Ce qu'il faut retenir sur la programmation orientée objet.

La programmation orientée objet (ou POO) est un paradigme de programmation. En d'autres termes, c'est une manière de "conceptualiser" un programme informatique où toutes les briques constituantes seraient des "objets". Dans la mesure où il s'agit d'une manière de concevoir son programme la POO n'est pas propre au C#.

Quatre grands principes constituent la programmation orientée objet :

1. **L'encapsulation** qui consiste à encapsuler le code à l'intérieur d'objet.
2. **L'abstraction** consiste à masquer la complexité à l'intérieur de ces objets afin de passer par l'utilisation de ceux-ci plutôt que directement via du code sans structure.
3. **L'héritage** qui permet de créer de nouvelles classes à partir d'autres classes dans le but de créer une cohérence entre ses objets.
4. **Le polymorphisme** ou une autre manière de dire que les classes enfants peuvent être assimilés aux classes parentes.

👉 "Je n'ai pas bien compris la différence entre une classe et un objet ?"

Une **classe**, c'est le moule, le template ou le descriptif de ce que votre objet doit être bref, c'est la recette de cuisine.

Un **objet**, c'est la matérialisation d'une classe. On parle aussi d'instance d'une classe. Pour continuer sur la métaphore culinaire, l'objet c'est le plat que vous allez cuisiner à partir de la recette de cuisine.

quand vous faites `var myObj = new MyObject();`

Vous instanciez **l'objet** `myObj` qui va donc être l'instance de la **classe** `MyObject`.

👉 "J'ai compris l'idée générale mais concrètement je ne vois pas comment ça s'appliquerait dans un vrai projet professionnel ?"

Supposons que vous souhaitez coder un site internet et plus particulièrement un Blog. Un blog sera composé d'articles. On peut facilement imaginer que chaque article sera composée d'un titre puis du contenu du texte. On pourrait tout à fait matérialiser cela sous forme d'objet à l'aide d'une classe.

```
Class Article {
public string Title {get;set;} // Mon titre
public string Content {get;set;} // Le contenu de mon article.
}
```

Je pourrais même si j'en ai envie créer une méthode qui me permettrait d'avoir une version réduite de l'article.

```
Class Article {
public string Title {get;set;} // Mon titre
public string Content {get;set;} // Le contenu de mon article.

  public string GetSummary(){
 // retourne une version sommaire du contenu de mon article.
 }
}
```

Mais ce n'est pas tout, on pourrait aussi envisager de créer une classe qui nous permettrait de récupérer des informations provenant d'un fichier de donnée par exemple au format (XML ou JSON), si vous ne savez pas ce que c'est comprenez simplement qu'il s'agit de fichier de données dans un format particulier.

Et on pourrait créer une classe qu'on appellerait XMLParser avec une série de propriétés et de méthodes pour traiter ces fichiers et les manipuler selon notre besoin.

Ce qu'il faut retenir de la POO

L'héritage :

```
class Shape {

public void Draw(){
 Console.WriteLine("Dessine la forme");
 }

}

class Square : Shape {

}
```

En faisant le code suivant la classe Square (Carré en anglais) hérite de la classe Shape (Forme). Cela est possible grâce à l'utilisation des ":" après Square.

Par conséquent, la classe Square aura les propriétés et méthodes de la classe Shape.

Je pourrai faire :

```
var square = new Square();
var shape = new Shape();
square.Draw();
shape.Draw();
```

Les deux lignes afficherons "Dessine la forme".

Si je considère qu'il n'y a pas d'intérêt d'avoir un objet shape parce que c'est une représentation abstraite je peux préfixer la classe par le mot abstract. **Je peux également rendre des méthodes abstraites ce qui impliquera dans la classe hérité je dois l'implémenter (l'écrire) à l'aide du mot clef override.**

```
abstract class Shape {

public void Draw(){
 Console.WriteLine("Dessine la forme");
 }

}
```

Je n'ai pas de limites dans l'héritage je peux avoir 10 niveaux d'héritages si ça me chante. Mais attention à ne pas avoir un code qui se complexifie inutilement et qui deviendrait difficile à maintenir.

Les mot clef `virtual` et `override`

Si j'ai une méthode qui a la même définition dans ma classe fille et dans ma classe mère comme l'illustre le code suivant :

```
class Shape {

public void Draw(){
 Console.WriteLine("Dessine la forme");
 }
}

class Square : Shape {

public void Draw(){
 Console.WriteLine("Dessine la forme d'un carré");
}

}
```

Je vais avoir un code techniquement juste mais qui m'avertira **qu'une méthode héritée en masque une autre.** Pour pallier à ce problème, je peux utiliser les mots clefs `override` et `virtual` .

1. Le mot clef `virtual` sert à définir qu'une méthode peut être remplacer par une autre de même définition dans une classe fille.
2. La mot clef `override` sert à expliciter qu'une méthode vient écraser une méthode virtuel (ou abstract) présente dans la classe mère.

Ainsi, si je fais :

```
class Shape
{
  public virtual void Draw(){
   Console.WriteLine("Dessine la forme");
  }
}

class Square : Shape
```

```
{
   public override void Draw(){
    Console.WriteLine("Dessine la forme d'un carré");
  }
}
```

Alors mon code suivant :

```
var square = new Square();
var shape = new Shape();
square.Draw();
shape.Draw();
```

Affichera :

pour `square.Draw()` ⇒ "Dessine la forme d'un carré".

pour `shape.Draw()` ⇒ "Dessine la forme".

Il faut bien comprendre que l'intérêt de ces mots clefs, c'est de proposer un contexte ou une relation entre objet que les futurs collègues de boulot pourront **enrichir selon leurs besoins**.

Je peux également accéder à toutes mes méthodes de ma classe parente depuis ma classe fille avec le mot clef `base`. Je peux faire par exemple base.LaMethodeQuiMinteresse(). En réalité, le mot clef base est optionnel, **il permet juste d'enlever les ambiguïtés** entre les méthodes fille et mère si il y en a.

Le polymorphisme

C'est un mot compliqué pour simplement dire **qu'une classe fille peut être assimilée à sa classe mère**. Un exemple sera plus parlant.

Si je reprends mon exemple avec mes classes `square` et `shape`. Alors si je crée une liste de `shape` je pourrai ajouter des `square` et des `shape` à ma liste comme l'illustre le code suivant :

var listOfShapes = new List<Shape>();listOfShapes.Add(new Shape());listOfShapes.Add(new Square());

En revanche, si je créée une liste de `square`, je ne pourrai **seulement** qu'ajouter des classes square parce que `square` hérite de `shape` et pas l'inverse.

Et ceci est valable pour les listes, les tableaux ainsi que les paramètres de fonctions !