabilité.

```
public class Compte
{
    private readonly IServiceNotification
    _serviceNotification;

    public Compte(IServiceNotification serviceNotification)
    {
        _serviceNotification = serviceNotification;
    }

    public void EffectuerOperation(decimal montant)
    {
        // Logique d'opération sur le compte
        Console.WriteLine($"Opération effectuée sur le
compte pour le montant de {montant}.");

        // Utilisation du service de notification
        _serviceNotification.EnvoyerNotification($"Une
opération de {montant} a été effectuée sur votre compte.");
    }
}
```

Classe Compte

Ensuite, la classe Compte utilise IServiceNotification. Au lieu de créer une instance spécifique de ServiceNotification, elle reçoit une référence à IServiceNotification via son constructeur. Cela permet d'injecter n'importe quelle implémentation de IServiceNotification, rendant la classe Compte plus flexible et plus facile à tester.

```
// Création de l'instance de service de notification
IServiceNotification serviceNotification = new
ServiceNotification();

// Injection de la dépendance via le constructeur de Compte
Compte compte = new Compte(serviceNotification);

// Utilisation du compte
compte.EffectuerOperation(100.0m);
```

Utilisation

Enfin, voici comment vous pourriez instancier et utiliser la classe Compte avec un ServiceNotification injecté

Gestion des dépendances



- **Introduction à NuGet**

- **Qu'est-ce que NuGet ?**

- NuGet est le gestionnaire de packages officiel pour .NET. Il permet aux développeurs de partager et consommer du code facilement. NuGet héberge des milliers de packages tiers ou open-source utilisables dans vos projets.

- **Pourquoi utiliser NuGet ?**

- Simplification de la gestion des dépendances.
 - Automatisation des mises à jour des bibliothèques.
 - Facilitation de la collaboration et du partage de code.
 - Réduction du risque d'incompatibilités et de conflits entre bibliothèques.

- **Installation de NuGet**

- Présentation de l'interface de gestion de NuGet dans Visual Studio.
- Comment rechercher et installer un package NuGet dans votre projet.
- Ajout d'un dépôt personnel

- **Mise à Jour des Packages**

- Importance de garder les packages à jour pour la sécurité et l'accès aux nouvelles fonctionnalités.
- Comment mettre à jour les packages via l'interface NuGet dans Visual Studio.

- **Bonnes Pratiques**

- Toujours tester votre application après la mise à jour d'un package pour s'assurer que la mise à jour n'a pas introduit de régressions.
- Lire les notes de version des packages pour comprendre les changements, améliorations, ou corrections de bugs.
- Utiliser les versions préliminaires (pré-releases) avec prudence dans les projets de production.

- Avant d'ajouter un package ...

- Compatibilité avec la version du framework:

- Assurez-vous que le package NuGet est compatible avec la version du framework .NET que vous utilisez dans votre projet. Certains packages peuvent être spécifiques à une version particulière de .NET, ce qui pourrait entraîner des problèmes de compatibilité si vous utilisez une version différente.

- Popularité et réputation du package:

- Il est utile de vérifier la popularité et la réputation du package auprès de la communauté des développeurs. Vous pouvez consulter les commentaires, les évaluations et les discussions sur la page du package sur NuGet.org, ainsi que d'autres sources en ligne telles que GitHub ou des forums de développeurs.

- Maintenance et activité du package:

- Assurez-vous que le package est activement maintenu par ses auteurs. Les packages qui ne sont plus mis à jour peuvent poser des problèmes de compatibilité avec les nouvelles versions du framework ou des dépendances. Vous pouvez vérifier la date de la dernière mise à jour du package sur NuGet.org ou sur le référentiel GitHub du projet si disponible.

- Avant d'ajouter un package ...

- Dépendances et conflits potentiels:

- Examinez les dépendances du package pour vous assurer qu'elles ne créent pas de conflits avec d'autres packages utilisés dans votre projet. La console NuGet peut vous aider à visualiser les dépendances du package avant de l'installer.

- Licence du package:

- Vérifiez la licence du package pour vous assurer qu'elle est compatible avec les politiques de licence de votre projet. Certains packages peuvent avoir des restrictions d'utilisation qui pourraient affecter la distribution ou la vente de votre application.

- Taille du package et performances:

- Prenez en compte la taille du package, surtout si vous travaillez sur des applications où la taille est un facteur critique, comme les applications mobiles. Des packages plus volumineux peuvent affecter les performances de votre application et le temps de déploiement.

Manipulation de fichier CSV



- **Définition**

- CSV Helper est une bibliothèque puissante et flexible pour travailler avec des fichiers CSV en C#.
- Elle simplifie grandement les opérations de lecture et d'écriture des données CSV, tout en offrant des fonctionnalités avancées pour la gestion des formats complexes, des mappings personnalisés, et plus encore.

- **1/ Définition d'une classe de modèle**

- Supposons que vous avez un fichier CSV avec des données sur des utilisateurs, avec des colonnes pour le Id, le Nom, et l'Email. Vous pouvez définir une classe de modèle correspondante comme suit :

```
public class Utilisateur
{
    public int Id { get; set; }
    public string Nom { get; set; }
    public string Email { get; set; }
}
```

• 2/ Lecture du fichier CSV

- Une fois que votre classe de modèle est définie, vous pouvez utiliser CSV Helper pour lire le fichier CSV et mapper les données à une liste d'objets Utilisateur.

```
using (var reader = new StreamReader("utilisateurs.csv"))
    using (var csv = new CsvReader(reader, CultureInfo.InvariantCulture))
    {
        var utilisateurs = csv.GetRecords<Utilisateur>();
        foreach (var utilisateur in utilisateurs)
        {
            Console.WriteLine($"Id: {utilisateur.Id}, Nom: {utilisateur.Nom}, Email: {utilisateur.Email}");
        }
    }
```

- **1/ Création d'une liste d'objets**

- Supposons que vous avez une liste d'objets Utilisateur que vous souhaitez écrire dans un fichier CSV.

```
var utilisateurs = new List<Utilisateur>
{
    new Utilisateur { Id = 1, Nom = "Alice", Email = "alice@example.com" },
    new Utilisateur { Id = 2, Nom = "Bob", Email = "bob@example.com" },
    new Utilisateur { Id = 3, Nom = "Charlie", Email = "charlie@example.com" }
};
```

- **2/ Écriture du fichier CSV**

- Pour écrire cette liste dans un fichier CSV, vous pouvez utiliser le code suivant :

```
using (var writer = new StreamWriter("utilisateurs.csv"))  
    using (var csv = new CsvWriter(writer, CultureInfo.InvariantCulture))  
    {  
        csv.WriteRecords(utilisateurs);  
    }
```

Manipulation de fichier Excel (COM)



- **Définition**

- COM Interop est une technologie qui permet aux applications .NET (et pas que) d'interagir avec les objets COM (Component Object Model), tels que Microsoft Excel. Cela permet de contrôler Excel directement depuis C#, notamment pour lire et écrire des données dans des fichiers Excel.

- **Ajout de la référence**

- Ouvrez votre projet dans Visual Studio.
- Cliquez avec le bouton droit sur "**Références**" dans l'Explorateur de solutions, puis sélectionnez "**Ajouter une référence...**".
- Dans la fenêtre de référence, allez à "**Assemblies > Extensions**".
- Recherchez "**Microsoft Excel X Object Library**" (où "X" correspond à la version d'Excel installée) et cochez la case.
- Cliquez sur "**OK**".

- Écriture

```
using Excel = Microsoft.Office.Interop.Excel;
using System;

public class ExcelInterop
{
    public void EcrireAvecMatrice()
    {
        // Créer une instance de l'application Excel
        var excelApp = new Excel.Application();
        excelApp.Visible = true;

        // Créer un nouveau classeur
        Excel.Workbook workbook = excelApp.Workbooks.Add(Type.Missing);
        Excel.Worksheet worksheet = workbook.Sheets[1];
        worksheet.Name = "Données";

        // Créer une matrice de données
        string[,] data = new string[,]{
            { "ID", "Nom", "Email" },
            { "1", "Alice", "alice@example.com" },
            { "2", "Bob", "bob@example.com" },
            { "3", "Charlie", "charlie@example.com" }
        };
    }
}
```

- Écriture

```
// Déterminer la plage de cellules à remplir
Excel.Range startCell = worksheet.Cells[1, 1];
Excel.Range endCell = worksheet.Cells[data.GetLength(0), data.GetLength(1)];
Excel.Range writeRange = worksheet.Range[startCell, endCell];

// Affecter la matrice à la plage de cellules en une seule opération
writeRange.Value2 = data;

// Enregistrer le fichier Excel
string chemin = @"C:\Temp\Exemple.xlsx";
workbook.SaveAs(chemin);

// Fermer et libérer les ressources
workbook.Close();
excelApp.Quit();
}
```

- Lecture

```
public void LireExcel()
{
    // Créer une instance de l'application Excel
    var excelApp = new Excel.Application();

    // Ouvrir le classeur existant
    string chemin = @"C:\Chemin\Vers\Votre\Fichier.xlsx";
    Excel.Workbook workbook = excelApp.Workbooks.Open(chemin);
    Excel.Worksheet worksheet = workbook.Sheets[1];

    // Lire les données des cellules
    string id1 = worksheet.Cells[2, 1].Value.ToString();
    string nom1 = worksheet.Cells[2, 2].Value.ToString();
    string email1 = worksheet.Cells[2, 3].Value.ToString();
}
```

Manipulation de fichier JSON



- **Définition**

- Le format JSON (JavaScript Object Notation) est largement utilisé pour l'échange de données entre les applications web et les services web.
- En C#, la bibliothèque Newtonsoft.Json est l'une des bibliothèques les plus populaires et les plus puissantes pour travailler avec des données JSON.
- Elle permet de sérialiser des objets en JSON, de désérialiser des chaînes JSON en objets, ainsi que de lire et écrire des fichiers JSON facilement.

- **Sérialisation d'un objet en JSON**

- La sérialisation est le processus de conversion d'un objet C# en une chaîne JSON, tandis que la désérialisation consiste à convertir une chaîne JSON en un objet C#.

```
public static void Main(string[] args)
{
    var personne = new Personne
    {
        Id = 1,
        Nom = "Alice",
        Email = "alice@example.com"
    };

    // Sérialisation en JSON
    string json = JsonConvert.SerializeObject(personne, Formatting.Indented);
    Console.WriteLine(json);
}
```


- **Désérialisation d'un objet en JSON**

- La désérialisation consiste à convertir une chaîne JSON en un objet C#.

```
public static void Main(string[] args)
{
    string json = @"{
        'Id': 1,
        'Nom': 'Alice',
        'Email': 'alice@example.com'
    }";

    // Désérialisation du JSON en objet
    Personne personne = JsonConvert.DeserializeObject<Personne>(json);
    Console.WriteLine($"ID: {personne.Id}, Nom: {personne.Nom}, Email: {personne.Email}");
}
```

Tests unitaires



• Définition

- Les tests unitaires sont une pratique essentielle en développement logiciel, permettant de vérifier le bon fonctionnement des unités de code (comme les méthodes ou les classes) de manière isolée.
- En C#, les tests unitaires jouent un rôle crucial dans la validation du code, garantissant que chaque composant fonctionne comme prévu.
- Cela aide à détecter les bugs tôt dans le processus de développement, facilite la maintenance et assure une meilleure qualité du code
- Un test unitaire est un petit morceau de code qui vérifie qu'une unité spécifique de votre programme (comme une méthode ou une fonction) fonctionne correctement. L'objectif est de tester le comportement du code dans différents scénarios, y compris les cas normaux, les cas limites et les cas d'erreur.
 - Par exemple, si vous avez une méthode qui calcule la somme de deux nombres, un test unitaire vérifierait que cette méthode renvoie la somme correcte pour différents couples de nombres.

```
public class Calculatrice
{
    public int Additionner(int a, int b)
    {
        return a + b;
    }
}

using Microsoft.VisualStudio.TestTools.UnitTesting;

[TestClass]
public class CalculatriceTests
{
    [TestMethod]
    public void Additionner_AvecDeuxNombres_ReturnsSomme()
    {
        // Arrange
        var calculatrice = new Calculatrice();
        int a = 5;
        int b = 10;
        int resultatAttendu = 15;

        // Act
        int resultat = calculatrice.Additionner(a, b);

        // Assert
        Assert.AreEqual(resultatAttendu, resultat);
    }
}
```

1. A partir du site Yahoo Finance, récupérer la série de prix de Apple.
2. Automatisez ce processus en fonction du code fourni par l'utilisateur.
3. Optimisez cette fonction en utilisant `async/await` pour lancer plusieurs requêtes simultanément.

4. Mettre en place la stratégie de trading suivante :

- Calcul de la moyenne mobile de taille n . Permettre le calcul de cette moyenne avec deux méthodes :
 - Classique : $\bar{x} = \frac{\sum_i X_i}{n}$
 - Exponentielle : soit N le nombre de période qui représente 86% du poids total, $\alpha = 1 - (1 - 0.86)^{1/N}$ la constante de lissage associée, $\bar{x}_i = \sum_i \alpha(1 - \alpha)^n x_{t-i} = \alpha(x_i + (1 - \alpha)x_{i-1} + (1 - \alpha)^2 x_{i-2} + \dots) = \bar{x}_{i-1}(1 - \alpha) + x_i \alpha$
- Soit deux moyennes mobiles : 1 dite de court terme (par exemple $n=50$) et 1 dite de long terme (par exemple $n=200$). Si la court terme croise à la hausse la long terme, alors c'est un signal d'achat. Si la court terme croise à la baisse la long terme, alors c'est un signal de vente.

5. Backtester cette stratégie : vous calculerez les indicateurs suivants : la volatilité, la performance totale, annualisée et la performance mois à mois.
6. Vous comparerez cette stratégie par rapport à une stratégie de Buy & Hold.
7. Exportez les résultats dans Excel (ce fichier sera un fichier « *template* » où l'on pourra avoir des graphiques etc.)