Mission 9 : Matière à apprendre

La programmation orientée objets :
masquage d'information, variables et méthodes de classe, héritage

Objects
- 1 - Classes and objects - Basics
- 2 - Classes and objects - Advanced
- 3 - Even more object-oriented programming
- 4 - Collections of objects
- 5 - Inheritance
- 6 - Linked lists
- Appendix - Source code of phone class
- Appendix - Source code of card game
- Appendix - Source code of linked lists
- Appendix - Worked out example: accounts



masquage d'information

attributs privés

méthodes accesseurs

variables de classe

méthodes de classe

héritage ***self*** ***super()***

Exemple: un compte en banque

```
class Compte :

  def __init__(self, titulaire):
    self.titulaire = titulaire
    self.solde = 0
```

Compte
titulaire solde

comment protéger ?

```
>>> a = Compte('kim')
>>> a.titulaire
'kim'
>>> a.solde
0
>>> a.solde += 1000
>>> b.solde -= 1000
```



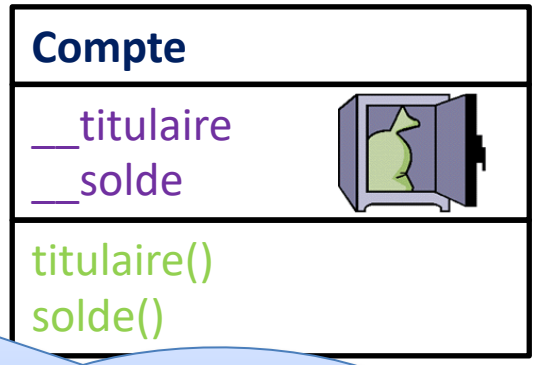
Masquage d'information

```
class Compte :

  def __init__(self, titulaire) :
    self.__titulaire = titulaire
    self.__solde = 0

  def titulaire(self):
    return self.__titulaire

  def solde(self):
    return self.__solde
```




```
>>> a = Compte('kim')
>>> a.__titulaire
AttributeError: 'Compte' object has no attribute '__titulaire'
>>> a.__solde
AttributeError: 'Compte' object has no attribute '__solde'
>>> a.titulaire()
'kim'
>>> a.solde()
0
```

Méthode __str__

```
class Compte :
    def __init__(self, titulaire) :
        self.__titulaire = titulaire
        self.__solde = 0
    def titulaire(self):
        return self.__titulaire
    def solde(self):
        return self.__solde
    def __str__(self) :
        return "Compte de " + self.__titulaire \
               + " : solde = " + str(self.__solde)
>>> print(a) # Compte de Kim : solde = 0
```

MIEUX : Utiliser les accesseurs

méthode __str__

```
def __str__(self) :
    return "Compte de " + self.titulaire() \
           + " : solde = " + str(self.solde())
```

Méthodes mutateurs

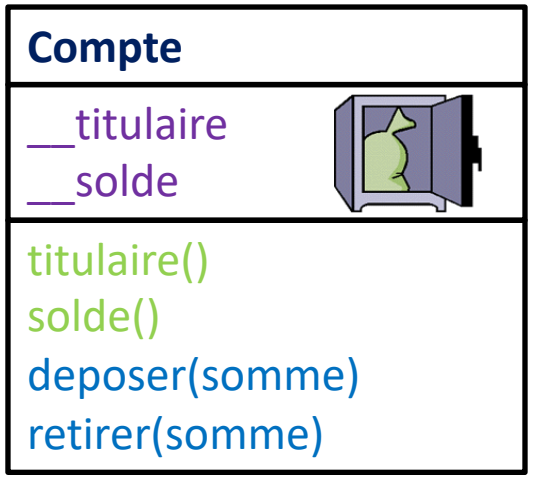
```
class Compte :

  …

  def deposer(self, somme):
    self.__solde += somme
    return self.solde()

  def retirer(self, somme):
    if self.solde() >= somme :
      self.__solde -= somme
      return self.solde()
    else :
      return "Solde insuffisant"
```

```
>>> a = Compte('kim')
>>> a.deposer(100)
100
>>> a.retirer(90)
10
>>> a.retirer(50)
Solde insuffisant
```



Variable de classe

```
class Compte :

  taux_interet = 0.02
  …
```

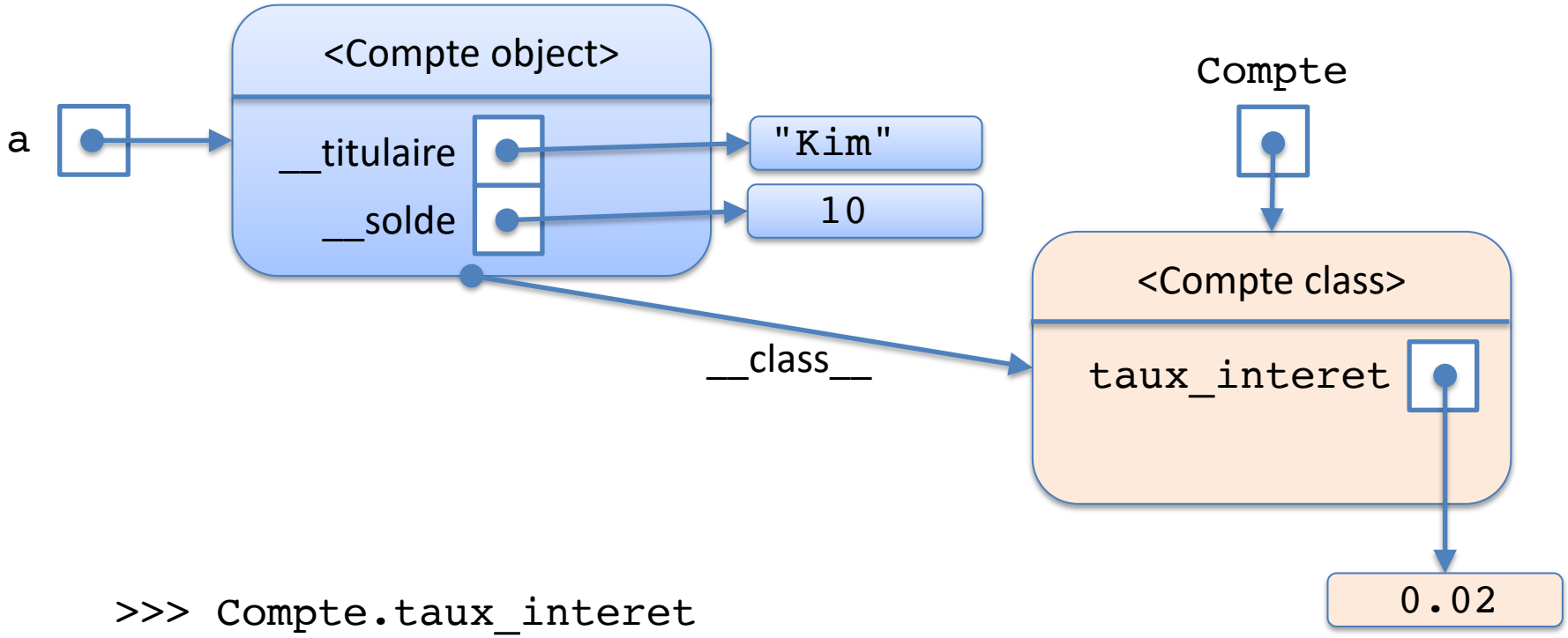
une variable pour la classe

```
>>> Compte.taux_interet
0.02
>>> a = Compte("Kim")
>>> a.taux_interet
0.02
>>> b = Compte("Siegfried")
>>> b.taux_interet
0.02
>>> Compte.taux_interet = 0.04
>>> a.taux_interet
0.04
>>> b.taux_interet
0.04
```

valeur partagée par toutes les instances de la classe

si la valeur change, elle change pour toutes les instances de la classe

Variable de classe

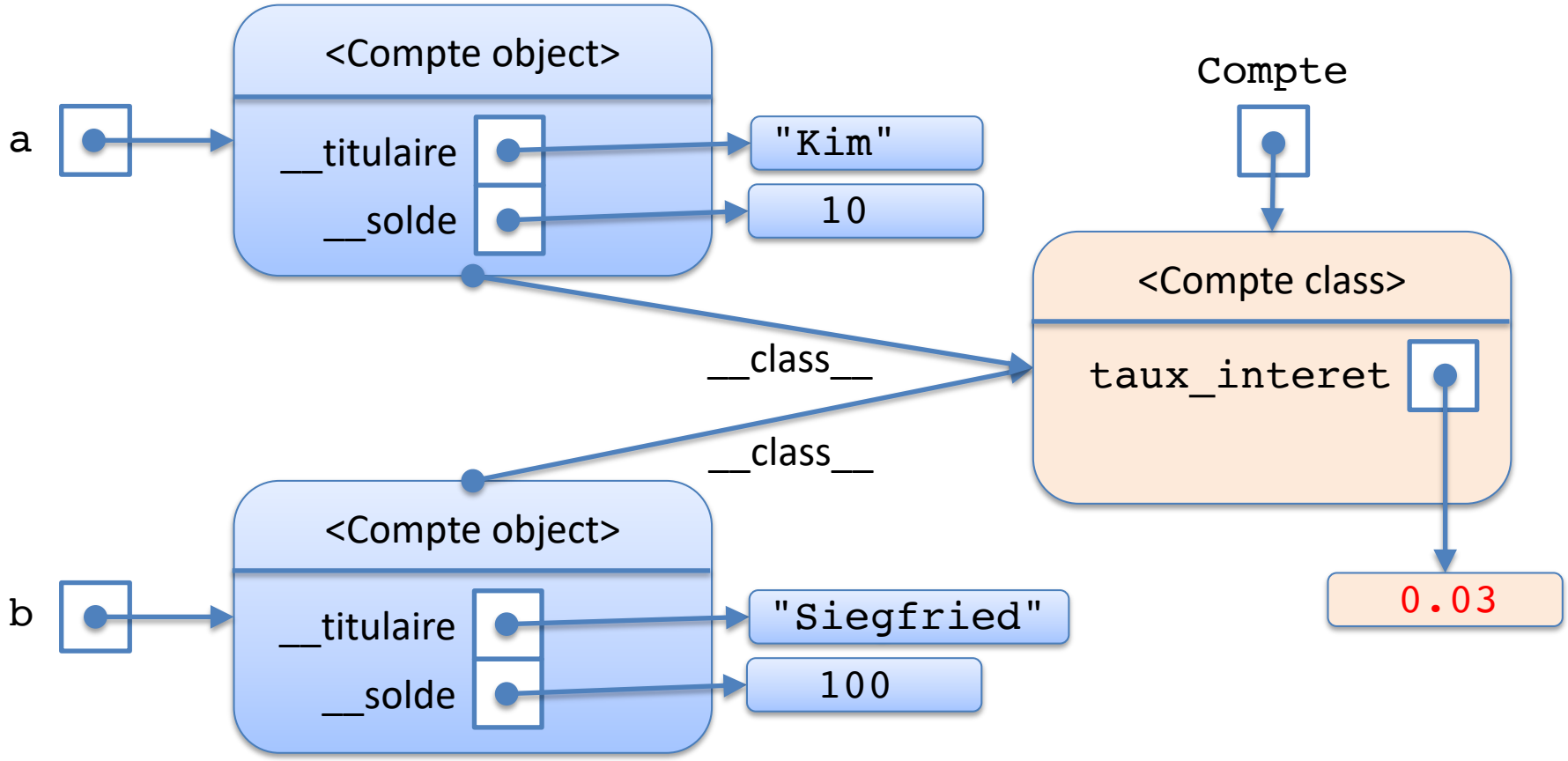



```
>>> Compte.taux_interet
0.02

>>> a.taux_interet
0.02
```

Si pas trouvé dans l'objet, Python va chercher dans la classe…

Variable de classe




```
>>> Compte.taux_interet = 0.03
>>> a.taux_interet
0.03
```

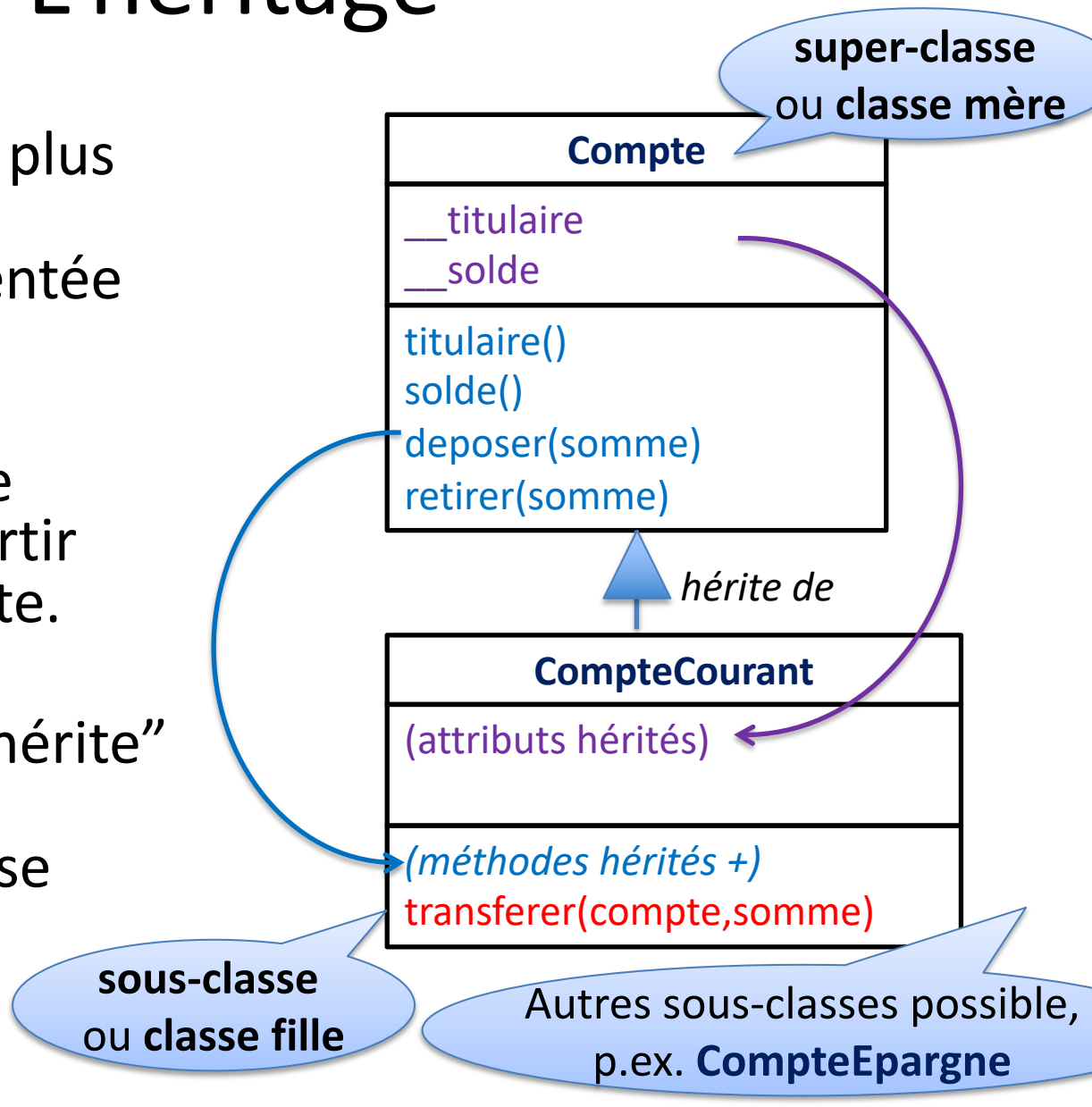
```
>>> b.taux_interet
0.03
```

L'héritage

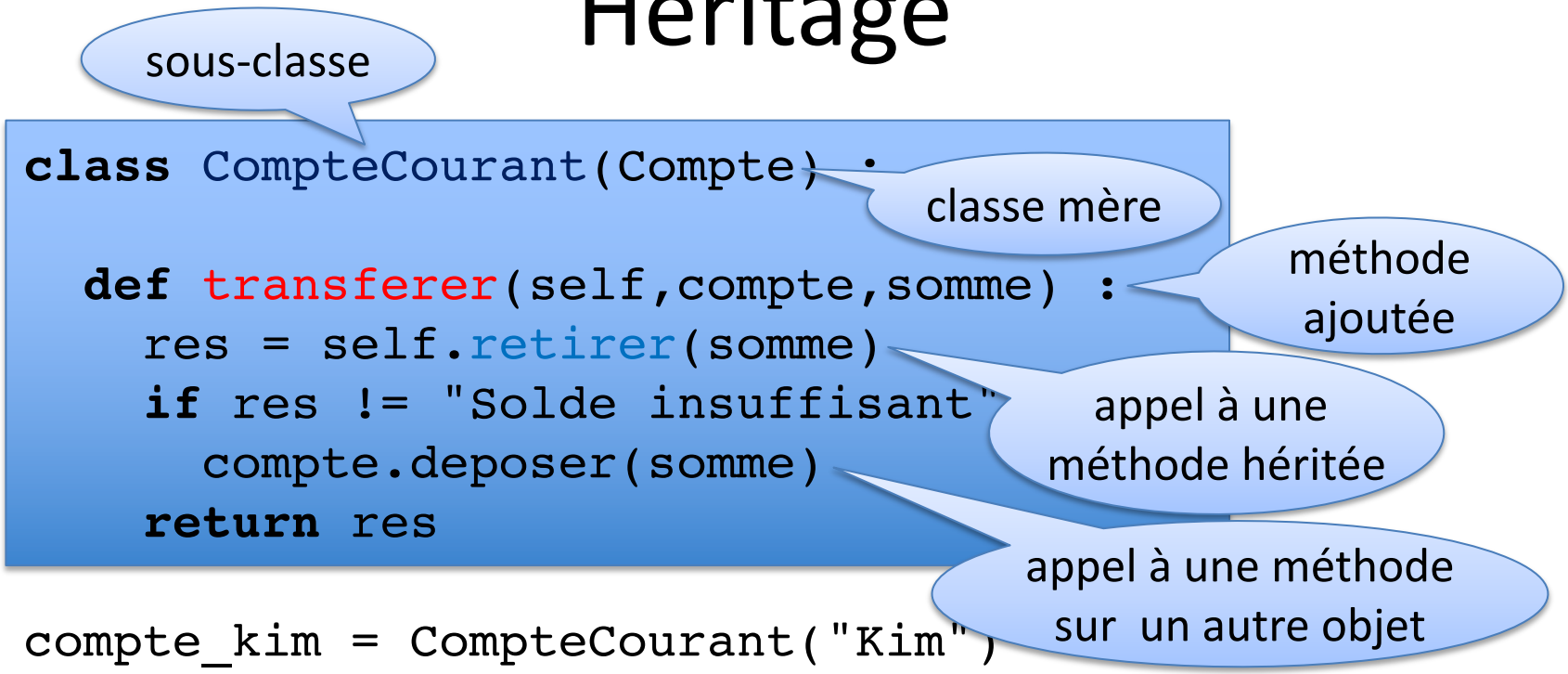
Un des concepts les plus importants de la programmation orientée objet.

Permet de créer une nouvelle classe à partir d'une classe existante.

La nouvelle classe "hérite" tous les attributs et méthodes de la classe existante.



Héritage




```
compte_kim = CompteCourant("Kim")
compte_charles = CompteCourant("Charles")
compte_kim.deposer(100)
compte_kim.transferer(compte_charles,50)

compte_kim.solde()      ➔ 50
compte_charles.solde()  ➔ 50
compte_kim.transferer(compte_charles,60)
                        ➔ Solde insuffisant
```

Redéfinition

*Avec l'heritage on peut aussi **redéfinir** les attributs ou méthodes d'une super-classe*

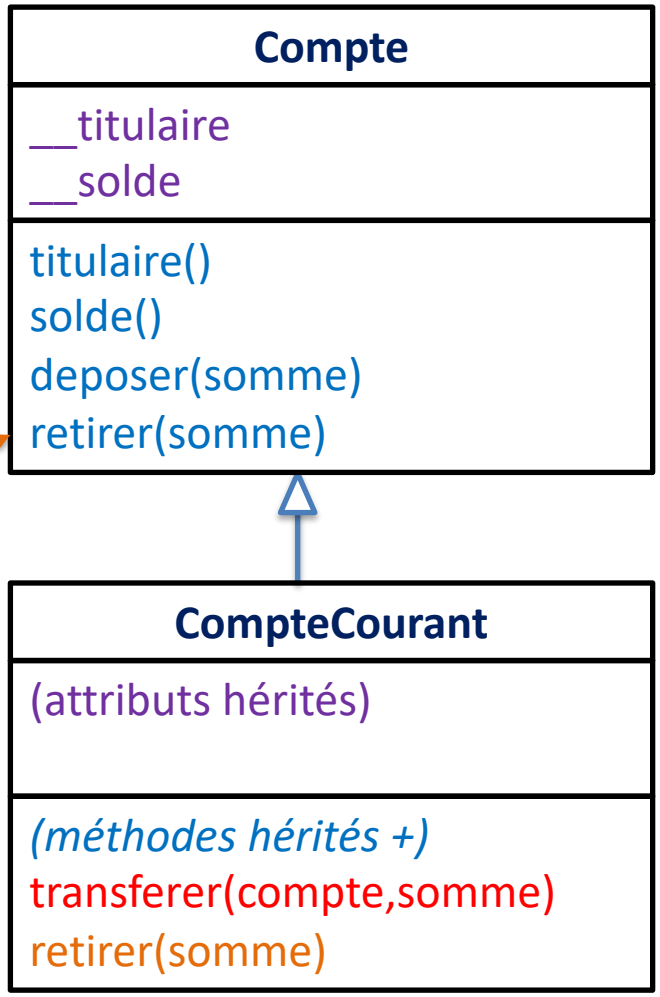
```
class CompteCourant(Compte) :
  __frais_retirer = 0.10

  def transferer(self,…) :
    …

  def retirer(self, somme):
    frais = self.__frais_retirer
    return super().
           retirer(somme+frais)
```

L'héritage c'est super!

```
compte_kim = CompteCourant("Kim")
compte_kim.deposer(1000)   → 1000
compte_kim.retirer(10)     → 989.9
compte_kim.retirer(10)     → 979.8
```



super()

```
class CompteCourant(Compte) :
  __frais_retirer = 0.10
  def __init__(self, titulaire, banque) :
    super().__init__(titulaire)
    self.__banque = banque
  def __str__(self) :
    return super().__str__() +
           "; banque = " + self.__banque
  …
```

`super()` permet de référer à une (méthode dans la) classe mère sans devoir la nommer explicitement.

Souvent utilisé pour étendre une méthode de la super-classe, par exemple:

`retirer(somme)`, `__init__`, `__str__`