Partie III

La programmation orientée objets



Informatique 1
Introduction à la programmation

Mission 9 : INTRODUCTION

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

Kim Mens

Mission 9 : Matière à lire

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

héritage

self

super()



Exemple: un compte en banque

```
class Compte :

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

Compte
titulaire solde

```
>>> a = Compte('kim')
>>> a.titulaire
'kim'
>>> a.solde
0
```



Exemple: un compte en banque

```
class Compte :

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

à protéger

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

à éviter



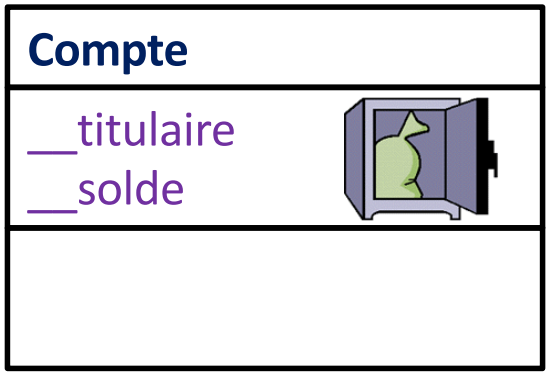
Variables d'instance privés

```
class Compte :

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

attributs ou variables d'instances privés

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



Méthodes accesseurs

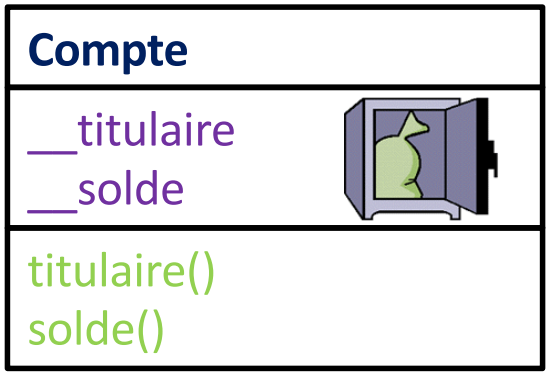
```
class Compte :

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

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

méthodes accesseurs

```
>>> a = Compte('kim')
>>> 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 {} : solde = {}".
           format(self.__titulaire,self.__solde)
```

méthode `__str__`

```
>>> a = Compte('kim')
>>> print(a)
Compte de Kim : solde = 0
```

Méthode __str__

```
class Compte :

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

    def titulaire(self):
        return self.__titu
    def solde(self):
        return self.__solde

    def __str__(self) :
        return "Compte de {} : solde = {}".
               format(self.titulaire(),self.solde())
```

MIEUX : Utiliser les accesseurs

Appel à self

méthodes accesseurs

```
>>> print(a)
Compte de Kim : solde = 0
```

Méthodes mutateurs

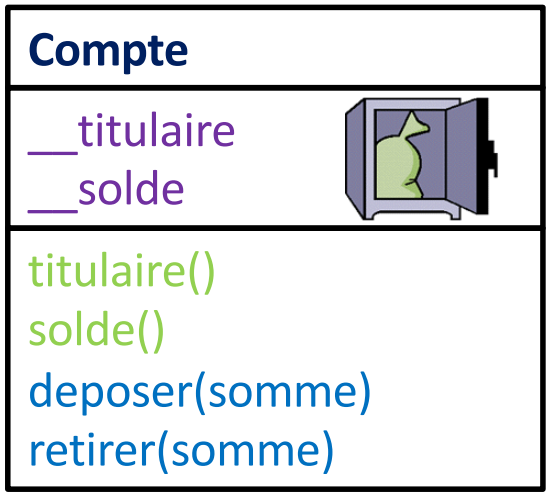
```
class Compte :

  …

  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
```



Exemple: un compte en banque

```
class Compte :

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

Compte
__titulaire __solde

```
>>> a = Compte('kim')
>>> b = Compte('charles')
>>> a.deposer(100)
100
>>> b.deposer(50)
50
>>> a.retirer(50)
50
>>> b.retirer(10)
40
```



Variable de classe

```
class Compte :

  taux_interet = 0.02
  …
```

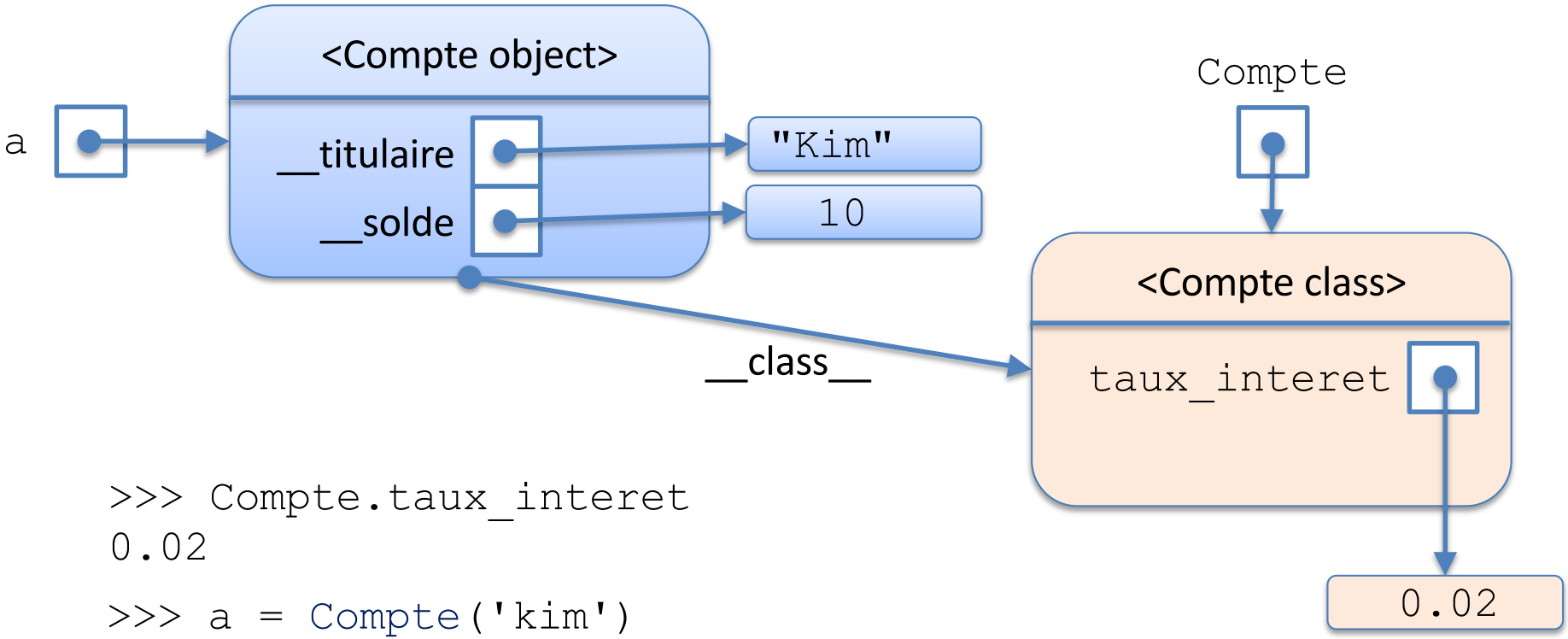
une variable pour la classe

```
>>> 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 même classe

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

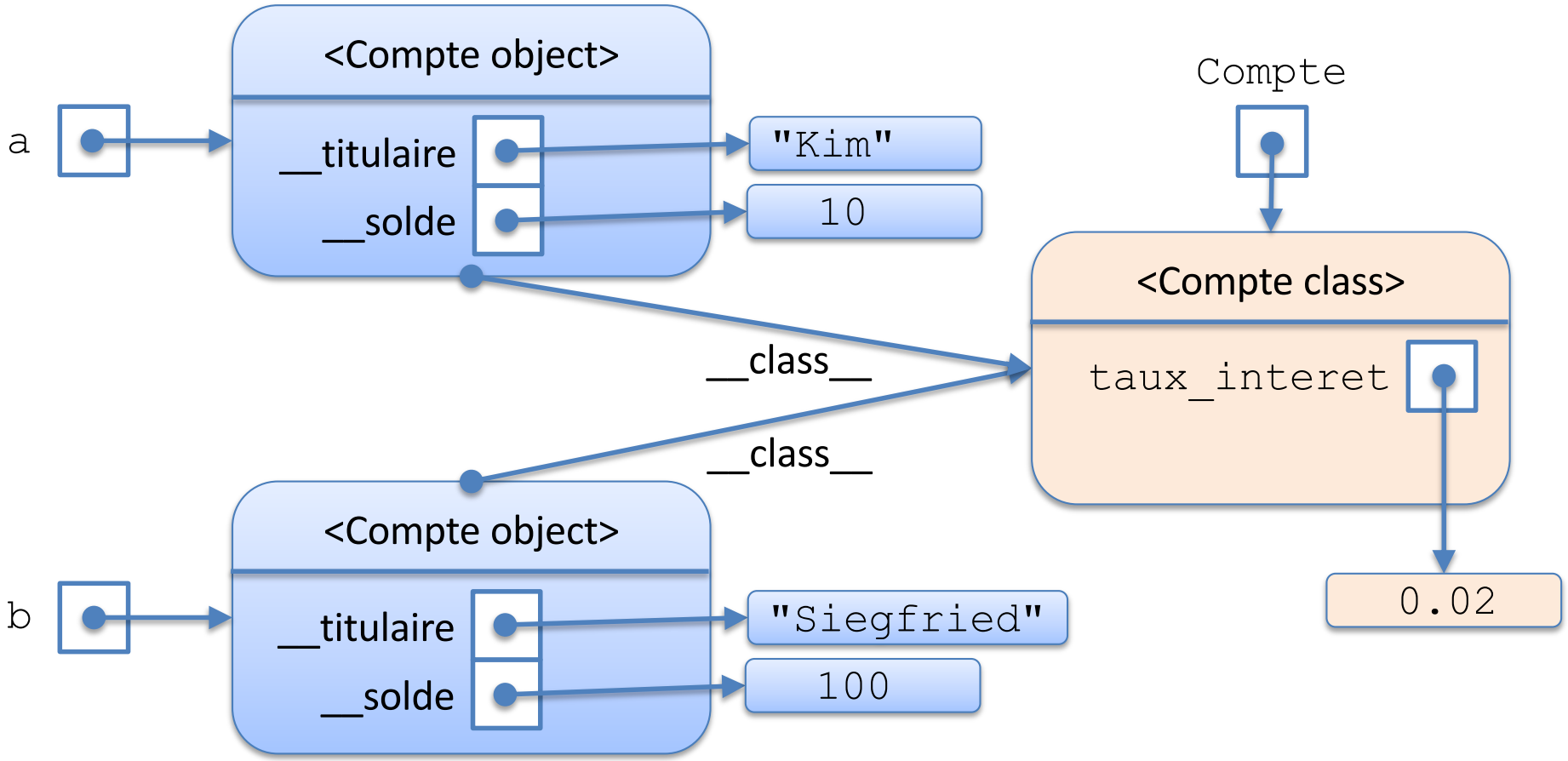
Variable de classe



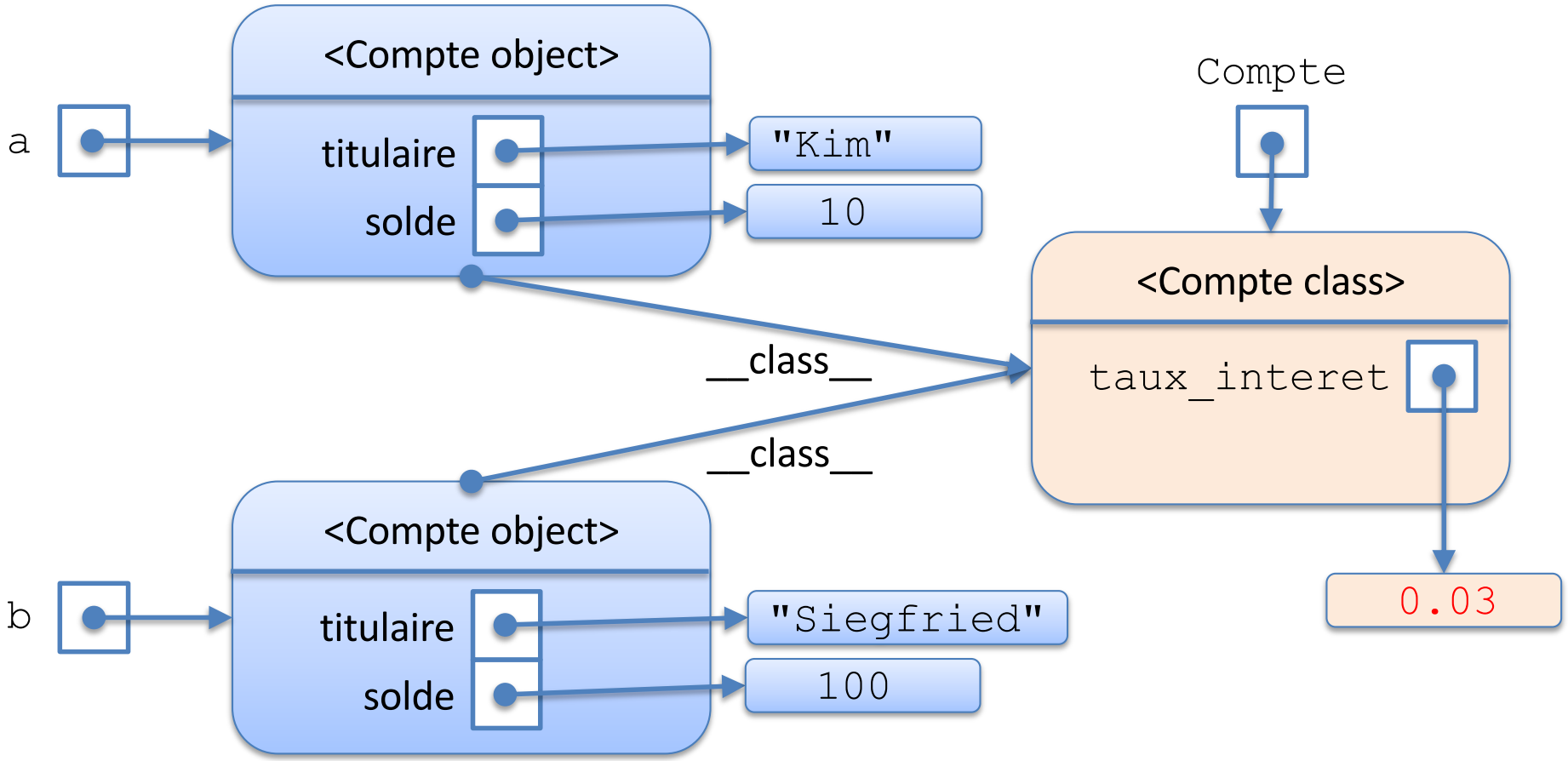

```
>>> Compte.taux_interet
0.02
>>> a = Compte('kim')
>>> a.deposer(10)
10
>>> a.taux_interet
0.02
```

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

Variable de classe



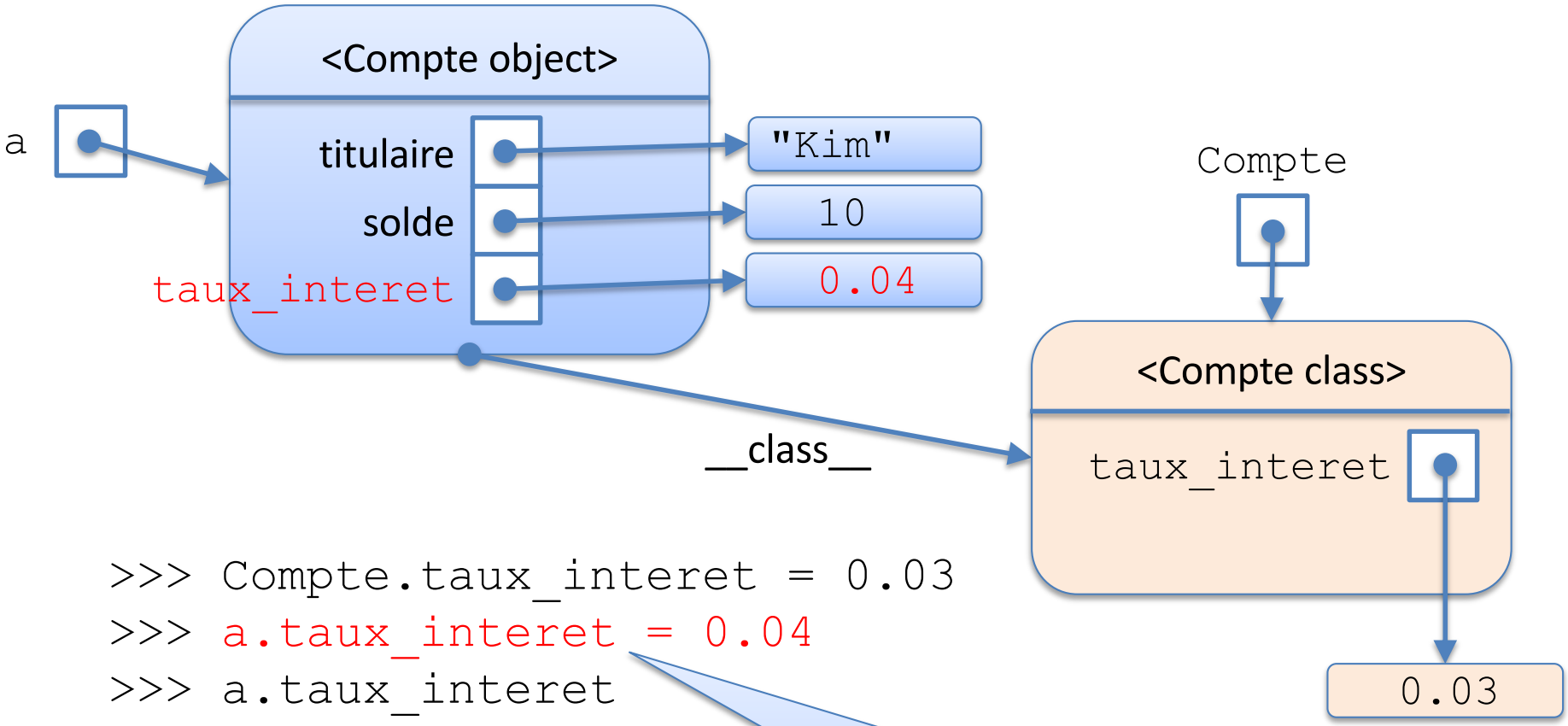
Variable de classe




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

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

« Shadowing »




```
>>> Compte.taux_interet = 0.03
>>> a.taux_interet = 0.04
>>> a.taux_interet
0.04
>>> Compte.taux_interet
0.03
>>> b.taux_interet
0.03
```

Attention!
Une variable d'instance peut avoir le même nom qu'une variable de classe

Variable de classe privée

```
class Compte :
    __taux_interet = 0.02
    …
```

variable de classe privée



```
>>> a = Compte("Kim")
>>> a.__taux_interet
AttributeError: 'Compte' object
has no attribute '__taux_interet'
>>> Compte.__taux_interet
AttributeError: 'Compte' class
has no attribute 'taux_interet'
```

Mais comment y accéder alors ?

Cf. Restructuration

Héritage

L'**héritage** est un des concepts les plus importants de la programmation orientée objet.

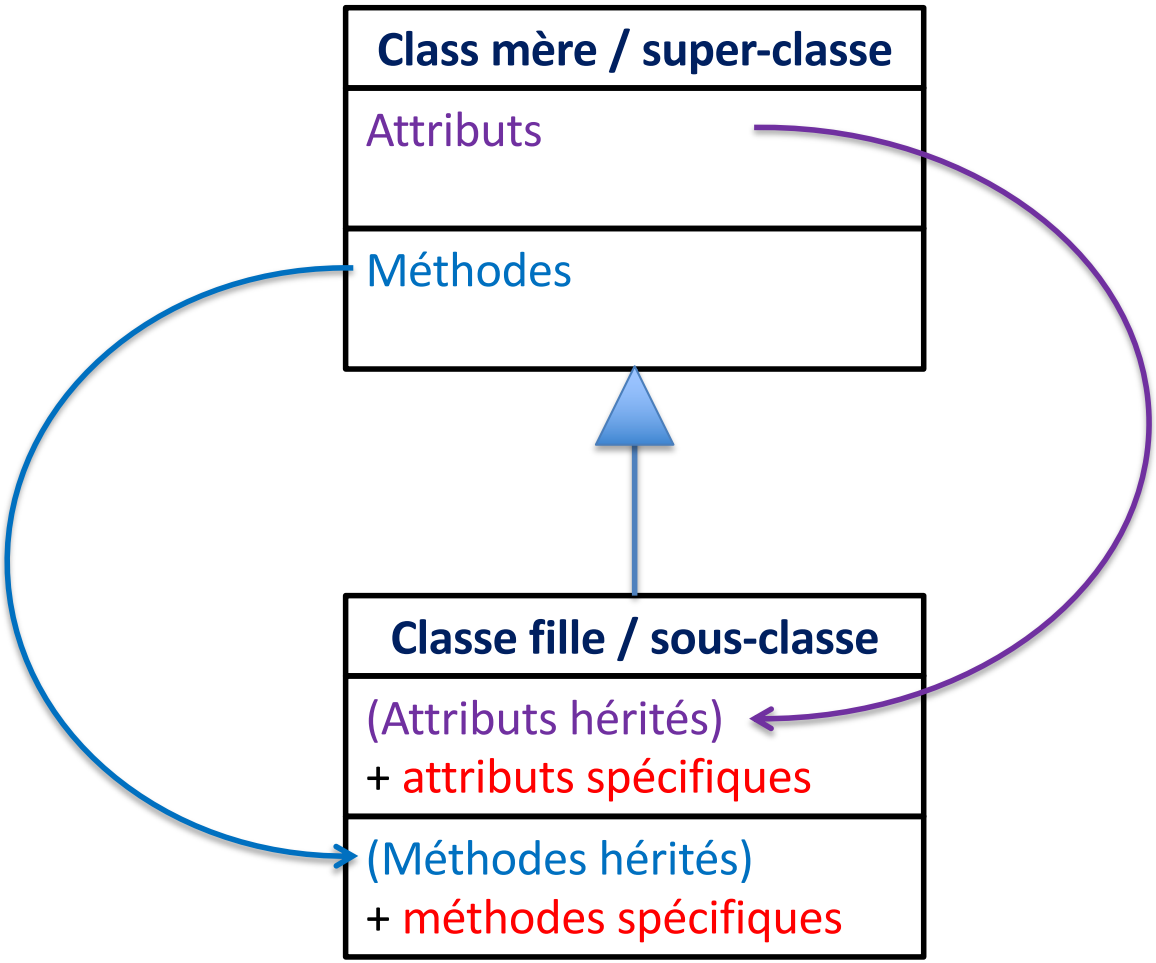
Permet de créer une nouvelle classe à partir d'une classe existante.

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

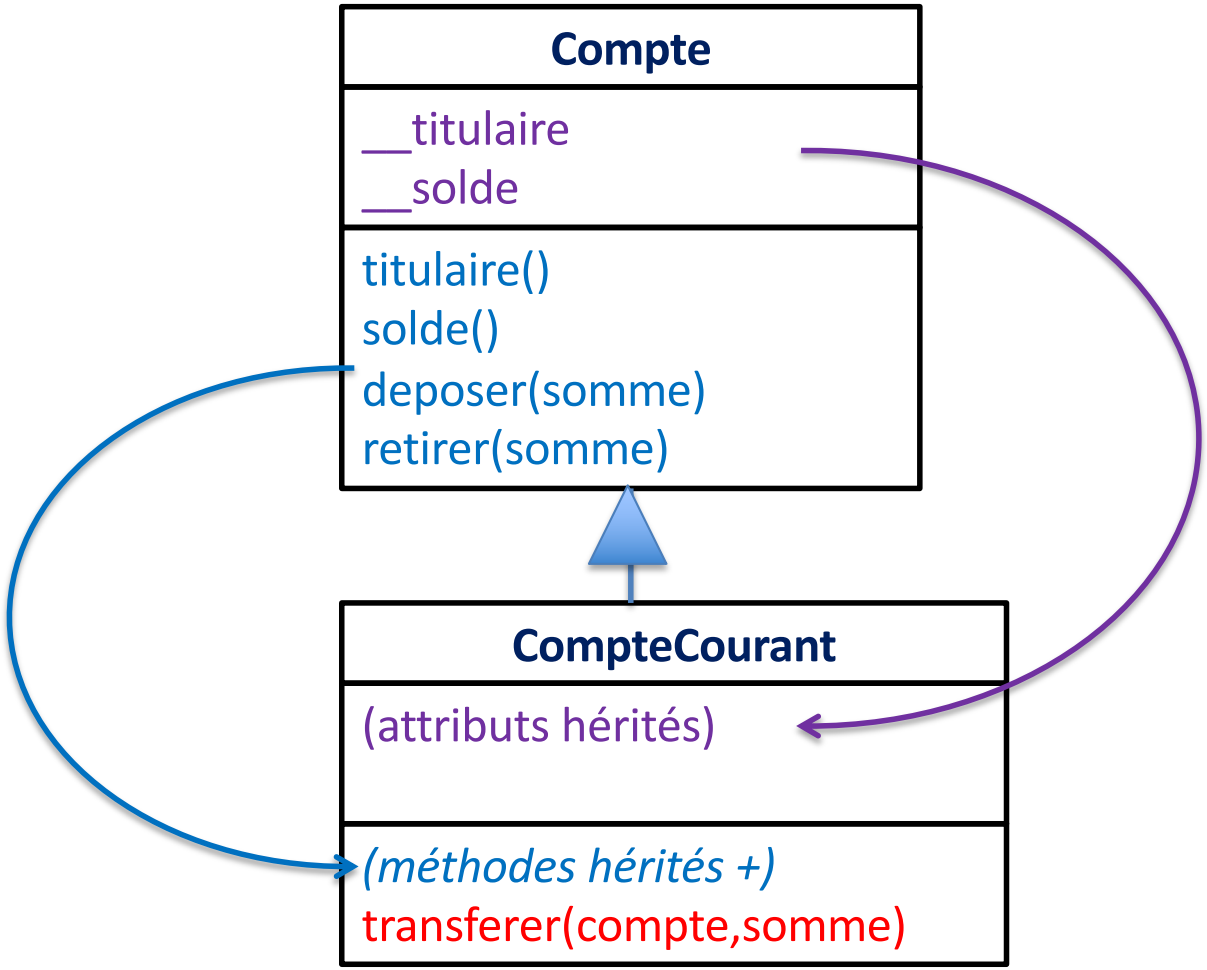
Terminologie:

- la classe existante = **super-classe** ou **classe mère**
- la nouvelle classe = **sous-classe** ou **classe fille**

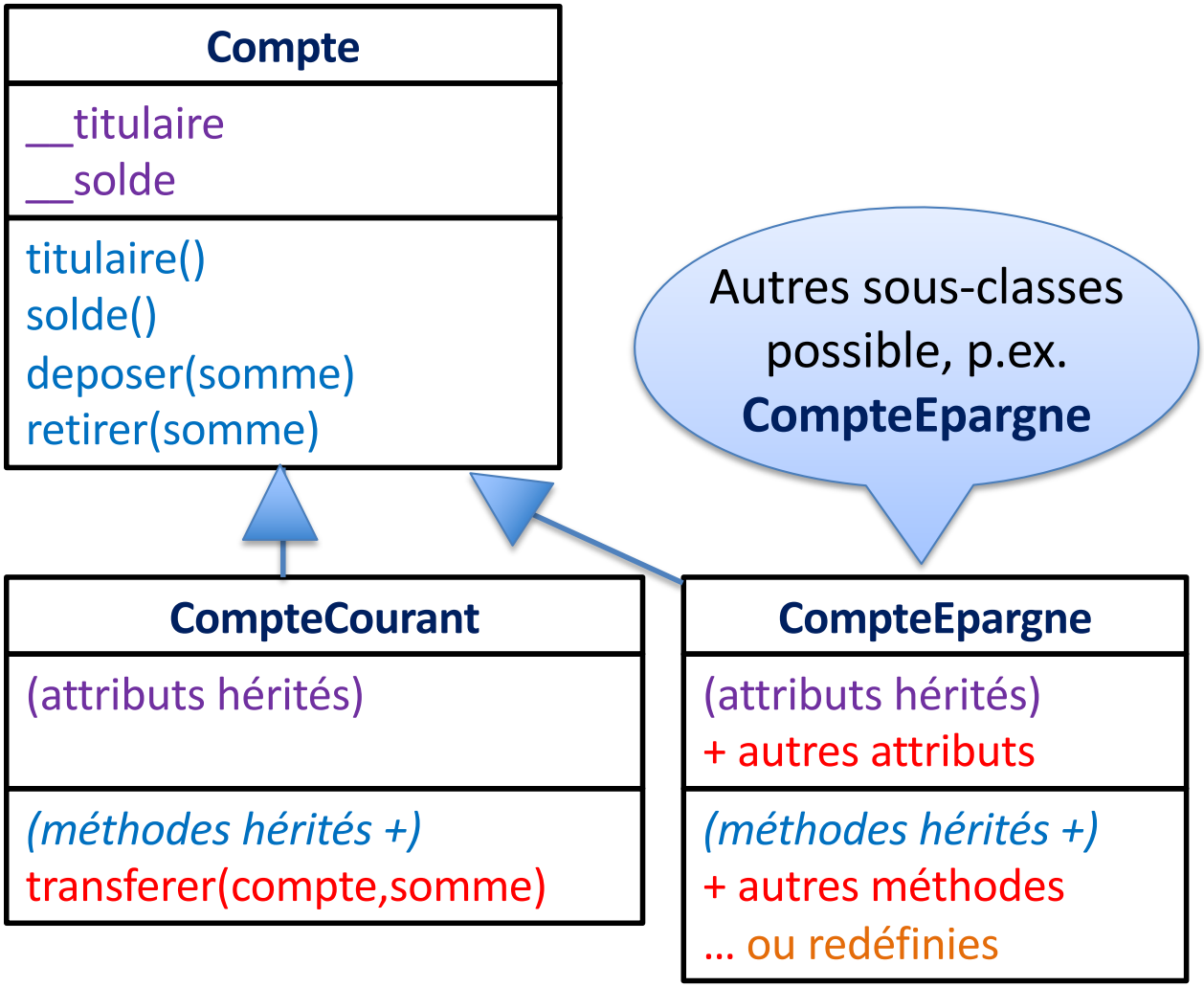
Héritage


Class mère / super-classe
Attributs
Méthodes
Classe fille / sous-classe
(Attributs hérités)
+ attributs spécifiques
(Méthodes hérités)
+ méthodes spécifiques

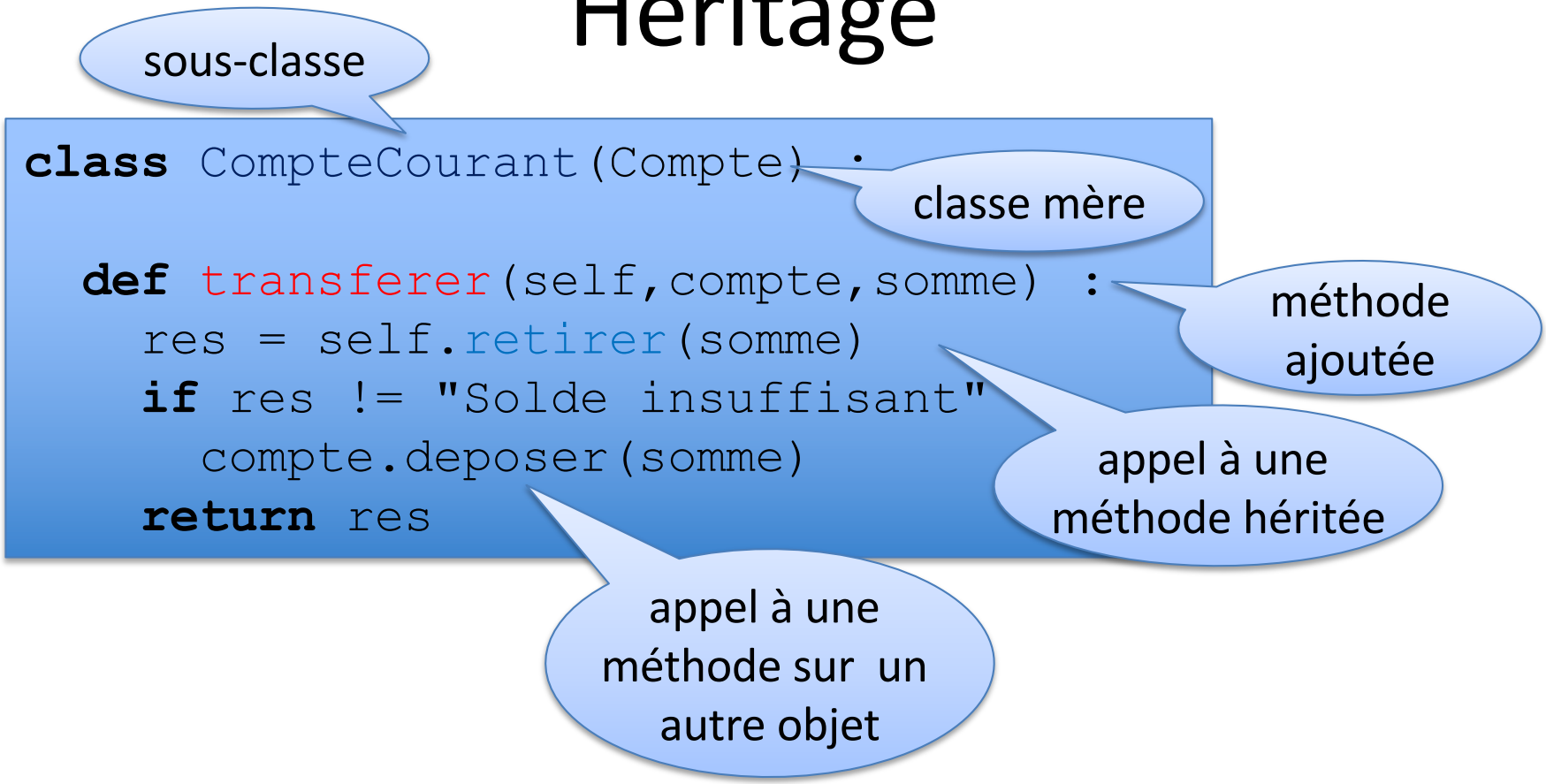
Exemple


Compte
__titulaire
__solde
titulaire()
solde()
deposer(somme)
retirer(somme)
CompteCourant
(attributs hérités)
(méthodes hérités +)
transferer(compte,somme)

Exemple


Compte
__titulaire
__solde
titulaire()
solde()
deposer(somme)
retirer(somme)
Autres sous-classes possible, p.ex. CompteEpargne
CompteCourant
(attributs hérités)
(méthodes hérités +)
transferer(compte,somme)
CompteEpargne
(attributs hérités)
+ autres attributs
(méthodes hérités +)
+ autres méthodes
... ou redéfinies

Héritage


sous-classe
class CompteCourant(Compte) :
classe mère
def transferer(self,compte,somme) :
méthode ajoutée
res = self.retirer(somme)
if res != "Solde insuffisant"
compte.deposer(somme)
appel à une méthode héritée
return res
appel à une méthode sur un autre objet

Héritage

```
class CompteCourant(Compte) :

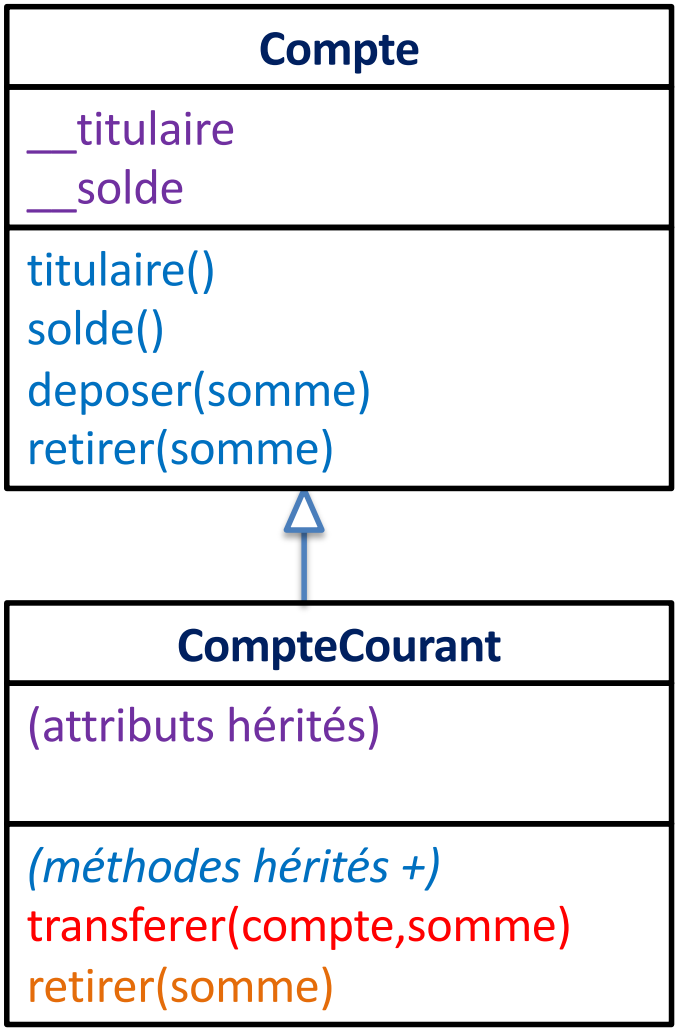
  def transferer(self,compte,somme) :
    res = self.retirer(somme)
    if res != "Solde insuffisant" :
      compte.deposer(somme)
    return res
```

```
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 classe mère*



Héritage

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

  def transferer(self,compte,somme) :
    …

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

Ajout d'une nouvelle variable de classe

Redéfinition d'une méthode d'instance

appel à la méthode héritée dans la super-classe

Héritage

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

  def transferer(self,compte,somme) :
    …

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

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

Héritage

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

  def transferer(self,compte,somme) :
    …

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

Pourquoi pas?

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

RecursionError: maximum recursion depth exceeded

Appel à super()

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

  def transferer(self,compte,somme) :
    …

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

Meilleure solution

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

super()

- `super()` permet de référer à une classe mère sans devoir la nommer explicitement.
- Souvent utilisé pour étendre une méthode de la super-classe, par exemple:

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

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

  …
```

```
compte_kim = CompteCourant("Kim","ING")
print(compte_kim)
Compte de Kim : solde = 0; banque = ING
```