**La Programmation Orientée Objet (POO)**

La programmation orientée objet (POO) est un paradigme de programmation basé sur la modélisation du monde réel sous forme d'objets. Elle facilite la modularité, la réutilisation du code et la maintenance des logiciels.

## **Concepts Fondamentaux de la POO**

### ****1.1 Classes et Objets****

* **Classe** : Modèle ou plan qui définit les attributs (données) et les méthodes (fonctions) d’un type d’objet.
* **Objet** : Instance d’une classe, ayant ses propres valeurs d’attributs.

**Example :**

**using System;**

**class Voiture**

**{**

**public string Marque { get; set; }**

**public string Couleur { get; set; }**

**public Voiture(string marque, string couleur)**

**{**

**Marque = marque;**

**Couleur = couleur;**

**}**

**public void AfficherDetails()**

**{**

**Console.WriteLine($"Marque: {Marque}, Couleur: {Couleur}");**

**}**

**}**

**class Program**

**{**

**static void Main()**

**{**

**Voiture voiture1 = new Voiture("Toyota", "Rouge");**

**Voiture voiture2 = new Voiture("BMW", "Noir");**

**voiture1.AfficherDetails();**

**voiture2.AfficherDetails();**

**}**

**}**

### ****1.2 Encapsulation****

L'encapsulation protège les données en limitant l'accès direct aux attributs.

En C#, les **modificateurs get et set** sont utilisés pour **encapsuler** les propriétés d’une classe. Ils permettent de **contrôler** l'accès et la modification des champs (**variables**) d'une classe tout en respectant le **principe d'encapsulation**.

#### Exemple :

**class Personne**

**{**

**private string nom; // Champ privé**

**public string Nom // Propriété avec get et set**

**{**

**get { return nom; } // Récupère la valeur**

**set { nom = value; } // Modifie la valeur**

**}**

**}**

**class Program**

**{**

**static void Main()**

**{**

**Personne p = new Personne();**

**p.Nom = "Alice"; // Appelle le set**

**Console.WriteLine(p.Nom); // Appelle le get -> Affiche "Alice"**

**}**

**}**

## **Propriétés Automatiques (**auto-properties**)**

C# permet d’écrire des propriétés plus simplement **sans champ privé explicite**.

### ****Exemple : Propriété Automatique****

**class Personne**

**{**

**public string Nom { get; set; } // C# crée automatiquement le champ privé**

**}**

**class Program**

**{**

**static void Main()**

**{**

**Personne p = new Personne { Nom = "Bob" };**

**Console.WriteLine(p.Nom); // ✅ Bob**

**}**

**}**

### ****1.3 Héritage****

L'héritage permet de créer une classe à partir d'une autre, facilitant la réutilisation du code.

L’héritage en **C#** utilise le symbole : pour indiquer qu'une classe hérite d'une autre.

#### Exemple :

**using System;**

**class Animal**

**{**

**public string Nom { get; set; }**

**public Animal(string nom)**

**{**

**Nom = nom;**

**}**

**public virtual void Parler()**

**{**

**Console.WriteLine("Cet animal fait un son.");**

**}**

**}**

**class Chien : Animal**

**{**

**public Chien(string nom) : base(nom) { }**

**public override void Parler()**

**{**

**Console.WriteLine($"{Nom} dit Woof!");**

**}**

**}**

**class Chat : Animal**

**{**

**public Chat(string nom) : base(nom) { }**

**public override void Parler()**

**{**

**Console.WriteLine($"{Nom} dit Miaou!");**

**}**

**}**

**class Program**

**{**

**static void Main()**

**{**

**Chien chien = new Chien("Rex");**

**Chat chat = new Chat("Minou");**

**chien.Parler(); // Rex dit Woof!**

**chat.Parler(); // Minou dit Miaou!**

**}**

**}**

### ****1.4 Polymorphisme****

Le polymorphisme permet d’utiliser la même interface pour différents types d’objets.

En **C#**, on peut utiliser le polymorphisme en déclarant des **méthodes virtuelles (virtual) et des méthodes redéfinies (override)**.

#### Exemple :

### ****1.5 Abstraction****

L’abstraction permet de définir des classes de base avec des méthodes obligatoires pour les classes dérivées.

L’abstraction en **C#** est implémentée avec des **classes abstraites (abstract) et des méthodes**

#### Exemple avec ABC (Abstract Base Class) :

using System;

abstract class Forme

{

public abstract double Aire(); // Méthode abstraite

}

class Cercle : Forme

{

private double rayon;

public Cercle(double rayon)

{

this.rayon = rayon;

}

public override double Aire()

{

return 3.14 \* rayon \* rayon;

}

}

class Program

{

static void Main()

{

Cercle cercle = new Cercle(5);

Console.WriteLine($"Aire du cercle: {cercle.Aire()}"); // 78.5

}

}

### ****1.6 Interface****

Une **interface** en **C#** est une définition de contrat qui spécifie un ensemble de méthodes et de propriétés qu'une classe doit implémenter. Contrairement aux classes abstraites, une interface **ne contient pas d'implémentation** : elle ne définit que la structure que les classes implémentantes doivent respecter.

Example :

using System;

// Définition de l'interface

interface IAnimal

{

void Parler(); // Méthode sans implémentation

}

// Classe implémentant l'interface

class Chien : IAnimal

{

public void Parler()

{

Console.WriteLine("Woof!");

}

}

// Classe implémentant l'interface

class Chat : IAnimal

{

public void Parler()

{

Console.WriteLine("Miaou!");

}

}

class Program

{

static void Main()

{

IAnimal chien = new Chien();

IAnimal chat = new Chat();

chien.Parler(); // Woof!

chat.Parler(); // Miaou!

}

}

Ici, **IAnimal** est une interface qui impose à toute classe qui l'implémente de définir la méthode **Parler()**.

## **2. Différences entre Interface et Classe Abstraite**

| **Aspect** | **Interface (interface)** | **Classe Abstraite (abstract class)** |
| --- | --- | --- |
| Héritage | Une classe peut implémenter **plusieurs** interfaces | Une classe ne peut hériter que d'**une seule** classe abstraite |
| Implémentation | Aucune implémentation, uniquement des déclarations | Peut contenir des méthodes implémentées et abstraites |
| Modificateurs | Ne contient pas de public, private, etc. (tout est public par défaut) | Peut avoir des attributs/méthodes private, protected, public |
| Constructeurs | **Non**, pas de constructeur | **Oui**, peut avoir des constructeurs |

**Exemple : Interface vs Classe Abstraite**

// Interface : uniquement des déclarations

interface IDeplacable

{

void Deplacer();

}

// Classe abstraite : peut contenir des méthodes avec implémentation

abstract class Animal

{

public abstract void Parler(); // Méthode abstraite (à implémenter)

public void Respirer() // Méthode concrète (déjà implémentée)

{

Console.WriteLine("Je respire...");

}

}

// Chien hérite d'Animal et implémente IDeplacable

class Chien : Animal, IDeplacable

{

public override void Parler()

{

Console.WriteLine("Woof!");

}

public void Deplacer()

{

Console.WriteLine("Je cours !");

}

}

class Program

{

static void Main()

{

Chien chien = new Chien();

chien.Parler(); // Woof!

chien.Respirer(); // Je respire...

chien.Deplacer(); // Je cours !

}

}

🔹 Ici, Chien hérite de Animal et **implémente** l’interface IDeplacable.  
🔹 La **classe abstraite Animal** a une méthode abstraite Parler() (obligatoire) et une méthode concrète Respirer() (optionnelle).  
🔹 L’**interface IDeplacable** oblige Chien à implémenter Deplacer().

## **3. Héritage Multiple avec Interfaces**

Une classe **peut implémenter plusieurs interfaces**, contrairement à l’héritage de classe unique.

### ****Exemple : Une classe implémente plusieurs interfaces****

interface IVolant

{

void Voler();

}

interface INageant

{

void Nager();

}

// Implémentation multiple

class Canard : IVolant, INageant

{

public void Voler()

{

Console.WriteLine("Je vole !");

}

public void Nager()

{

Console.WriteLine("Je nage !");

}

}

class Program

{

static void Main()

{

Canard canard = new Canard();

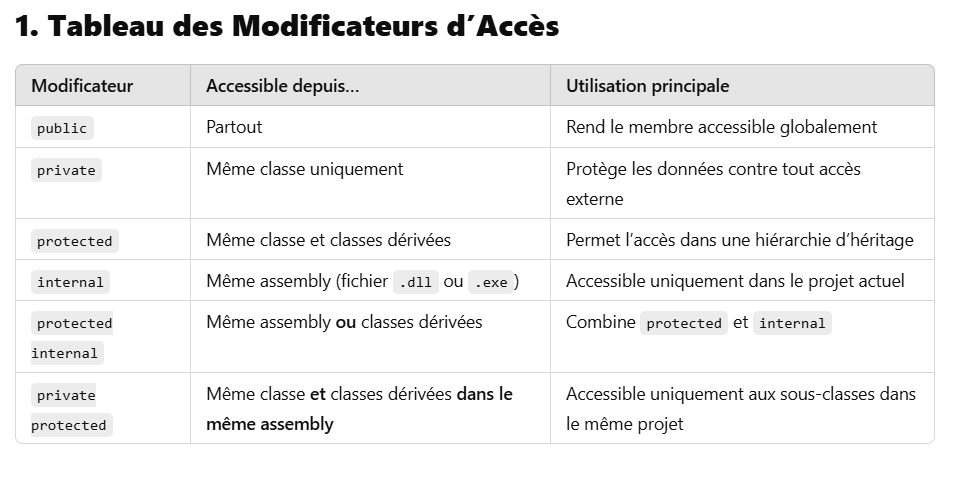
canard.Voler(); // Je vole !

canard.Nager(); // Je nage !

}

}

🔹 Canard **implémente** IVolant et INageant, ce qui lui permet de **voler** et **nager**.  
🔹 C’est un grand avantage par rapport aux classes abstraites, qui ne permettent **pas** l’héritage multiple.



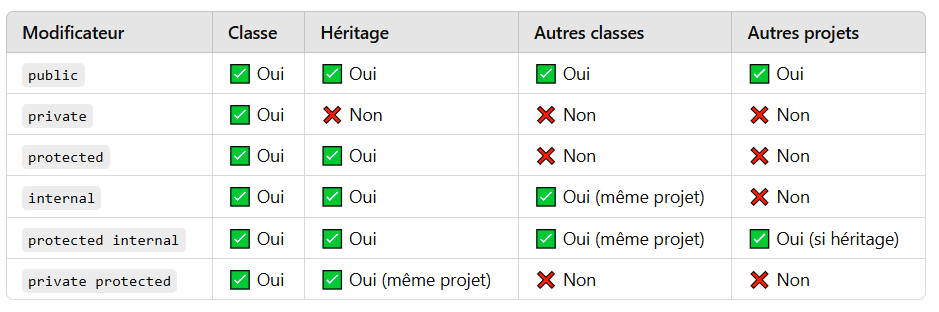
private : Accessible uniquement dans la classe

protected : Accessible dans la classe et les classes dérivées

internal : Accessible dans le même assembly (même projet)

protected internal : Accessible dans le même assembly **ou** par héritage

private protected : Accessible uniquement dans les classes dérivées du même assembly



## **4. Avantages de la POO**

✅ **Modularité** : Code mieux organisé.  
✅ **Réutilisation** : Héritage permet de réutiliser des classes existantes.  
✅ **Maintenance facilitée** : Modification d’une classe sans impacter tout le programme.  
✅ **Encapsulation** : Sécurise les données.  
✅ **Polymorphisme** : Flexibilité et extensibilité du code.