L'héritage

L'héritage dans le contexte de la POO permet de créer des **classes mères**, et des **classes filles**. Les classes filles vont **hériter** des **propriétés** et **méthodes** accessibles de la classe mère. L'héritage permet de diviser les fonctionnalités **en plusieurs classes**, et créer une classe mère qui sert de **pattern de base** pour les classes filles. On dit que les classes filles **dérivent** de la classe mère.

```
class User:
 def __init__(self, nom: str, prenom: str):
  self.nom = nom
  self.prenom = prenom

 def get_full_name(self):
  print(f"Utilisateur {self.nom.upper()} {self.prenom.capitalize()}")


# La classe Admin dérive de la classe mère
class Admin(User):
 def __init__(self, nom: str, prenom: str):
  super().__init__(nom, prenom) # Appel du constructeur de la classe mère
  self.admin = True # Propriété spécifique à la classe Admin


admin = Admin("Clooney", "Georges")
admin.get_full_name() # méthode de la classe mère accessible depuis la classe
fille
```

`super()` représente la classe mère. On appelle le constructeur de la classe mère dans le constructeur de la classe fille pour profiter de ses propriétés.

La visibilité

Il existe 3 types de visibilité en programmation orientée objet.

La visibilité `public` d'une propriété ou méthode permet qu'elle soit accessible depuis l'intérieur et l'extérieur de la classe. C'est le réglage par défaut pour toute propriétés ou méthode créées en python.

La visibilité `protected` rend une propriété ou méthode accessible uniquement depuis l'intérieur de la classe, et depuis toutes les classes héritant de la classe où cette propriété ou méthode a été définie. Pour définir une propriété ou méthode en `protected`, la convention prescrit de la préfixer d'un `_`.

Enfin, la visibilité `private` rend la propriété ou méthode accessible uniquement à l'intérieur de la classe dans laquelle elle a été construite. Pour déclarer une propriété ou méthode `private`, vous pouvez les préfixer par `__`.

> [!warning] Particularité de python En python, seul le `__` permet de définir une propriété ou méthode en `private`. Le `_` n'est qu'une convention pour définir une propriété ou méthode en `protected`. Sinon, toutes les propriétés et méthodes sont en `public`.

```
from dataclasses import dataclass

@dataclass
class Visibility:
 name: str      # propriété en public    (par défaut)
 _age: int      # propriété en protected (convention => accessibilité public)
 __solde: float # propriété en private   (effectif)
```

L'encapsulation

> Définition

L'encapsulation en programmation orientée objet constitue le fait de rendre les propriétés d'une classe `private` et de ne permettre l'accès et le changement de ses valeurs par des méthodes. Ces méthodes sont appelés `getters` (accès à la propriété), et `setters` (changement de la valeur de la donné).

L'encapsulation permet de contrôler le cycle de vie des propriétés d'une classse, et de créeer des méthodes ou non, en fonction de si on veut autoriser l'accès ou la mutabilité.

```
class User:

  def __init__(self, nom: str, prenom: str):
    self.__nom = nom
    self.__prenom = prenom

  # Getter & setter => propriété en lecture/écriture
  @property
  def nom(self):
    return self.__nom

  @nom.setter
  def nom(self, valeur):
    self.__nom = valeur

  # Getter sans setter => propriété en lecture seule
  @property
  def prenom(self):
    return self.__prenom


user = User("Pitt", "Brad")
print(user.nom)        # Utilisation d'un getter, la bonne pratique
user.nom = "Tipp"      # Utilisation du setter pour changer le nom
```

```
print(user._User__nom) # Fonctionnera mais mauvaise pratique
print(user.__nom)      # Soulèvera une erreur car la propriété nom est privée
```

> Le décorateur `@property`

Il va permettre de déclarer une méthode dans notre classe qui pourra être utilisée comme une propriété. C'est la bonne pratique en python pour implémenter des `getters`.

> Le décorateur `@nom.setter`

Va nous permettre de modifier la valeur d'une propriété `nom` d'un objet en appelant la méthode comme une propriété.

> **Contrainte à la création d'un objet**
>
> Lors de la création d'un objet, il est parfois nécessaire de vérifier que les valeurs passées au constructeur sont valides. Pour cela, on met les **validations** dans les **getters** et **setters** et on les appelle lors de l'initialisation de l'objet.

```
class User:

  def __init__(self, nom: str):
    self.nom : str = nom  # Appel du setter

  @property
  def nom(self):
    return self.__nom

  @nom.setter
  def nom(self, valeur):
    if not isinstance(valeur, str) or len(valeur) < 2:
      raise ValueError("Le nom doit être une chaîne de caractères d'au moins 2
caractères.")
    self.__nom = valeur
```

La surcharge

Surcharger une méthode définie dans une **classe mère** est l'action de la **redéfinir dans la classe fille** pour **modifier** son comportement.

```
class User:
  def __init__(self, nom: str, prenom: str):
    self.nom = nom
    self.prenom = prenom

  def get_full_name(self):
```

```
    print(f"{self.__class__.__name__}: nom={self.nom.upper()}, prenom=
{self.prenom.capitalize()}")


class Admin(User):
  def __init__(self, nom: str, prenom: str):
    super().__init__(nom, prenom)
    self.admin = True

  # Surcharge de la méthode de la classe mère
  def get_full_name(self):
    print(f"{super().get_full_name()}, admin={self.admin}")


admin = Admin("Clooney", "Georges")
admin.get_full_name() # "Administrateur CLOONEY Georges"
```

Grâce à l'héritage, nous pouvons **hériter** de toutes les fonctionnalités qu'on veut dans la classe fille, mais si certaines méthodes ne nous conviennent plus pour le contexte de la classe fille, nous pouvons toujours **surcharger les méthodes de la classe mère** tout en gardant l'avantage de l'héritage.

L'Abstraction

L'**abstraction** permet de créer des classes abstraites en cachant les détails d'implémentation. Cette approche permet de gérer la complexité des systèmes logiciels en se concentrant sur ce qu'un objet fait plutôt que sur comment il le fait. L'abstraction est essentielle pour créer une architecture logicielle claire, modulaire et facile à maintenir.

Python ne propose par de classe abstraite *stricto sensu*, mais il propose le module `abc` (Abstract Base Classes) qui permet une telle implémentation.

Une classe abstraite est une classe qui ne peut pas être instanciée directement, mais qui sert de modèle pour d'autres classes concrètes. Elle définit des méthodes abstraites que les classes filles doivent implémenter.

> Il est obligatoire d'hériter de la classe `ABC` du module `abc` et d'avoir au moins une méthode abstraite avec le décorateur `@abstractmethod` pour créer une classe abstraite.

```
from abc import ABC, abstractmethod

class Utilisateur(ABC):
  def __init__(self, nom, prenom):
    self.nom = nom
    self.prenom = prenom

  @abstractmethod
  def afficher_nom_complet(self):
```

```
    pass

class Admin(Utilisateur):
  def afficher_nom_complet(self):
    return f"Administrateur {self.nom.upper()} {self.prenom.capitalize()}"


# Exemples d'utilisation
admin = Admin("Clooney", "Georges")
print(admin.afficher_nom_complet())  # "Administrateur CLOONEY Georges"
```

L'utilisation de l'abstraction et de la surcharge se ressemblent beaucoup.

L'idée fondamentale de l'abstraction est d'empêcher l'instanciation d'une classe et d'exiger l'implémentation de méthodes abstraites, créant ainsi un contrat que les classes filles doivent respecter. Cela garantit que chaque classe fille fournira une implémentation spécifique de ces méthodes, mais cela ne modifie pas le comportement des méthodes héritées de la classe mère.

En revanche, la surcharge consiste à redéfinir une méthode existante dans la classe fille pour modifier son comportement. Cela permet à la classe fille de personnaliser ou d'étendre le comportement hérité de la classe mère sans nécessairement ajouter de nouvelles méthodes abstraites.

> [!INFO] les interfaces Concept que l'on ne retrouve pas en python.
>
> Le meilleur mot pour le représenter est un contrat car tous les attributs et les méthodes des interfaces sont abstraites<.>