Bachelier en Informatique de Gestion

Programmation Orientée Objet

Enseignement supérieur économique de type court

Code FWB : 7525 21 U32 D3

Code ISFCE : 4IPO3

= BUROTIX ()

Table des matières

- Introduction
  - 00. Propos liminaires
  - 01. Programmation procédurale : rappel
- Concepts de Programmation Orientée Objet
  - 11. Programmation orientée objet : bases
  - 13. Programmation orientée objet : aspects avancés
  - 17. Patron de conception (design pattern)
- Applications de la POO
  - 21. Graphical User Interface

= BUROTIX ()

Partie 13 :
Programmation Orientée Objet
Aspects Avancés

Syllabus & Exercices

13. POO : Aspects Avancés

- 13-1 Manipulation d'objets
  - 13-11 Comparaison
  - 13-12 Opération
- 13-3 Association entre classes
  - 13-33 Association forte
  - 13-34 Association faible
  - 13-35 Assoc. bi-directionnelle
  - 13-37 Dépendance
- 13-5 Famille de classes
  - 13-51 Héritage
    - Taxonomie
    - Super-classe, sous-classe
  - 13-52 Redéfinition
    - Class abstraite
    - Héritage simple
    - Héritage multi-niveau
    - Héritage hiérarchique
  - 13-55 Polymorphisme
    - Surcharge
  - 13-57 Héritage multiple
- 13-9 Révisions
  - 13-91 HTML builder

= BUROTIX ()

Chapitre 13-1 Manipulation d'objets

comparaison d'objets

opération sur les objets

Section 13-11 : Comparaison d'objets

__eq__

__gt__

__lt__

__ge__

__le__

__ne__

Egalité entre objets

```
class NokiaPhone :

  def __init__(self,s,p,t) :
    self.marque = "Nokia"
    self.serie = s
    self.poids = p
    self.taille = t

  …

  def __eq__(self, other) :
    return (self.marque == other.marque)      \
           and (self.serie == other.serie)    \
           and (self.poids == other.poids)    \
           and (self.taille == other.taille)

nokia_kim = NokiaPhone(5110,170,"132x47.5x31")
nokia_tom = NokiaPhone(5110,170,"132x47.5x31")
print(nokia_kim == nokia_tom)
```

True

Méthodes magiques

		Appelé “magiquement” par
`__init__`	***méthode d’initialisation***	**le constructeur d’une classe**
`__str__`	***conversion en string***	**str() , print()**
`__eq__`	***égalité***	**==**
`__lt__`	*inférieur à*	<
`__ge__`	*supérieur ou égal à*	>=
`...`	*autres comparateurs*	<= ,!= , >, …

= BUROTIX ()

Exo 13-11-03 : jouons avec les dates

Date
+ jour: int + mois: int + annee: int

- cf diagramme de classe UML ci-contre
- implémenter cette classe en Python, en privatisant les attributs.
- constructeur :
  - prévoir un dispositif pour éviter les dates impossibles (du genre 32/14/2020)
    - Génération d'une erreur : instruction `raise`
- méthode `__str__()`
  - afficher la date sous la forme "25 janvier 2023"
  - noms des mois définis comme attribut de classe à l'aide d'une liste
- Méthode `__repr__()`
  - afficher la date sous la forme "20230125"
- méthode `__lt__()`
  - comparer deux dates
  - `d1 < d2` renvoie `True` ou `False`
- *Référence : https://info.blaisepascal.fr/nsi-exercices-poo*

= BUROTIX ()

Exo 13-11-05 : Comparons des formes géométriques

- Posons que deux rectangles sont égaux ssi ils ont la même largeur et la même longueur.
- Ecrivez la classe `Rectangle` qui répond au test suivant :

```
r1 = Rectangle( 5 , 3 )
r2 = Rectangle( 5 , 3 )
r3 = Rectangle( 5 , 4 )
print( r1 == r3 ) # False
print( r1 == r2 ) # True
```

- Tuyau : définir la méthode magique `__eq__`

= BUROTIX ()

Exo 13-11-06 : Comparons des formes géométriques

- Posons que deux rectangles sont égaux ssi ils ont la même surface
- Ecrivez la classe `Rectangle` qui répond au test suivant :

```
r1 = Rectangle( 4 , 9 )
r2 = Rectangle( 6 , 6 )
r3 = Rectangle( 5 , 4 )
print( r1 == r3 ) # False
print( r1 == r2 ) # True
```

Exo 13-11-07 : Comparons des formes géométriques

- Posons qu'un rectangle R1 est plus grand qu'un rectangle R2 ssi la surface de R1 est plus grande que la surface de R2.
- Ecrivez la classe Rectangle qui répond au test suivant :

```
r1 = Rectangle( 4 , 9 )
r2 = Rectangle( 6 , 6 )
r3 = Rectangle( 5 , 4 )
print( r1 >= r3 ) # True
print( r1 == r2 ) # True
```

- Tuyau : définir la méthode magique `__ge__`

Section 13-12 : Opérateur

"surcharge d'opérateurs" __add__

__sub__

__mul__

Opérations sur les objets

- La surcharge d'opérateurs nous permet d'utiliser des opérateurs standards tels que +, -, *, … sur nos classes.
- Nous pouvons ainsi additionner, soustraire, multiplier des objets ... à condition d'avoir défini l'algorithme approprié.

Opérations sur les objets

- Référence : https://www.cosmiclearn.com/lang-fr/python-overload.php

Méthodes magiques

		Appelé “magiquement” par
`__add__`	*addition*	+
`__sub__`	*soustraction*	-
…	*autres opérateurs*	* , // , / , % , **, …

Chapitre 13-3 Association entre classes

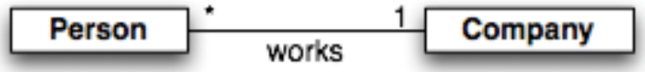
agrégation
composition
association directionnelle faible
association directionnelle forte
association bi-directionnelle
auto-association
dépendance

Association, composition, agrégation

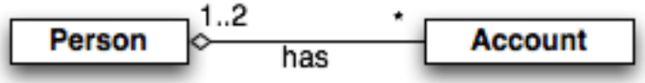
- Une **association** est une relation sémantique entre des classes qui définit un ensemble de liens.
  - Un homme est associé à sa femme
- Une **agrégation** est une association dans laquelle il y a un lien d'appartenance entre les deux objets associés (contenant/contenu, possession, …).
  - Un homme possède un compte en banque
- Une **composition** (ou agrégation forte) est une agrégation dans laquelle la disparition du composite entraine la disparition des composants.
  - Si un arbre meurt, ses feuilles ne servent plus à rien (on ne peut pas les mettre sur un autre arbre, au contraire de roues sur une voiture)
- Il s'agit surtout d'une différence sémantique qui impliquera des changements dans votre implémentation au niveau du cycle de vie des objets.

Association, composition, agrégation

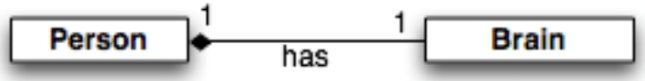
Association : Une personne travaille pour une et une seule compagnie



Agrégation : Une personne possède entre 0 et n comptes

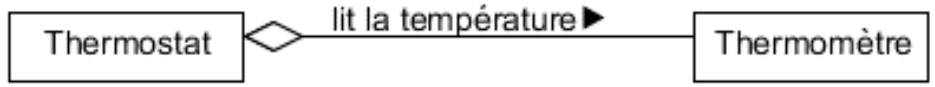


Composition : Une personne a un et un seul cerveau

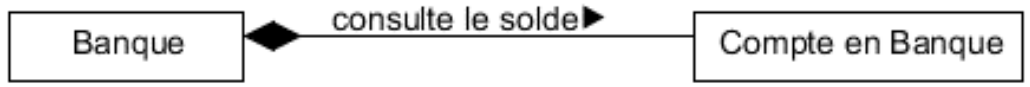


- Cardinalités :
  - *, *n*, *m*..*, où *n*>*0* et *m*>=*0*
  - Par défaut, la cardinalité est *un*.
  - Dans une composition, la cardinalité du côté de l'agrégat ne peut être que 1 ou 0..1
- Le nom de l'association est facultatif

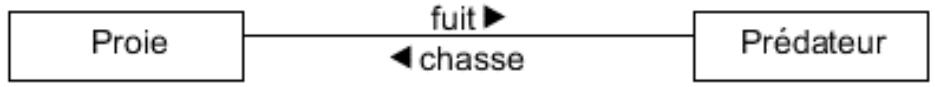
Associations entre classes



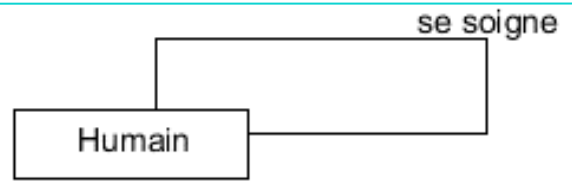
- association directionnelle faible; agrégation
  - objet « thermostat » contient objet « thermomètre »



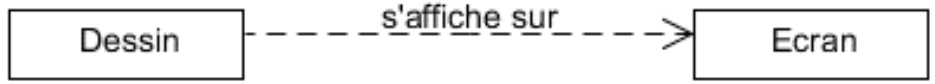
- association directionnelle forte; composition
  - objet « banque » détruit => objet « compte en banque » détruit



- association bidirectionnelle
  - réciprocité



- auto-association
  - l'objet s'utilise lui-même (pour une autre fonction)

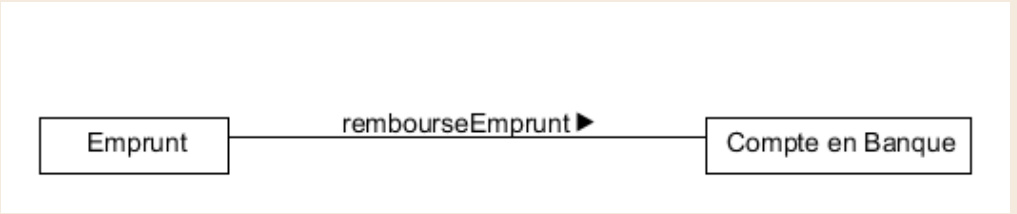


- dépendance
  - méthode « s'affiche sur » exige argument : objet « écran »

Il s'agit ici de sémantique, et non de POO !

= BUROTIX ()

Messages entre objets



- Le « message » est un outil de communication entre objets.
- « Message » car il y a
  - Expéditeur
  - Destinataire
- Tous les objets peuvent collaborer entre eux en s'envoyant des messages.
- Développer une application "OO" consiste en :
  - définir un réseau de classes, donc d'objets
    - chacun chargé d'une tâche spécifique
  - en interactions par message
    - soit locaux
    - soit via internet
  - formant un tissu de relations

= BUROTIX ()

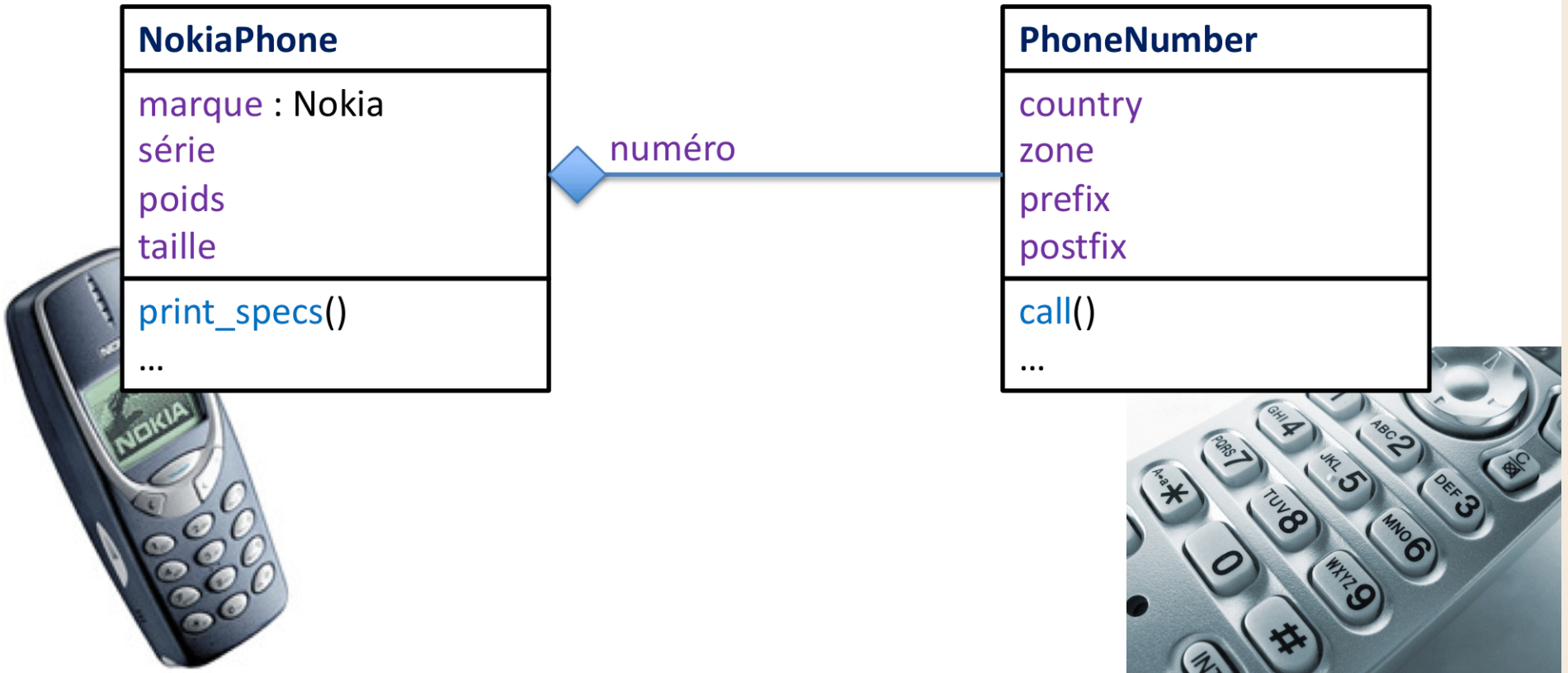
Section 13-33 : Association directionnelle forte

= BUROTIX ()

Composition de classes

Une classe peut être composé d'une autre.

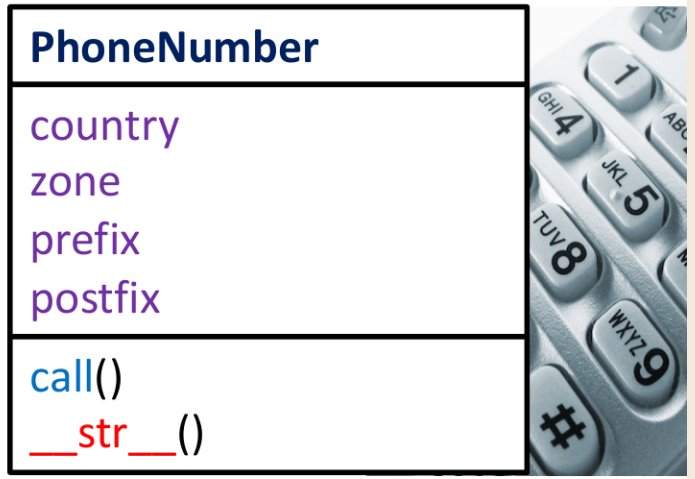
Un objet peut contenir (une référence vers) un autre.



= BUROTIX ()

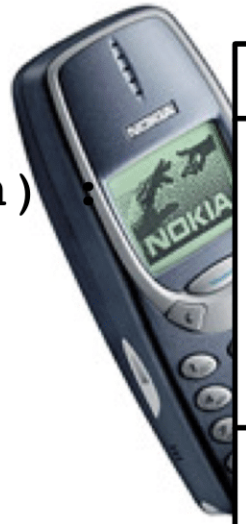
Composition de classes

```
class PhoneNumber :
  def __init__(self,c,z,pr,po):
    self.country = c
    self.zone = z
    self.prefix = pr
    self.postfix = po

  def call(self):
    # code pour appeler ce numéro de téléphone

  def __str__(self):
    return "+{}(0){}/{}{}".format(self.country,
           self.zone,self.prefix,self.postfix)

my_number = PhoneNumber(32,10,4,79111)
print(my_number) ➔ +32(0)10/479111
```



Composition de classes

```
class NokiaPhone :

  def __init__(self,s,p,t,n) :
    self.marque = "Nokia"
    self.serie = s
    self.poids = p
    self.taille = t
    self.numero = n

  …

  def print(self)
        self.print_specs()
        print(self.numero)

my_phone = NokiaPhone(5110,170,"132x48x31",my_number)
```



NokiaPhone
marque : Nokia série poids taille numero
… print ()

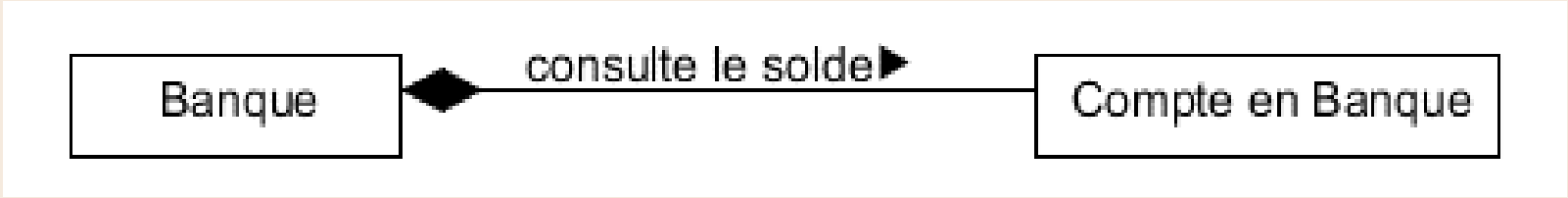
Nouvel attribut

Nouvelle méthode

Exo 13-33-07 : Banque

```
b = Banque("Belfius")
b.ajouter_compte("Kim")
b.ajouter_compte("Sandra")
print(b)
   banque de nom : Belfius
   compte de Kim
   compte de Sandra
b = None
   compte de Kim détruit
   compte de Sandra détruit
   banque détruite
```

- Ecrivez les classes `Banque` et `Compte` correspondant à ce scénario.
- Remarque : le code appelant ne crée pas les objets `Compte`.

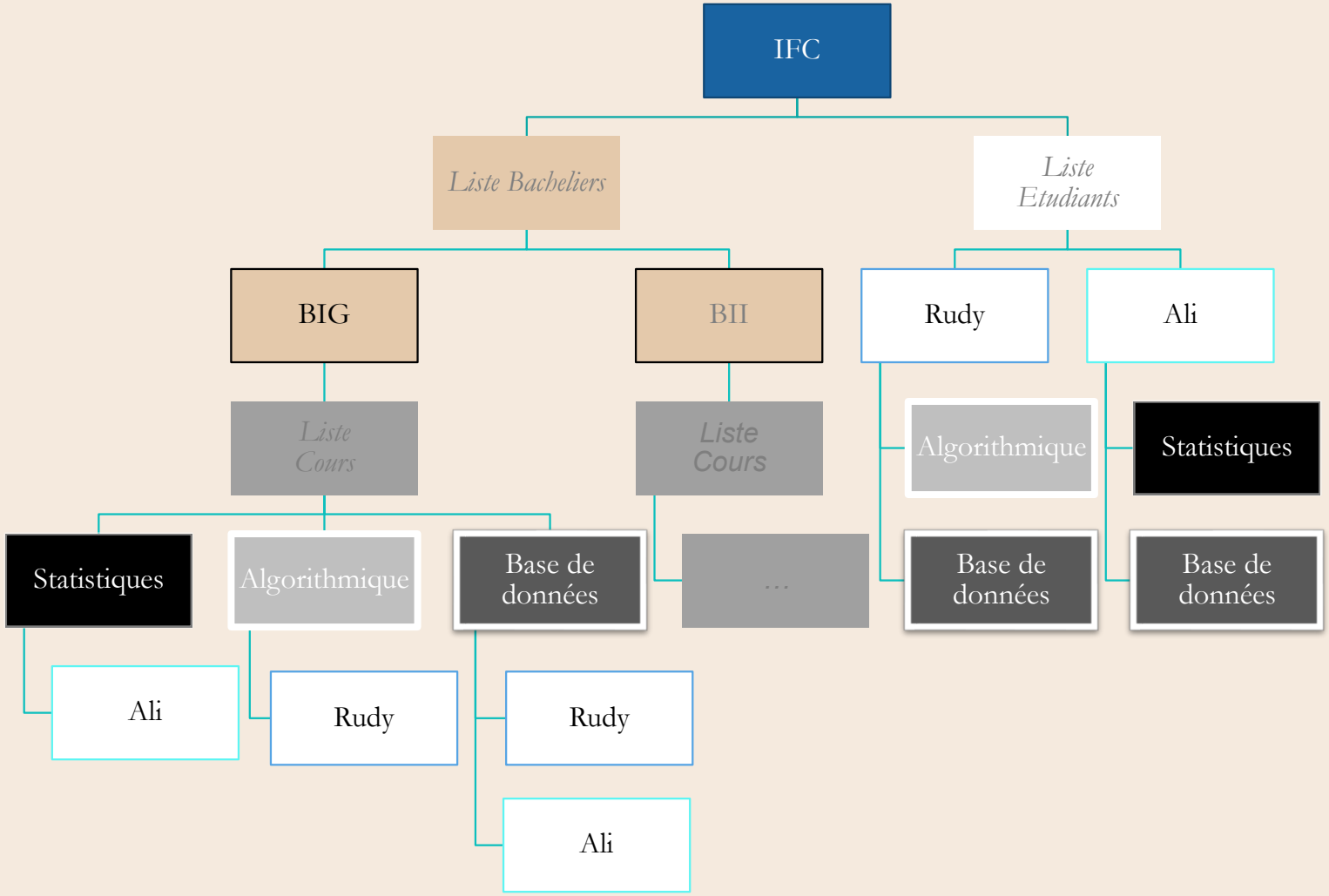


Exo 13-33-09 : École

- Ecrivez le POO d'une école
  - L'école organise des bacheliers, ceux-ci constitués d'un ensemble de cours.
  - L'école organise un cours dans un bachelier en y inscrivant les étudiants.
  - Un étudiant peut se désinscrire d'un cours.
  - L'école peut renvoyer un étudiant.
  - L'écoel peut supprimer un cours.
  - Cf schéma, slide suivant
- Scénario d'utilisation
  - Créer une école "IFC".
  - Créer un bachelier "BIG" avec trois cours:
    - Statistiques
    - Algorithmique
    - Database
  - Créer deux étudiants : Rudy et Ali.
  - Créer l'inscription de chaque étudiant à deux cours.
    - Rudy : algorithmique, database
    - Ali : database, statistiques
  - Imprimer l'état des inscriptions dans le bachelier BIG
    - par cours
    - par étudiant
  - Supprimer le cours d'Algorithmique.
  - Désinscrire l'étudiant Rudy du cours de Database
  - Renvoyer l'étudiant Ali.
  - Ajouter l'étudiant Youssef, inscrit à tous les cours.
  - Imprimer l'état des inscriptions.

= BUROTIX ()

Exo 13-33-09 : École



= BUROTIX ()

Exo 13-33-09 : École

nom : IFC

liste bacheliers:

bachelier : BIG

cours : statistiques

student : Rudy

student : Ali

cours : algorithmique

student : Rudy

cours : database

student : Rudy

student : Ali

bachelier : BII

cours : interface

cours : électronique

student : Rudy

liste étudiants:

student : Rudy : algorithmique, database, électronique, statistiques

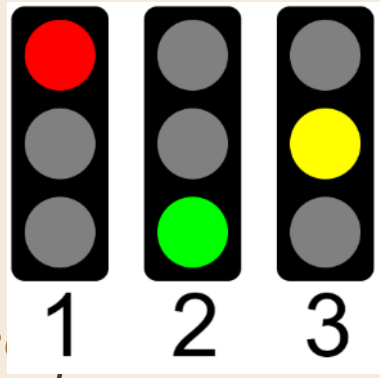
student : Ali : database, statistiques

= BUROTIX ()

Section 13-34 : Association directionnelle faible

= BUROTIX ()

Exo 13-34-11 : one light, one car



- Programmez un feu de signalisation lié à une voiture autonome
  - Définissez la classe `Light`
  - Définissez la classe `Car`
  - Le feu commande la voiture (méthodes `start`, `stop`)
  - Corrigés : 3 versions
    - attributs publics
    - attributs privés, get/set classique
    - attributs privés, get/set décorateur)
- Output

```
Light: RED, follow Peugeot
Peugeot: waiting green light
Light: GREEN, follow Peugeot
Peugeot: running on the road
```

```
Light: YELLOW; follow Peuge
Peugeot : running on the road
Light : RED, follow Peugeot
Peugeot : waiting green light
Light : GREEN, follow Peugeot
Peugeot : running on the road
```

- Input

```
voiture_A = Car("Peugeot")
feu01 = Light(voiture_A)
print(feu01)
print(voiture_A)
for i in range(4):
    feu01.change()
    print(feu01)
    print(voiture_A)
```

= BUROTIX ()

Exo 13-34-13 : two cars on the road

- Deux voitures autonomes
  - Classe `Car`
  - Une voiture A est déclarée, en position 0 km, avec vitesse 120 km/h.
  - Une voiture B est déclarée, en position 10 km, avec vitesse 60 km/h.
  - Les deux voitures sont amies et liées.
  - Elles démarrent en même temps et vérifient à chaque instant laquelle est devant l'autre.

- Input

```
# on déclare la voiture A
voiture_A = Car("A")
voiture_A.speed = 120
# on déclare la voiture B
voiture_B = Car("B", 10)
voiture_B.speed = 60
# on lie les deux voitures
voiture_B.friend = voiture_A
voiture_A.friend = voiture_B
print(voiture_A)
print(voiture_B)
for i in range(6):
  # les voitures se mettent à rouler
  # duration = 1/12 h = 5 min
  voiture_A.duration = 1/12
  voiture_B.duration = 1/12
  print(voiture_A)
  print(voiture_B)
```

= BUROTIX ()

Exo 13-34-13 : two cars on the road

- Output

```
A: 120km/h, 0.0h,  0.0km, B beyond me
B:  60km/h, 0.0h, 10.0km, A behind me

A: 120km/h, 0.1h, 10.0km, B beyond me
B:  60km/h, 0.1h, 15.0km, A behind me

A: 120km/h, 0.2h, 20.0km, B beyond me
B:  60km/h, 0.2h, 20.0km, A beyond me

A: 120km/h, 0.2h, 30.0km, B behind me
B:  60km/h, 0.2h, 25.0km, A beyond me

A: 120km/h, 0.3h, 40.0km, B behind me
B:  60km/h, 0.3h, 30.0km, A beyond me

A: 120km/h, 0.4h, 50.0km, B behind me
B:  60km/h, 0.4h, 35.0km, A beyond me

A: 120km/h, 0.5h, 60.0km, B behind me
B:  60km/h, 0.5h, 40.0km, A beyond me
```

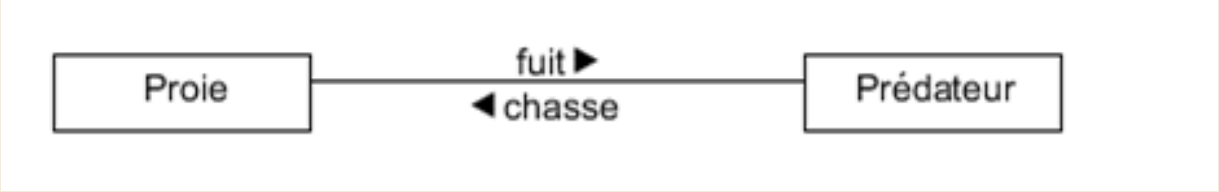
= BUROTIX ()

Section 13-35 : Association bi-directionnelle

= BUROTIX ()

Exo 13-35-23 : Jungle

- Ecrivez les classes `Prey` et `Predator` correspondant à ce scénario.
- Un seul prédateur et une seule proie.
- Remarque : le code appelant, que l'on pourrait comparer à la jungle, crée les objets `Predator` et `Prey`.



= BUROTIX ()

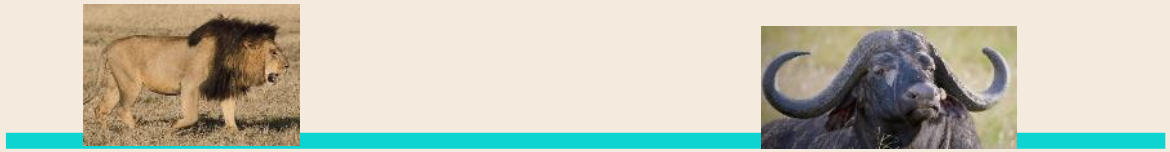
Exo 13-35-23 : Jungle

- INPUT

```
lion = Predator(0) position 0
buffle = Prey(100) position 100
lion.set_prey(buffle)
buffle.set_predator(lion)
while True:
        lion.move_to_prey()
         buffle.escape()
        if lion.prey_caught:
                print("Bon appétit !")
                break
```

- OUTPUT

```
predator : 10 / prey : quiet 100
predator : 20 / prey : quiet 100
…
predator : 80 / prey : escaping 105
predator : 90 / prey : escaping 110
predator : 100 / prey : escaping 115
predator : 110 / prey : escaping 120
predator : 120 / predator caught prey
```

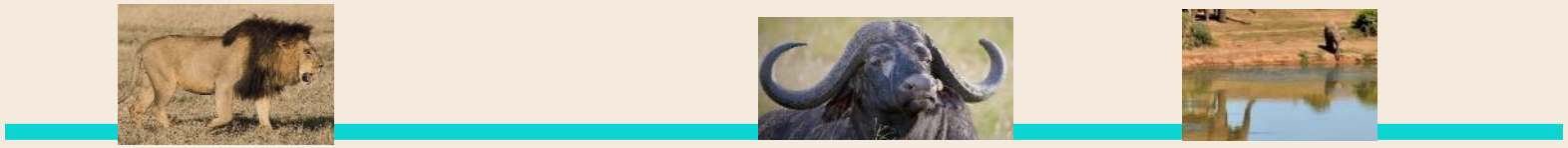


0 100

= BUROTIX ()

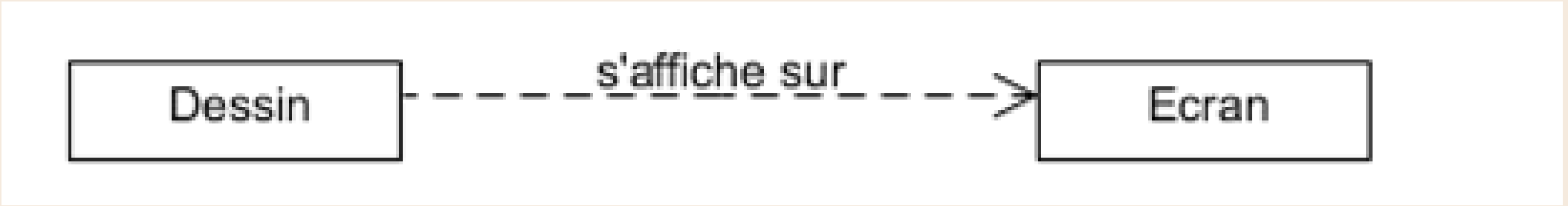
Exo 13-35-24 : Jungle

- Ecrivez les classes `Water`, `Prey` et `Predator` correspondant aux spécifications suivantes :
  - Un lac se trouve à 150m, une proie à 100m et un prédateur à 0m.
  - Le lac contient une réserve d'eau infinie.
  - Les animaux se déplacent à leur rythme vers le lac (cf figure).
  - Quand un animal rencontre le lac, il s'y arrête pour y boire.
  - Le prédateur voit la proie quand celle-ci est à moins de 50m, et se met alors à courir vers elle pour l'attraper.
  - La proie voit le prédateur quand celui-ci est à moins de 25m, et se met alors à fuir.
  - Boire ou fuir, il faut choisir … la fuite. ;-)



= BUROTIX ()

Section 13-37 : Dépendance



= BUROTIX ()

Exo 13-37-03 : afficher un point

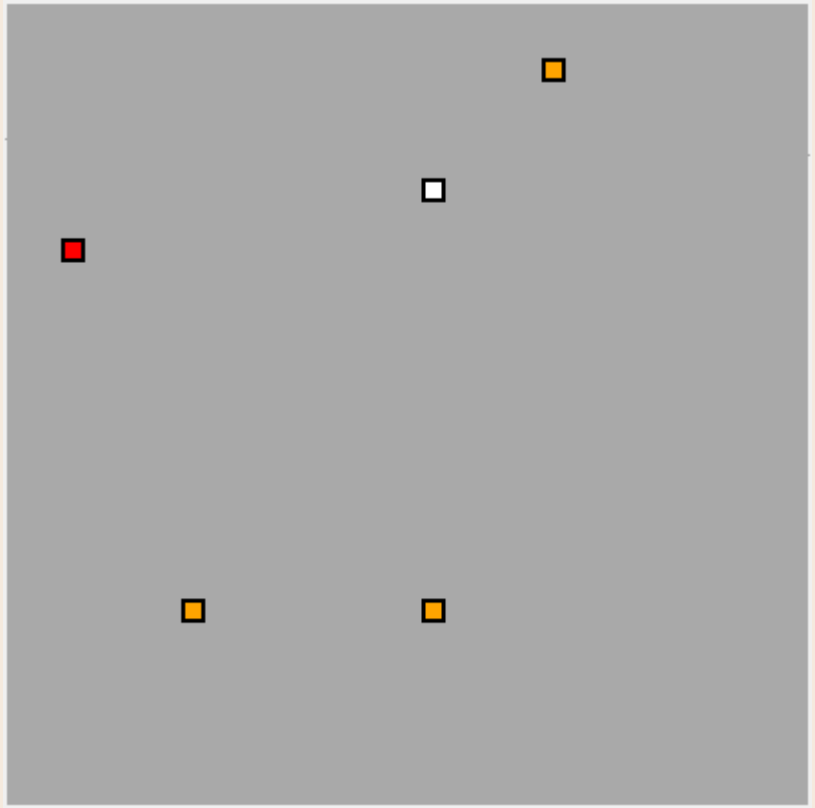
- Point de départ : la classe `Point`
- On veut afficher un objet `Point` de deux manières
  - Soit textuel
  - Soit graphique
- On crée un objet spécifiquement dédié à l'affichage
  - Soit instance de `class TextOutput`,
  - Soit instance de `class GraphicalOutput`
    - Classe basée sur le module `Tkinter`
- Télécharger et faire tourner 13-37-03.py

Exo 13-37-03 : les deux outputs

Output ? 1 pour texte, 2 pour graphique 1

```
Point(9,2) ; distance à l'origine =  9.22
Point(3,7) ; distance à l'origine =  7.62
Point(5,5) ; distance à l'origine =  7.07
Point(9,5) ; distance à l'origine = 10.30
Point(1,10) ; distance à l'origine = 10.05

Point(11,2) ; distance à l'origine = 11.18
Point(5,7) ; distance à l'origine =  8.60
Point(7,5) ; distance à l'origine =  8.60
Point(11,5) ; distance à l'origine = 12.08
Point(3,10) ; distance à l'origine = 10.44
…
```

Output ? 1 pour texte, 2 pour graphique 2



= BUROTIX ()

Exo 13-37-03 : principes

- objets `TextOutput` et `GraphicalOutput`
  - Ils gèrent l'affichage
    - Initialisé dans `.__init__()`
  - Ils bufferisent les objets `Point` à afficher
    - méthode `.append()`
  - Tout s'affiche en une fois `avec print()`
    - méthode magique `.__str__()`

- objet `Point`
  - Il est défini par deux coordonnées `x` et `y` modifiables.
    - Méthodes `.__init__()` et `.forward()`
  - L'output est fourni en paramètre à la création de l'objet.
  - Il ne se préoccupe plus de son affichage.
    - Plus de méthode magique `.__str__()`

= BUROTIX ()

Chapitre 13-5 Famille de classe

Taxonomie
Héritage
Types d'héritage
Polymorphisme

Class abstraite
Héritage multiple

Section 13-51 : Héritage

taxonomie

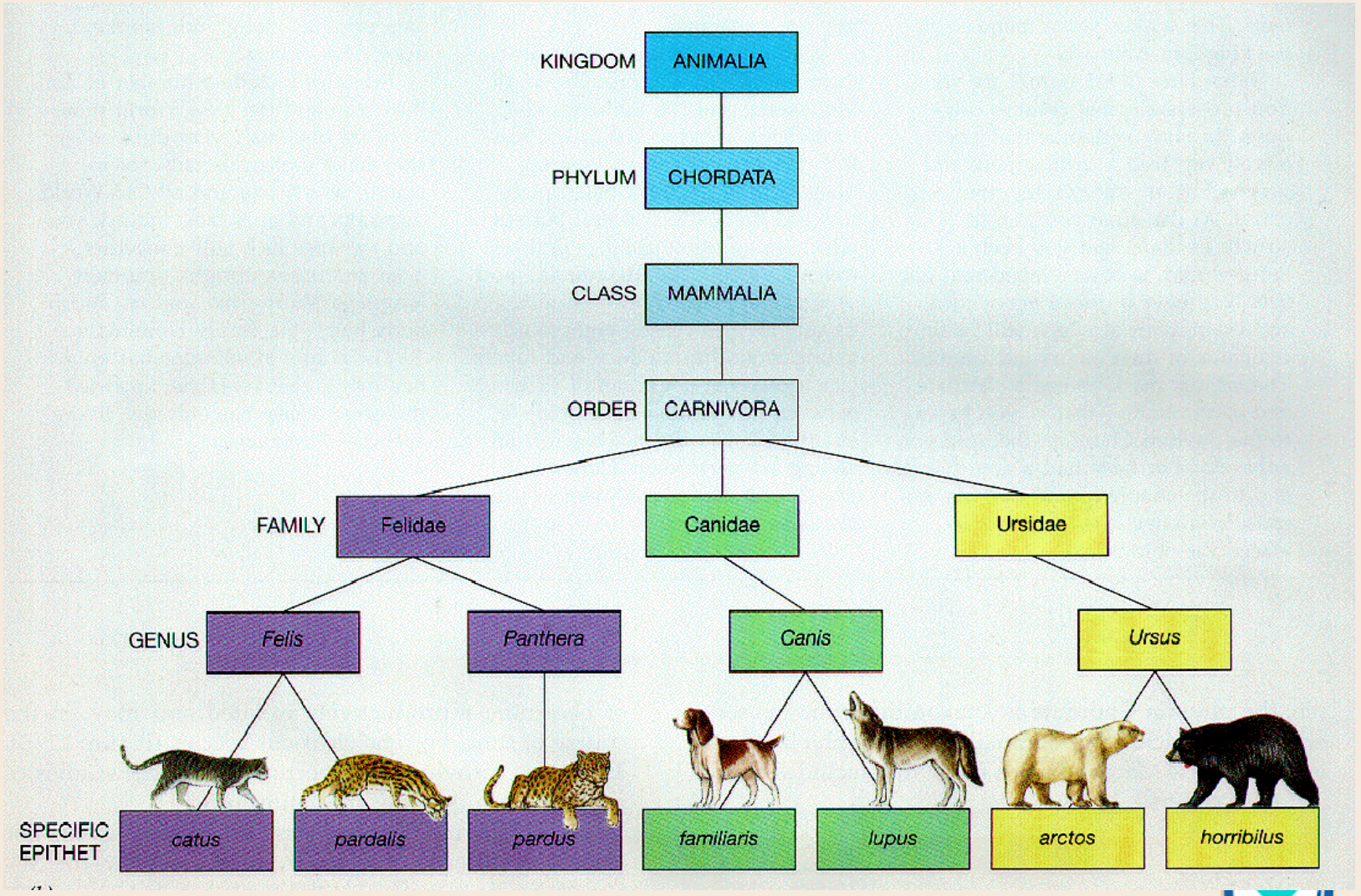
parent

enfant

À lire

Taxonomie

L'héritage permet une représentation taxonomique des classes.



= BUROTIX ()

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.

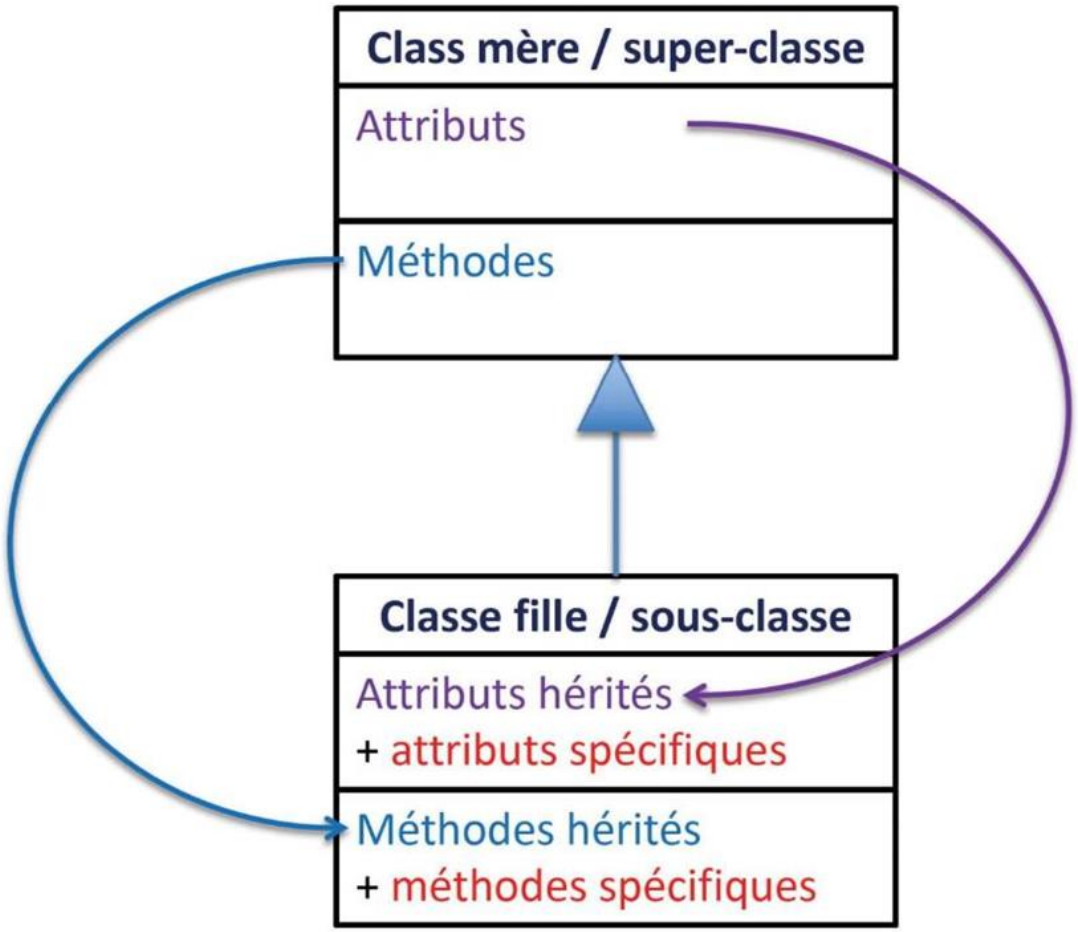
La nouvelle classe “hérite” tous les attributs et méthodes de la classe existante.

Terminologie:

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

BUROTIX ()

Héritage



BUROTIX ()

Avantages et inconvénients de l'héritage

- Avantages de l'héritage
  - Économie de représentation
  - Clarification du problème
  - Ré-emploi
  - Stabilité
  - Framework de développement
- Inconvénients de l'héritage
  - Dispersion du code
  - Lisibilité affaiblie

Exo 13-51-01 : le compte en banque "Renaissance"

- A la Renaissance, on ne pouvait que déposer ou retirer de l'argent à la banque
- Class "Compte"
  - Attributs
    - Titulaire
    - Solde
  - Méthodes
    - Déposer()
    - Retirer()

- Scénario

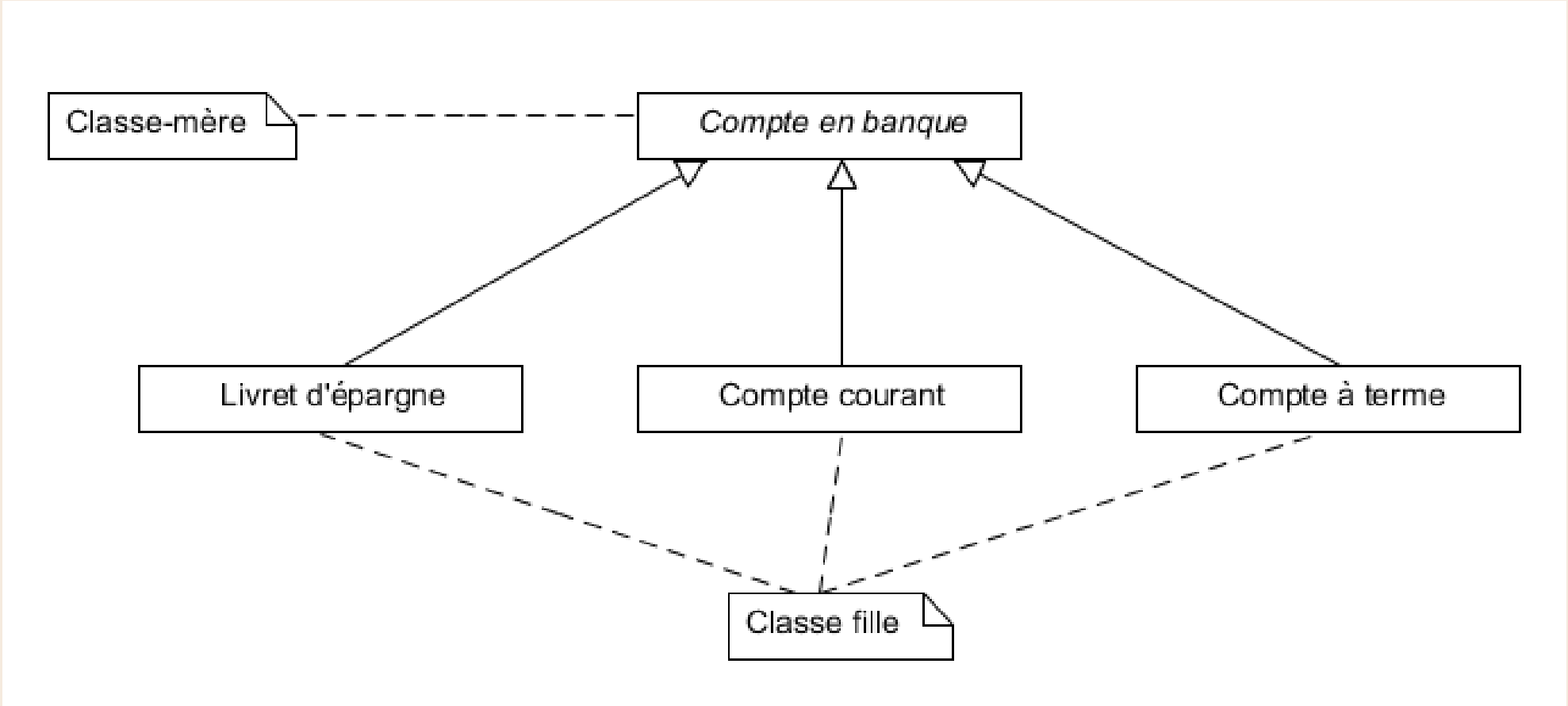
```
kim_c = Compte("Kim")
kim_c.deposer(1000)
print(kim_c)
```

- Output

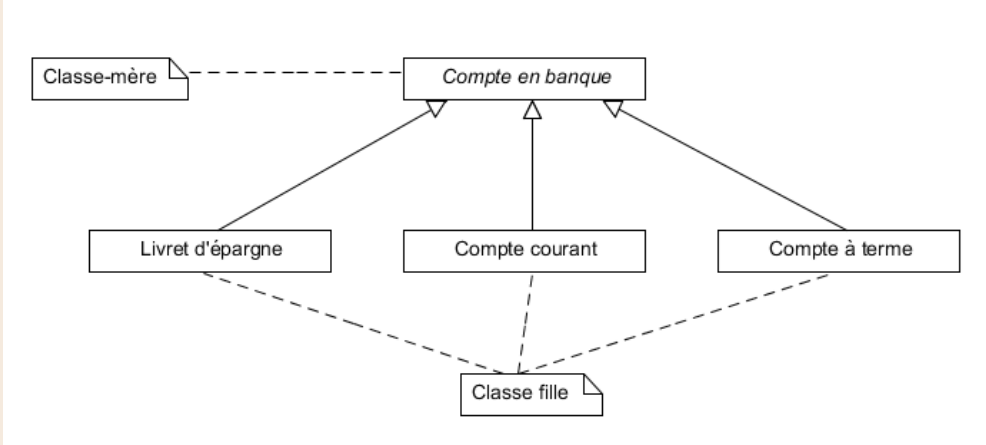
```
owner : Kim;
balance : 1000
```

= BUROTIX ()

Héritage



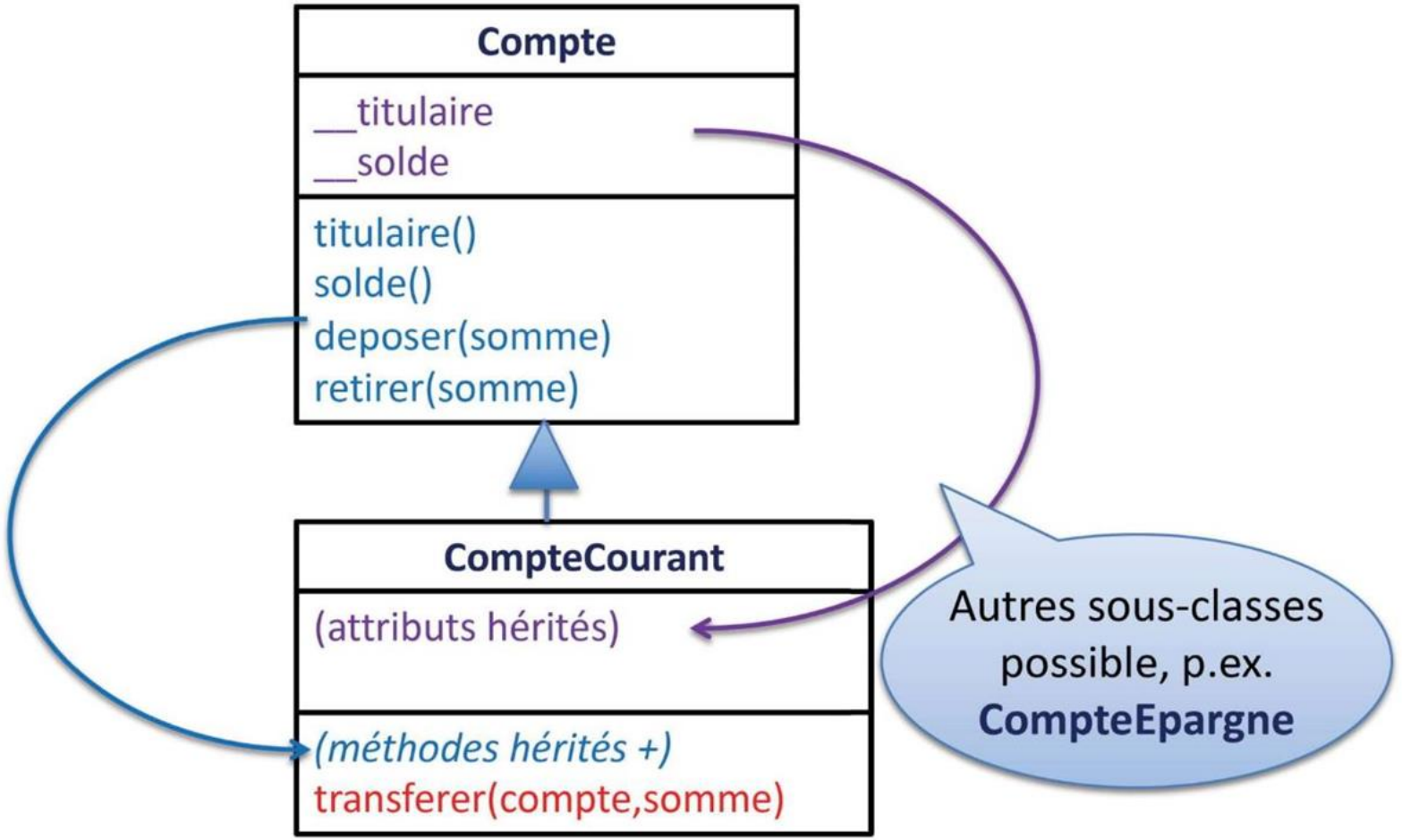
Héritage



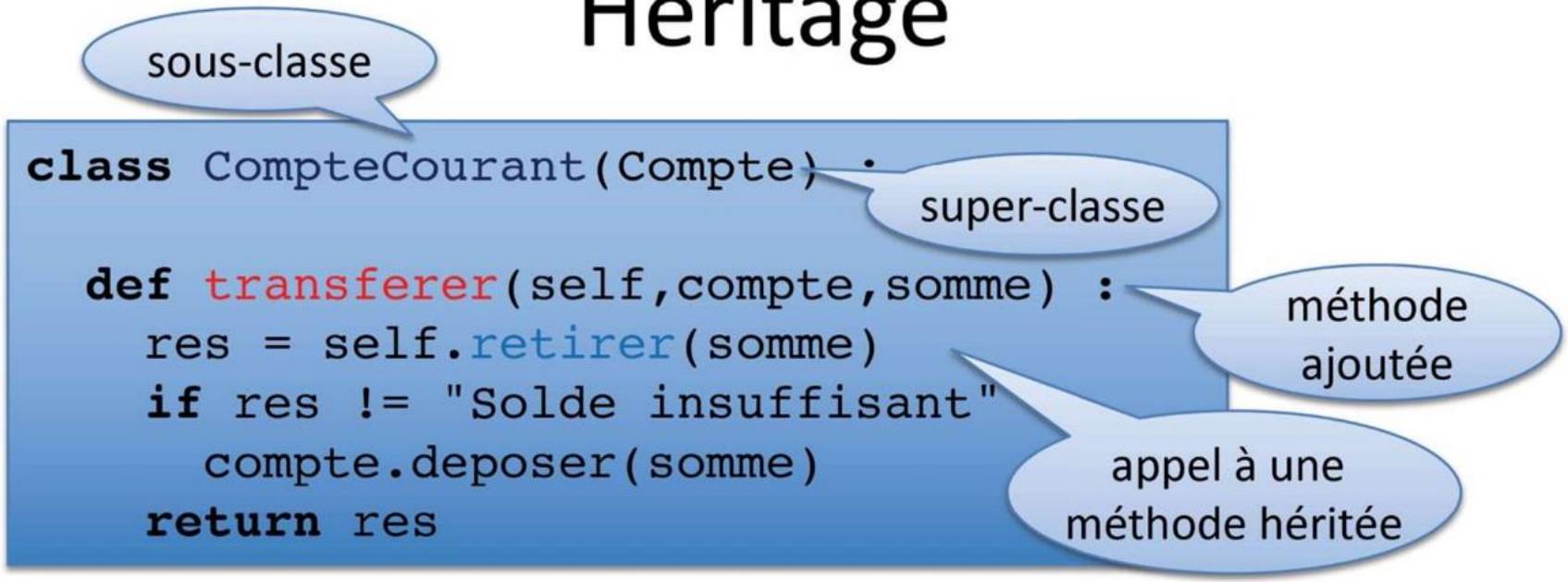
- classe-mère
  - « Compte en banque »
- classes-filles
  - « Livret d'épargne », « compte courant » et « compte à terme »
- Les classes-filles héritent de tous les attributs et méthodes de la classe-mère
  - par exemple, le livret d'épargne comme le compte courant et le compte à terme ont tous un solde.
- Les classes-filles vont par contre implanter des attributs et méthodes plus spécialisées.
  - par exemple, le livret d'épargne a un taux d'intérêt plus élevé que le compte courant
  - ou, le livret ne peut jamais avoir un solde négatif.

= BUROTIX ()

Exemple



Héritage



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
```

BUROTIX ()

Exo 13-51-02 : le compte en banque moderne

- Le compte en banque moderne permet également de transférer de l'argent sur un autre compte.
- Class "CompteCourant"
  - Enfant de "Compte"
  - Méthode
    - `.transférer()`

- Scénario

```
kim_c = Compte("Kim")
kim_c.deposer(1000)
glo_c = Compte("Glo")
kim_c.transferer(glo_c, 150)
print(kim_c)
print(glo_c)
```

- Output

```
owner:Kim; balance:850
owner:Glo; balance:150
```

= BUROTIX ()

Section 13-52 : Redéfinition des attributs et des méthodes

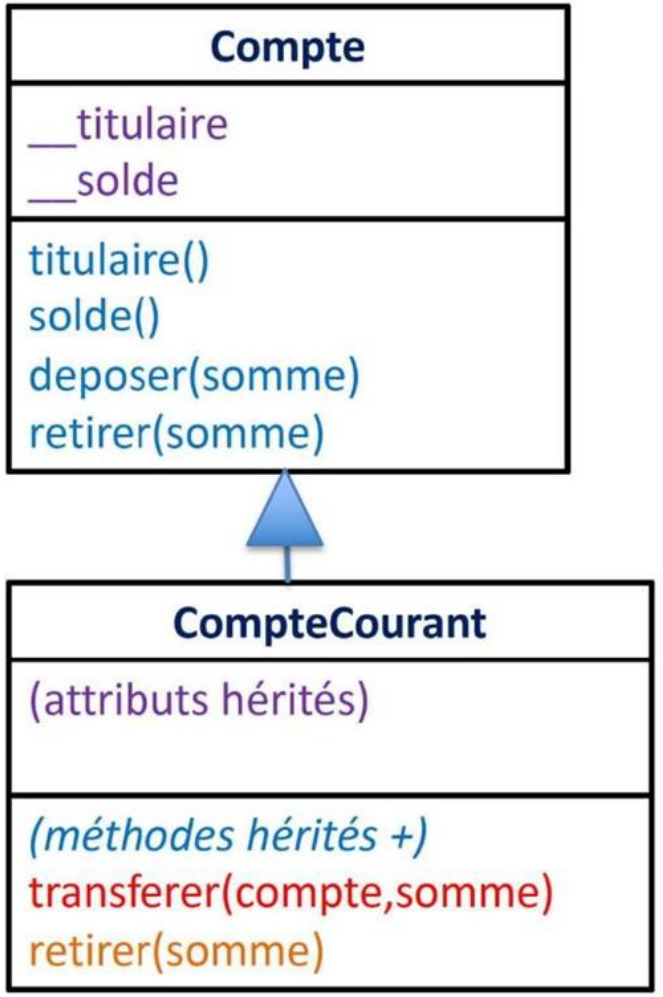
Attributs hérités

Méthodes héritées

Super()

Redéfinition

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



BUROTIX ()

Héritage

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

  def transferer(self,compte,somme) :
    …

  def retirer(self, somme):
    frais = __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

BUROTIX ()

Héritage

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

  def transferer(self,compte,somme) :
    …

  def retirer(self, somme):
    frais = __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
```

BUROTIX ()

Exo 13-52-03 : le compte en banque moderne, avec frais de retrait

- Des frais de 1 € par retrait ou transfert sont comptés.
- Class "CompteCourant"
  - Attribut
    - Frais = 1 €

- Scénario

```
kim_c = CompteCourant("Kim")
kim_c.deposer(1000)
glo_c = CompteCourant("Glo")
kim_c.transferer(glo_c, 150)
kim_c.retirer(200)
print(kim_c)
print(glo_c)
```

- Output

```
owner:Kim; balance:648
owner:Glo; balance:150
```

Héritage

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

  def transferer(self,compte,somme) :
    …

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

Pourquoi pas?

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

RecursionError: maximum recursion depth exceeded

BUROTIX ()

Appel à super()

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

  def transferer(self,compte,somme) :
    …

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

Meilleure solution

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

BUROTIX ()

super()

- `super()` permet de référer à une classe mère sans devoir la nommer explicitement.
- Souvent utiliser 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
```

Python `super()` function

- Dans la classe enfant, nous pouvons faire référence à la classe parent en utilisant la fonction `super()`.
- La fonction `super()` renvoie un objet temporaire de la classe parent qui nous permet d'appeler une méthode de classe parent à l'intérieur d'une méthode de classe enfant.

- Avantages
  - Le programmeur ne doit pas mentionner explicitement le nom de la classe parente pour accéder à ses méthodes.
  - Utilisation dans les héritages simples et multiples.
  - Réutilisation du code

Exo 13-52-05a : Python super() function

```
class Company:
    def company_name(self):
        return 'Google'

class Employee(Company):
    def info(self):
        Calling the superclass method using super()function
        c_name = super().company_name()
        print("Jessa works at", c_name)

Creating object of child class
emp = Employee()
emp.info()
```

OUTPUT

Jessa works at Google

= BUROTIX ()

Exo 13-52-05b : le compte en banque moderne, avec frais de retrait + super()

- Reprendre exo 13-03-03
- Class "CompteCourant"
  - Implanter `super()`

- Scénario (inchangé)

```
kim_c = Compte("Kim")
kim_c.deposer(1000)
glo_c = Compte("Glo")
kim_c.transferer(glo_c, 150)
kim_c.retirer(200)
print(kim_c)
print(glo_c)
```

- Output

```
owner:Kim; balance:648
owner:Glo; balance:150
```

Section 13-53 : Héritage varié …

Single inheritance

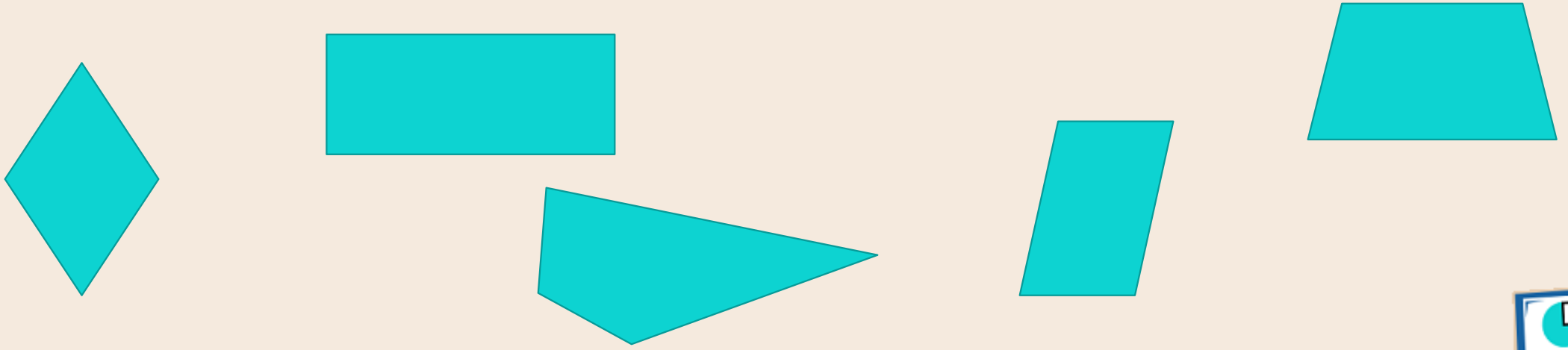
Multilevel inheritance

Hierarchical Inheritance

Hybrid Inheritance

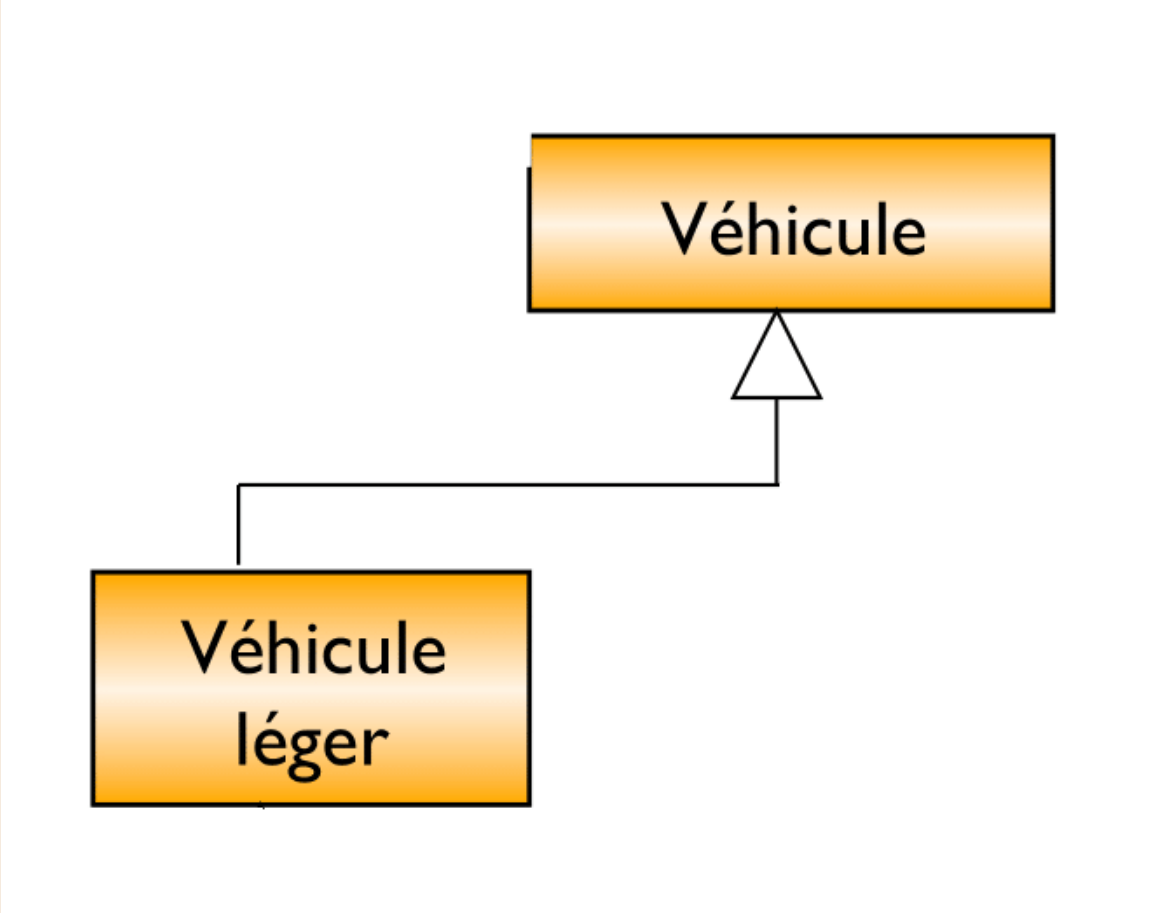
Exo 13-53-04 : les quadrilatères

- Comment classer ces quadrilatères ?
- Raisonnement :
  - Question : *forme 1* est-elle une *forme 2* ?
  - Si oui : alors *forme 1* hérite de *forme 2*



= BUROTIX ()

Héritage en Python : Single inheritance



- Une classe enfant hérite d'une classe monoparentale.

= BUROTIX ()

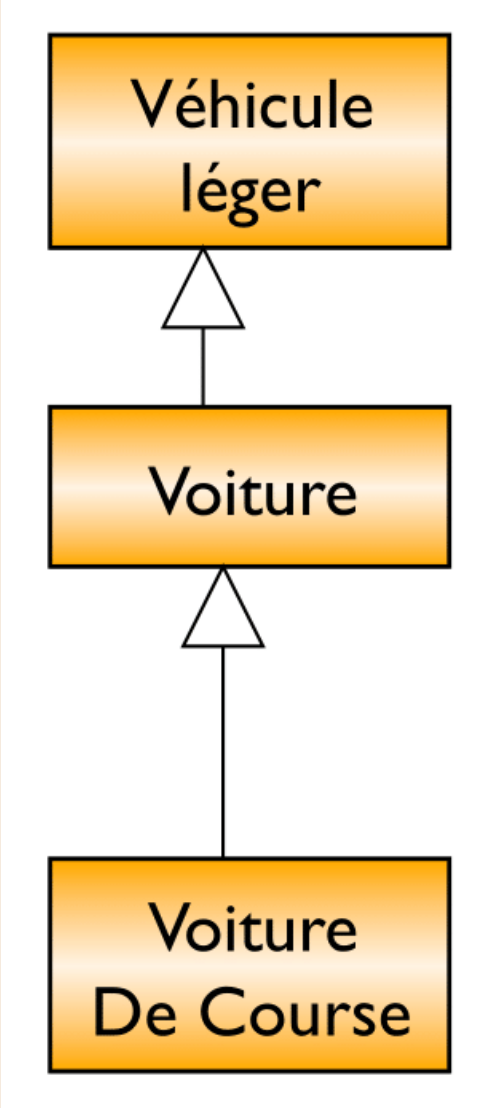
Exo 13-53-08a : single inheritance

```
Base class
class Vehicle:
    def vehicle_info(self):
        print('Vehicle')

Child class
class Car(Vehicle):
    def car_info(self):
        print('Car')
```

```
Create object of Car
car = Car()
access Vehicle's info
using car object
car.vehicle_info()
car.car_info()

OUTPUT
Vehicle class
Car class
```

= BUROTIX ()

Héritage en Python : Multilevel inheritance



- Une classe hérite d'une classe enfant.
- Supposons trois classes A, B, C.
  - A est la superclasse
  - B est la classe enfant de A
  - C est la classe enfant de B.
- "chaîne de classes" ou "héritage à plusieurs niveaux"

= BUROTIX ()

Exo 13-53-08b : multilevel inheritance

```
# Base class
class Vehicle:
    def vehicle_info(self):
        print('Vehicle')

# Child class
class Car(Vehicle):
    def car_info(self):
        print('Car')

# Child class
class SportsCar(Car):
    def sports_car_info(self):
        print('SportsCar')
```

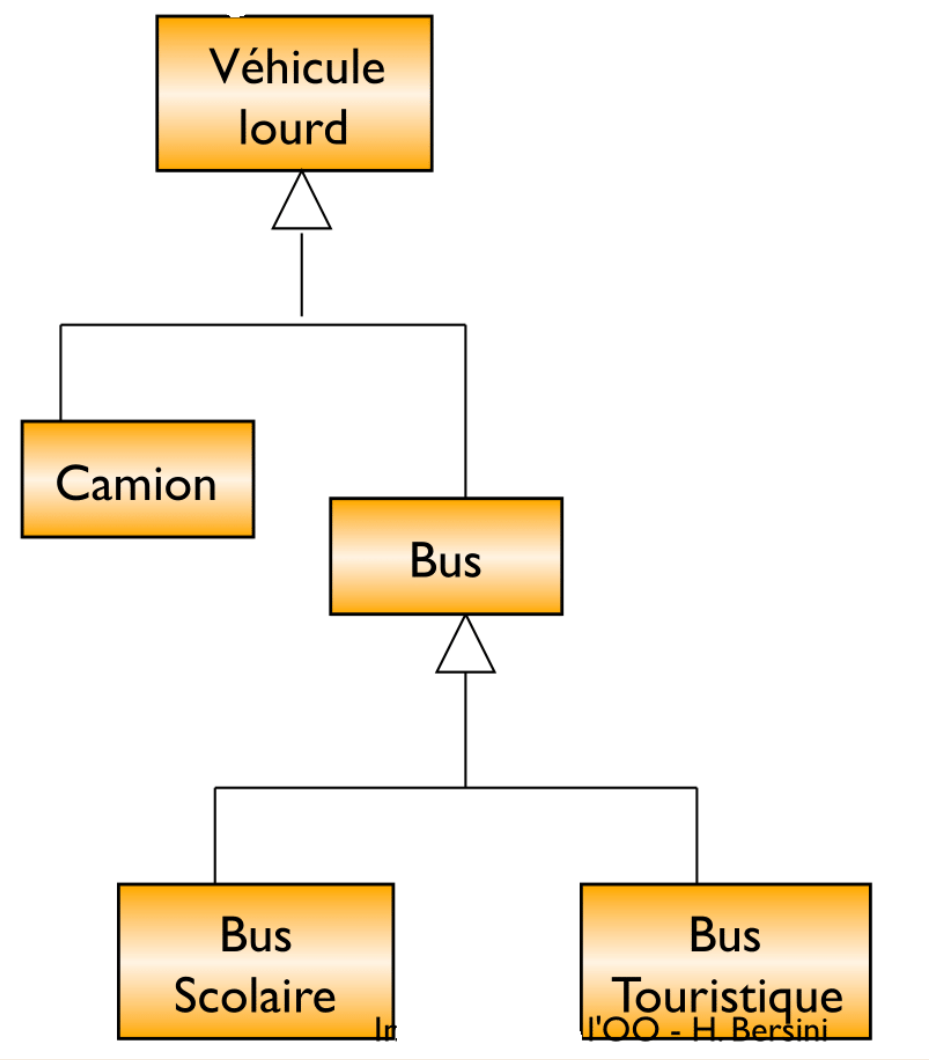
```
# Create object of SportsCar
s_car = SportsCar()
# access Vehicle's and Car info
using SportsCar object
s_car.vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

OUTPUT

Vehicle class
Car class
SportsCar class

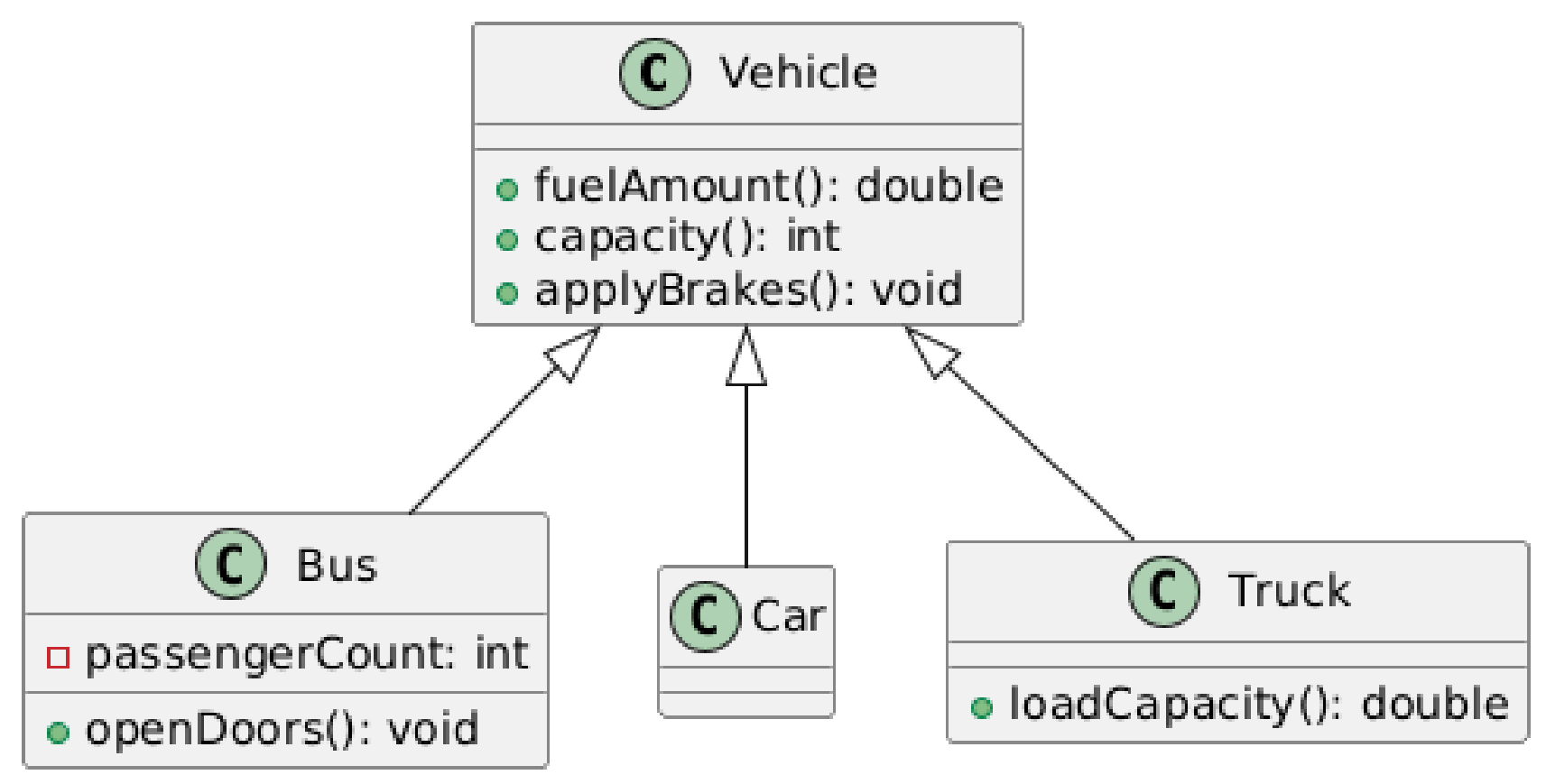
= BUROTIX ()

Héritage en Python : Hierarchical inheritance



- Plus d'une classe enfant est dérivée d'une seule classe parent.
- Il y a une classe parent et plusieurs classes enfants.

Exo 13-53-08c : hierarchical inheritance



= BUROTIX ()

Exo 13-53-08c : hierarchical inheritance

```
class Vehicle:
    def info(self):
        print("This is Vehicle")

class Car(Vehicle):
    def car_info(self, name):
        print("Car is:", name)

class Truck(Vehicle):
    def truck_info(self, name):
        print("Truck is:", name)
```

```
obj1 = Car()
obj1.info()
obj1.car_info('BMW')
obj2 = Truck()
obj2.info()
obj2.truck_info('Ford')
```

OUTPUT

```
This is Vehicle
Car name is: BMW
This is Vehicle
Truck name is: Ford
```

= BUROTIX ()

Attributs "protégés" vs public ou privé

- Attribut public
  - accessible depuis tous les objets, en dehors de la classe
    `frais_retrait = 0`
- Attribut privé
  - accessible seulement dans sa propre classe
    `__codedacces = "abcd"`
- Attribut protégé
  - accessible seulement dans sa classe et les classes enfants
    `_type = "Renaissance"`

- Sample: exo 13-03-09a
- Implémentation différente suivant le langage :
  - Python : attributs protégés émulés (non implantés)
  - Java, PHP : attributs protégés implantés
  - Les discussions perdurent entre les "pro" et les "con" des attributs protégés

Exo 13-53-09b : les comptes en banque modernes : c-courant, c-épargne

- Le retrait d'argent suit des règles différentes en fonction du type de compte
- Compte Renaissance
  - Seulement dépôt et retrait
  - Solde négatif interdit - refus émis quand le retrait rend le solde négatif
- Compte Courant
  - Solde négatif autorisé - avertissement émis quand le solde devient négatif
- Compte Epargne
  - Solde négatif interdit - refus émis quand le retrait rend le solde négatif
  - Intérêt créditeur de 1%

- Scénario

```
kim_cc = CompteCourant("Kim")
kim_cc.deposer(200)
kim_cc.retirer(500)
print(kim_cc)
scénario similaire pour Compte Renaissance
et Compte Epargne
```

- Output

```
Kim (CC) : solde de 200
Kim (CC) : solde de -302
Kim (CE) : solde de 200
Kim (CE) : solde insuffisant
Kim (CE) : solde de 200
```

= BUROTIX ()

Exo 13-53-11 : classes Vehicle et Bus

- Créer une classe parent `Vehicle`
  - Attributs d'objet `name` et `max_speed`
    - Passage par le constructeur
  - Print : *`Name: Tesla, Speed: 180`*
- Créez une classe enfant `Bus` qui héritera de tous les attributs et méthodes de la classe `Vehicle`.
  - Pas d'attribut ou de méthode spécifique à `Bus`
- Créez un objet `Bus` qui héritera de toutes les variables et méthodes de la classe `Vehicle`.
  - Print : *`Name: Volvo, Speed: 120`*

= BUROTIX ()

Exo 13-53-12 : classes Vehicle et Bus

- Partir de l'exo précédent
- Donnez à tous les véhicules une couleur blanche
  - Attribut de classe : `color`
  - Print : *Name: Tesla, color : white*
    Name: Volvo, color: white
- Donnez aux véhicules une capacité de places assises
  - `Vehicle` : 4 places assises
  - `Bus` : 50 places assises
  - Attribut de classe : `seating_capacity`
  - Print : *Name: Tesla, seats : 4*
    Name: Volvo, seats : 50

= BUROTIX ()

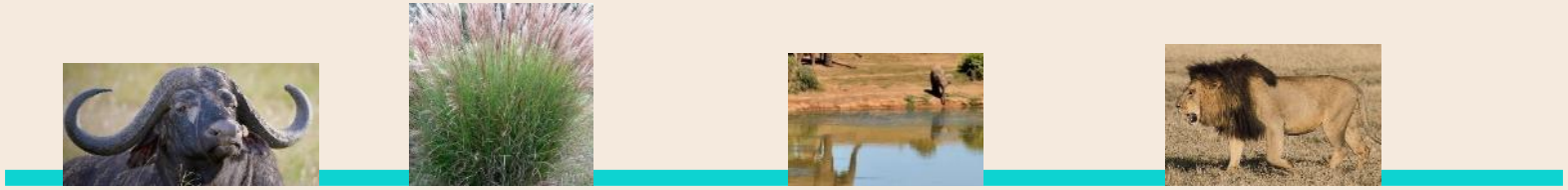
Exo 13-53-13 : classes Vehicle et Bus

- Partir de l'exo précédent
- Calculez le tarif de location de chaque véhicule :
  - tarif par défaut = nombre places assises * 50 €
  - tarif bus = nombre places assises * 50 € + majoration 10%
  - méthode `fare()` à définir dans les deux classes `Vehicle` et `Bus`
  - Print
```
Name: Tesla, Tarif : 200
Name: Volvo, Tarif : 2750.0
```

Exo 13-53-24 : Fauna & Resource

- Reprenez l'exo 13-35-23 avec les trois instances des classes `Water`, `Prey` et `Predator`.
- Etablissez l'héritage et supprimez tout le code redondant.
- Créez une classe `Fauna`
  - `Prey` et `Predator` en deviennent les classes-filles.
- Créez une classe `Resource`
  - `Water` en devient la classe-fille
- Créez la classe `Archaeplastida` sous la classe `Resource`.

- Spécifications supplémentaires :
  - La proie mange la plante
  - L'eau est disponible en quantité limitée.
  - La plante est disponible en quantité limitée.
  - Les animaux gagnent en énergie en mangeant ou en buvant.
  - Les animaux perdent leur énergie en marchant ou, encore plus, en courant.
  - Sans énergie, les animaux ne peuvent plus se déplacer.
  - Prévoir une méthode **`draw()`** qui affiche l'état de l'objet.



0 50 100 150

= BUROTIX ()

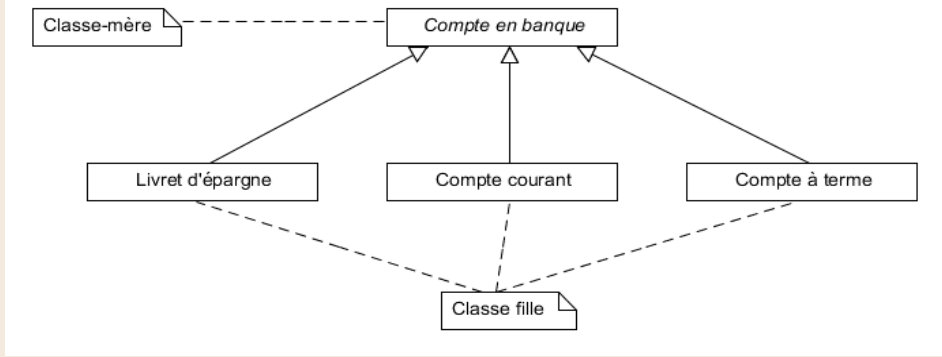
Fonctions Python intéressantes

- `type( <object> )`
- `isinstance( <object> )`
- `issubclass( <child>, <parent> )`

Section 13-55 :
Classe abstraite & Polymorphisme

= BUROTIX ()

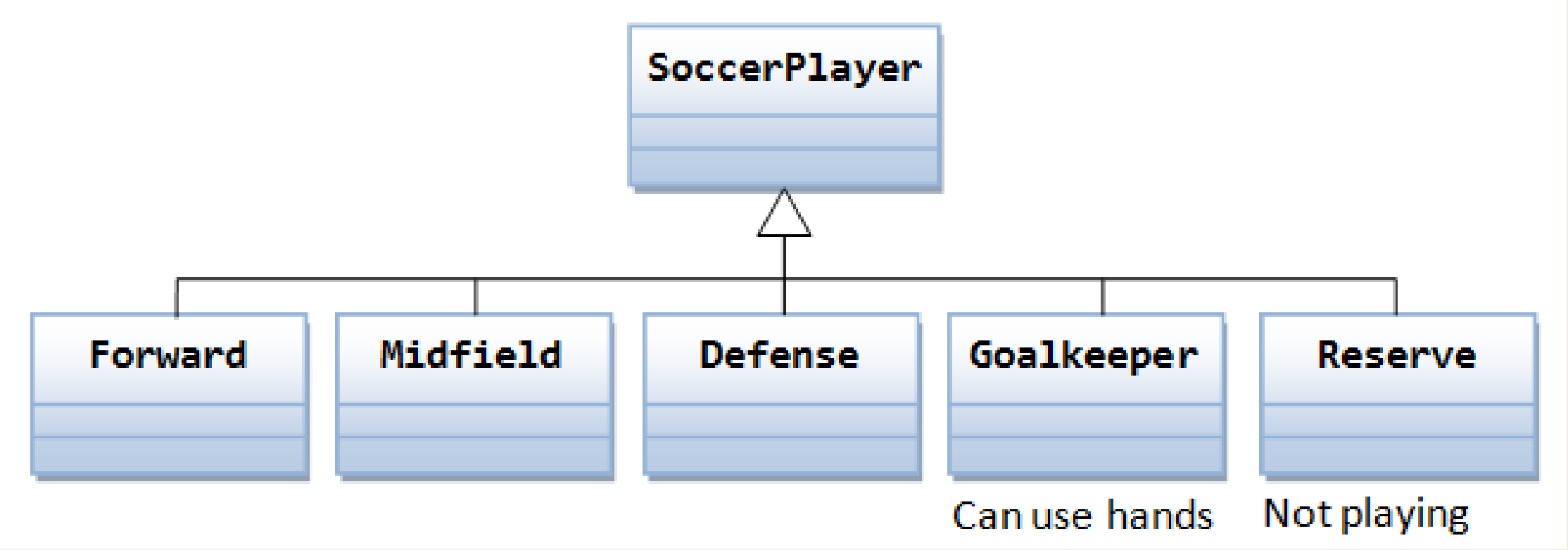
Classe abstraite



- Dans certains cas, la classe-mère ne peut pas être instanciée
  - La classe mère est dite "classe abstraite"
  - **Seules les classes-filles peuvent être instanciées.**
- Exemple
  - À notre époque, il n'est plus question d'ouvrir un compte en banque de type "Renaissance".
  - Pourtant, il faut le conserver car il est le père des comptes modernes. Ses attributs et méthodes restent utilisés.
  - Si vous voulez ouvrir un compte en banque, vous devez vous contenter d'un livret d'épargne, d'un compte courant ou d'un compte à terme.
  - Le compte "Renaissance" est décrété obsolète.

= BUROTIX ()

L'équipe de football (*soccer*)



Class abstraite en Python

- De base, pas de classe abstraite en Python.
- Utiliser le module `Abstract Base Classes (ABC)`
  - Architecture: `decorator`
  - Attribut abstrait : decorator `@abstractproperty`
  - Méthode abstraite : decorator `@abstractmethod`
- Sample: exo 13-55-02

Exo 13-55-05 : les comptes en banque modernes : c-courant, c-épargne

- Reprendre l'exo 13-33-09
- Le compte Renaissance est déclaré obsolète. Plus personne ne peut ouvrir un tel compte. Sa classe reste mère des comptes courant et épargne, mais devient abstraite.
- Adaptez le code en conséquence.
- A quelle(s) méthode(s) applique-t-on le decorator `@abstractmethod` ?

- Scénario

```
kim_cc = CompteCourant("Kim")
kim_cc.deposer(200)
kim_cc.retirer(500)
print(kim_cc)
scénario similaire pour Compte Epargne
scénario similaire pour Compte Renaissance => ERREUR !
```

= BUROTIX ()

Polymorphisme en POO

- Approche
  - Plusieurs classes
  - **Même** structure
  - **Mêmes** attributs
  - **Même** nom de méthodes
  - Mais définis **différemment** !
- Première implantation
  - Classes indépendantes
  - Sample: exo 13-05-07

= BUROTIX ()

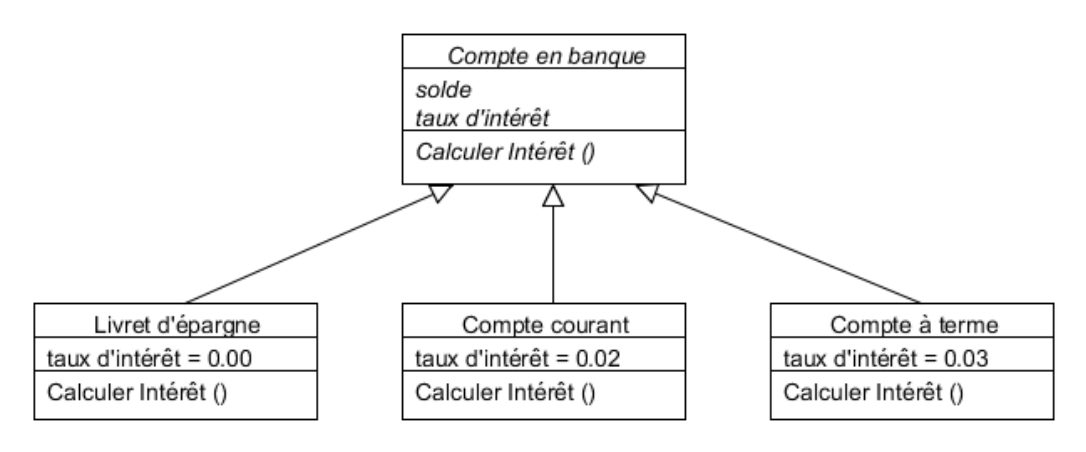
Polymorphisme : exo 13-55-07

```
class Cat:
    def __init__(…):
        self.name = name
        self.age = age
    def info(self):
        print(…)
    def make_sound(self):
        print("Meow")

class Dog:
    def __init__(…):
        self.name = name
        self.age = age
    def info(self):
        print(…)
    def make_sound(self):
        print("Bark")
```

```
# MAIN

animal_l = [
  Cat("Kitty", 2.5),
  Dog("Fluffy", 4)
]

for animal in animal_l:
    animal.info()
    animal.make_sound()
```

= BUROTIX ()

Polymorphisme



- Exemple : calcul de l'intérêt
  - sur **tous** les comptes bancaires
  - mais d'une façon **propre** à chaque type de compte (courant, épargne, à terme).
- Corollaire direct de l'héritage.

= BUROTIX ()

Polymorphisme en POO

- Approche
  - Plusieurs classes
  - **Même** structure
  - **Mêmes** attributs
  - **Même** nom de méthodes
  - Mais définis **différemment** !
- Première implantation
  - Classes indépendantes
- Deuxième implantation
  - Utiliser ensemble l'héritage et l'abstraction, ça vous dit?
  - Classes filles d'une classe abstraite
  - Sample : exo 13-05-08

Polymorphisme : exo 13-55-08

```
class Animal(ABC):
    def __init__(…):
        self.name = name
        self.age = age

    def info(self):
        print(…)

    @abstractmethod
    def make_sound(self):
        pass
```

```
class Cat(Animal):
    type = "cat"

    def make_sound(self):
        print("Meow")

class Dog(Animal): …

# MAIN
animal_l = [
  Cat("Kitty", 2.5),
  Dog("Fluffy", 4)
] …
```

= BUROTIX ()

Exo 13-55-09 : les comptes en banque modernes : c-courant, c-épargne

- Reprendre l'exo 13-35-09
- Développez une structure OO polymorphique.
- Méthodes abstraites
  - `.déposer()`
  - `.retirer()`
  - `.transférer()`

- Scénario

```
kong_cc = CompteCourant("Kong")
kong_cc.deposer(50)

kim_cc = CompteCourant("Kim")
kim_cc.deposer(200)
kim_cc.transferer(kong_cc, 100)
kim_cc.retirer(500)
print(kim_cc)  # -402

kim_ce = CompteEpargne("Kim")
kim_ce.deposer(200)
kim_ce.transferer(kong_cc, 100) # error
kim_ce.retirer(500) # error
print(kim_ce)  # 200

print(kong_cc)  # 150
```

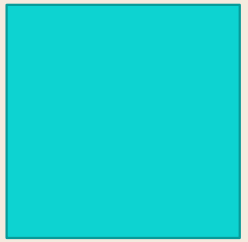
= BUROTIX ()

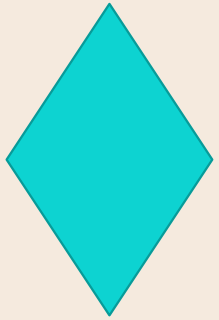
Section 13-57 : Héritage multiple

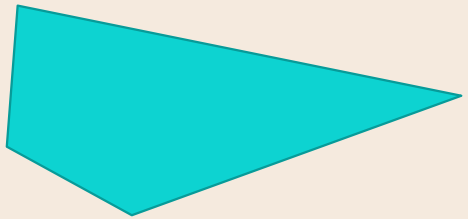
= BUROTIX ()

Exo 13-57-04 : les quadrilatères

Comment classer ces quadrilatères ?

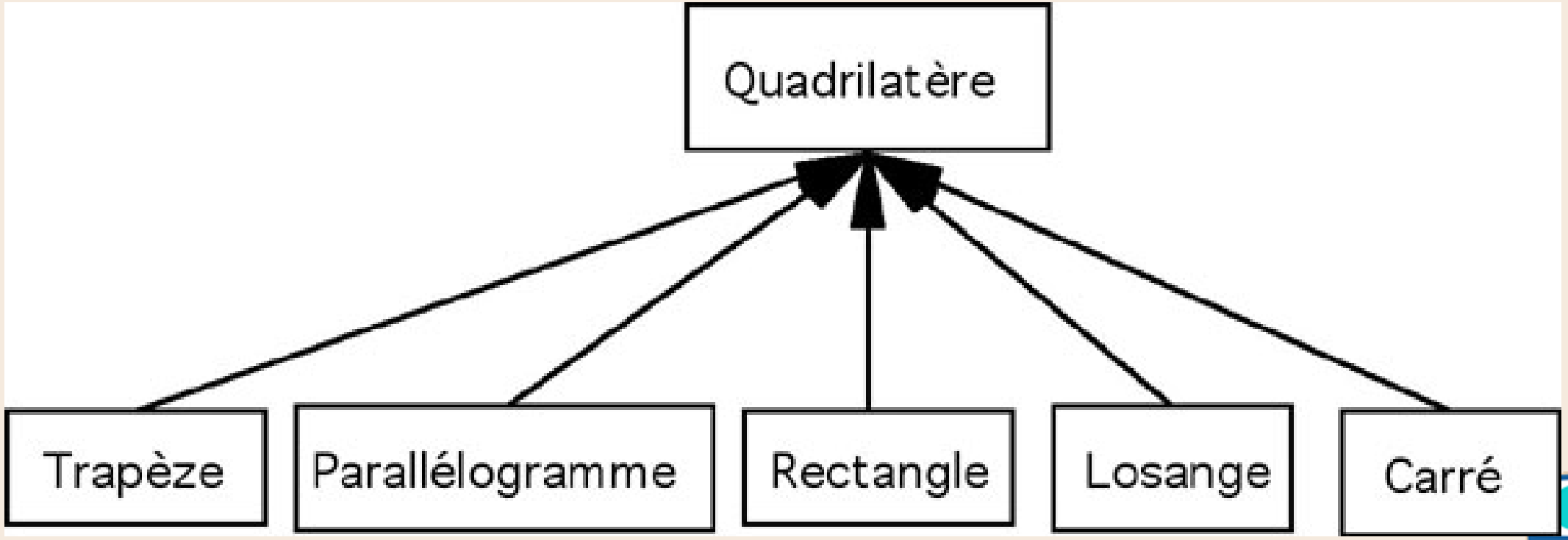






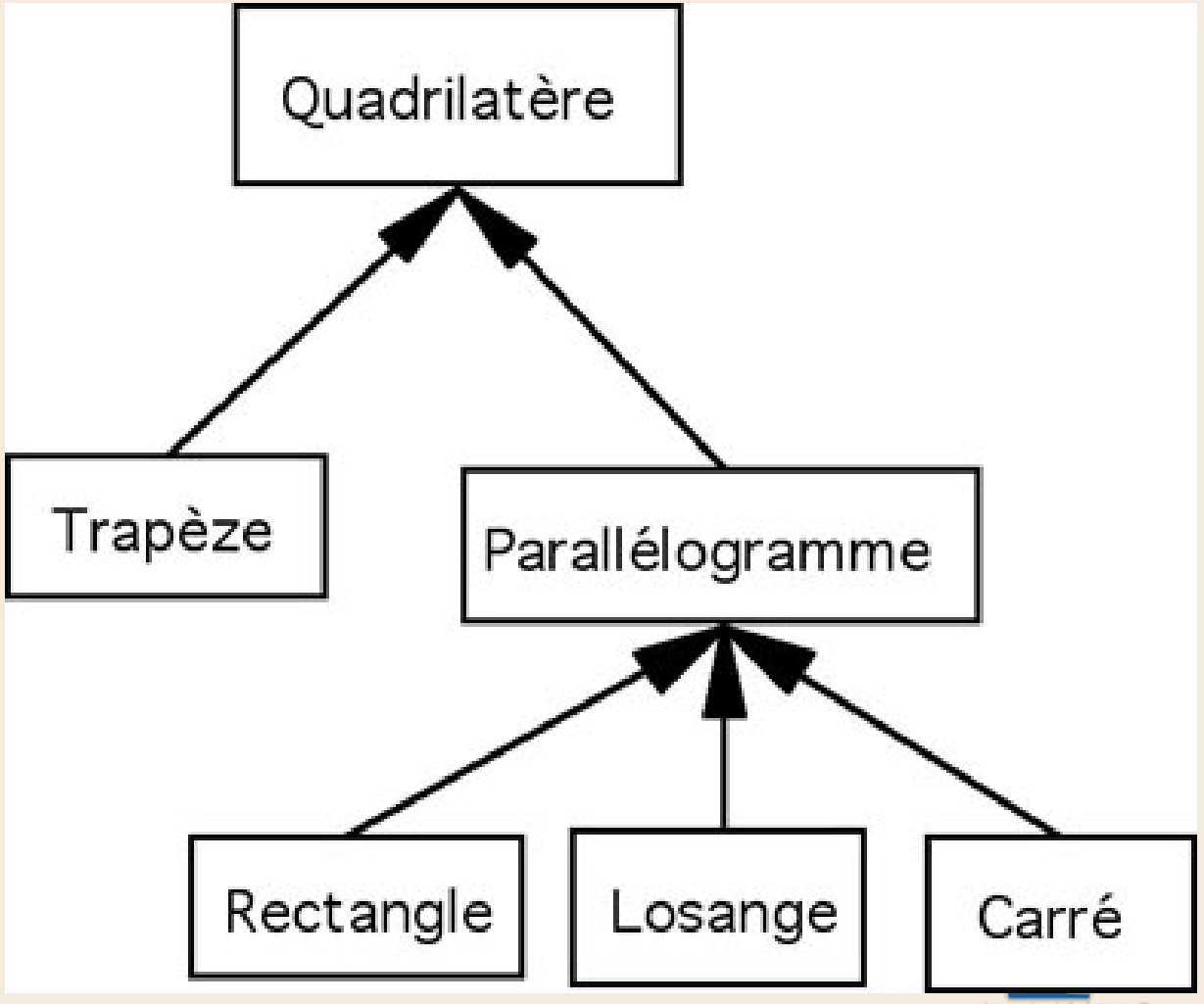
Exo 13-57-04 : les quadrilatères

Que pensez-vous de ce modèle de classes ?



Exo 13-57-04 : les quadrilatères

Ou de celui-ci ?

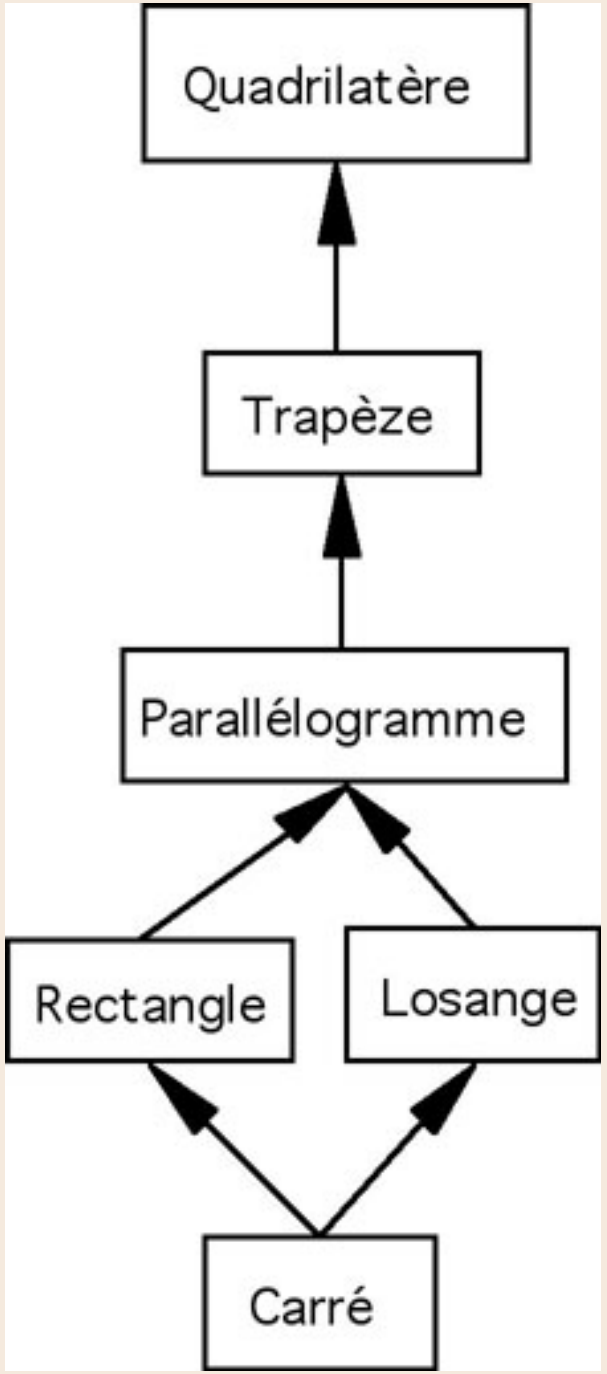


Exo 13-57-04 : les quadrilatères

Encore mieux ?

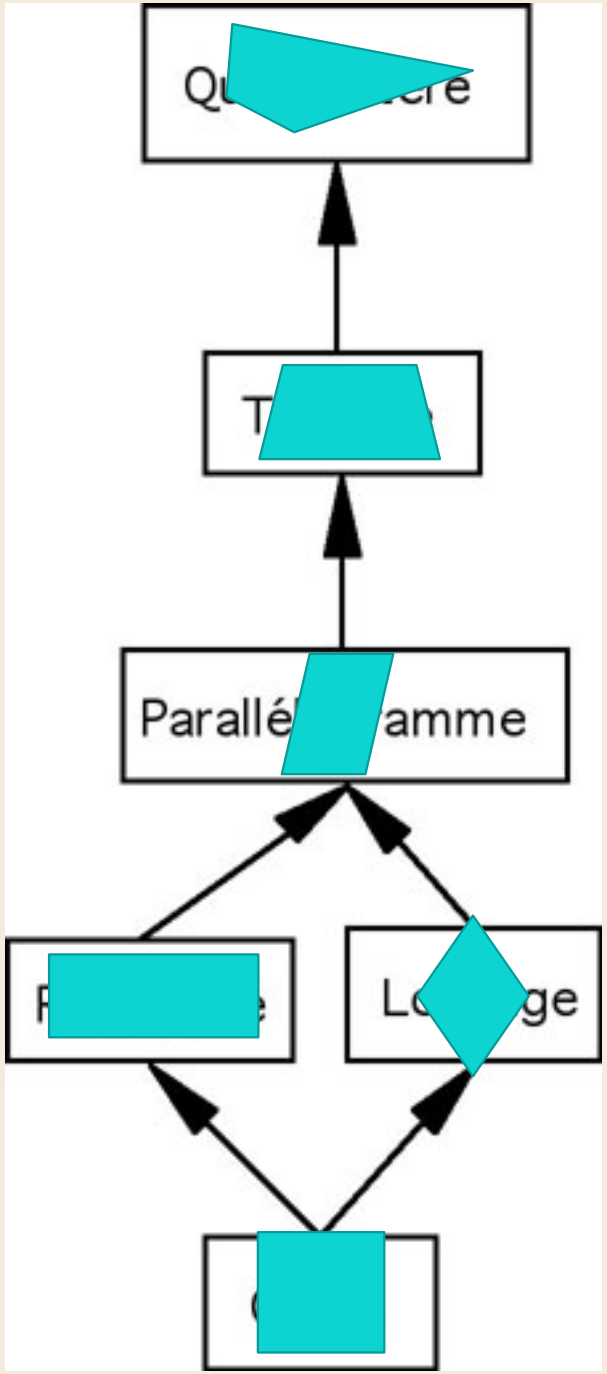
Réfléchissez à des assertions telles que :

- Le trapèze est-il un quadrilatère ?
  - OUI => héritage
- Le trapèze est-il un carré ?
  - NON => le trapèze n'hérite pas du carré
- Le carré est-il un trapèze ?
  - OUI => héritage



Exo 13-57-04 : les quadrilatères

Plus en détail



4 côtés

2 côtés parallèles

2 autres côtés parallèles

4 côtés égaux
(4 angles égaux)

4 angles droits

What else ?

Héritage multiple.

= BUROTIX ()

Exo 13-57-04 : les quadrilatères : en Python

```
class Quadrilatere:
    quatreCotes = True

class Trapeze(Quadrilatere):
    deuxCotesPar = True

class Parallelo(Trapeze):
    deuxAutresCotesPar = True
```

```
class Losange(Parallelo):
    quatreCotesEgaux = True

clas Rectangle(Parallelo):
    quatreAnglesDroits = True

class Carre(Losange, Rectangle):
    pass
```

= BUROTIX ()

Dangers de l'héritage multiple

- Gare aux attributs et méthodes définis dans les deux classes parentes.
- Exo 13-57-06
  - Héritage multiple de `Carre`, enfant de `Rectangle` et `Losange`
  - Attribut name défini dans `Rectangle` et `Losange`
  - Le `dump` affiche la valeur de l'attribut name de `Losange`, pas de `Rectangle`. Pourquoi ?

= BUROTIX ()

Chapitre 13-9 : Révisions

Rien que des exos ;-)

Exo 13-91 : HTML Page Builder

- Pour chacun des exercices suivants
  - Ecrivez les classes
    - correspondant au code Python donné
    - produisant le code HTML donné
  - En utilisant tous les mécanismes de l'héritage classique
    - pas de polymorphisme
    - pas d'héritage multiple

Exo 13-91-02

```
html_o = HtmlPageBuilder()

div_o = HtmlDiv()
div_o.contents = "Hello World"

html_o.append(div_o)
print(html_o)

# dév. intermédiaire
print(div_o)
```

```
<html>
<head></head>
<body>
      <div>
            Hello World
      </div>
</body>
</html>

# dév. intermédiaire
<div>
      Hello World
</div>
```

= BUROTIX ()

Exo 13-91-04

```
html_o = HtmlPageBuilder()
div1_o = HtmlDiv("Hello World")
html_o.append(div1_o)
div2_o = HtmlDiv("Hello You")
html_o.append(div2_o)
pre1_o = HtmlPre("I'am "pre".")
html_o.append(pre1_o)
print(html_o)
```

```
<html>
<head></head>
<body>
        <div>Hello World</div>
        <div>Hello You</div>
        <pre>I am "pre".</pre>
</body>
</html>
```

Exo 13-91-06

```
builder = HtmlPageBuilder()
div0 = HtmlDiv("Hi ! ")
builder.append(div0)
div1 = HtmlDiv("Hello World")
div0.append(div1)
div2 = HtmlDiv("Hello You")
div0.append(div2)
pre1 = HtmlPre("""I'm "pre".""")
div0.append(pre1)
print(builder)
```

```
<html>
<head></head>
<body>
  <div>
    Hi !
    <div>Hello World</div>
    <div>Hello You</div>
    <pre>I am "pre".</pre>
  </div>
</body>
</html>
```

Exo 13-91-08

```
builder = HtmlPageBuilder()
section = HtmlSection()
builder.append(section)
div1 = HtmlDiv("Hello World")
section.append(div1)
div2 = HtmlDiv("Hello You")
section.append(div2)
pre1 = HtmlPre("""I'm "pre".""")
section.append(pre1)
print(builder)
```

```
<html>
<head></head>
<body>
  <section>
      <div>Hello World</div>
    <div>Hello You</div>
    <pre>I am "pre".</pre>
  </section>
</body>
</html>
```

Exo 13-92 : droits d'accès

- Une application web propose différents niveaux d'accès pour ses utilisateurs.
- Chaque rôle possède des droits spécifiques, et certains rôles étendent les droits d'autres rôles.
- Déterminez la hiérarchie de classes.
  - diagramme UML
  - nom de classe
  - méthodes
  - remarque : certains rôles peuvent avoir plusieurs parents

Exo 13-92 : droits d'accès : les rôles

Visiteur

- Consulter les contenus publics de l'application.
- Ni publier ni interagir avec les contenus.

Membre

- Consulter les contenus publics de l'application.
- Publier des contenus (articles, images, etc.).
- Commenter les contenus publiés par d'autres utilisateurs.

= BUROTIX ()

Exo 13-92 : droits d'accès : les rôles

Administrateur

- Consulter les contenus.
- Publier des contenus.
- Commenter les contenus.
- Supprimer des contenus.
- Avertir les utilisateurs.
- Bannir des utilisateurs de l'application.
- Modifier les rôles des autres utilisateurs
  - ex: promouvoir un Membre en Modérateur
- Modifier les contenus publiés par les Contributeurs.
- Ne pas commenter les contenus.

= BUROTIX ()

Exo 13-92 : droits d'accès : les rôles

Modérateur

- Consulter les contenus.
- Publier des contenus.
- Commenter les contenus.
- Supprimer des contenus inappropriés.
- Avertir les utilisateurs en cas de manquement aux règles.

Éditeur

- Consulter les contenus.
- Publier des contenus.
- Modifier les contenus publiés par d'autres Contributeurs.
- Ne pas commenter les contenus.

= BUROTIX ()

Exo 13-92 : droits d'accès : les rôles

Contributeur

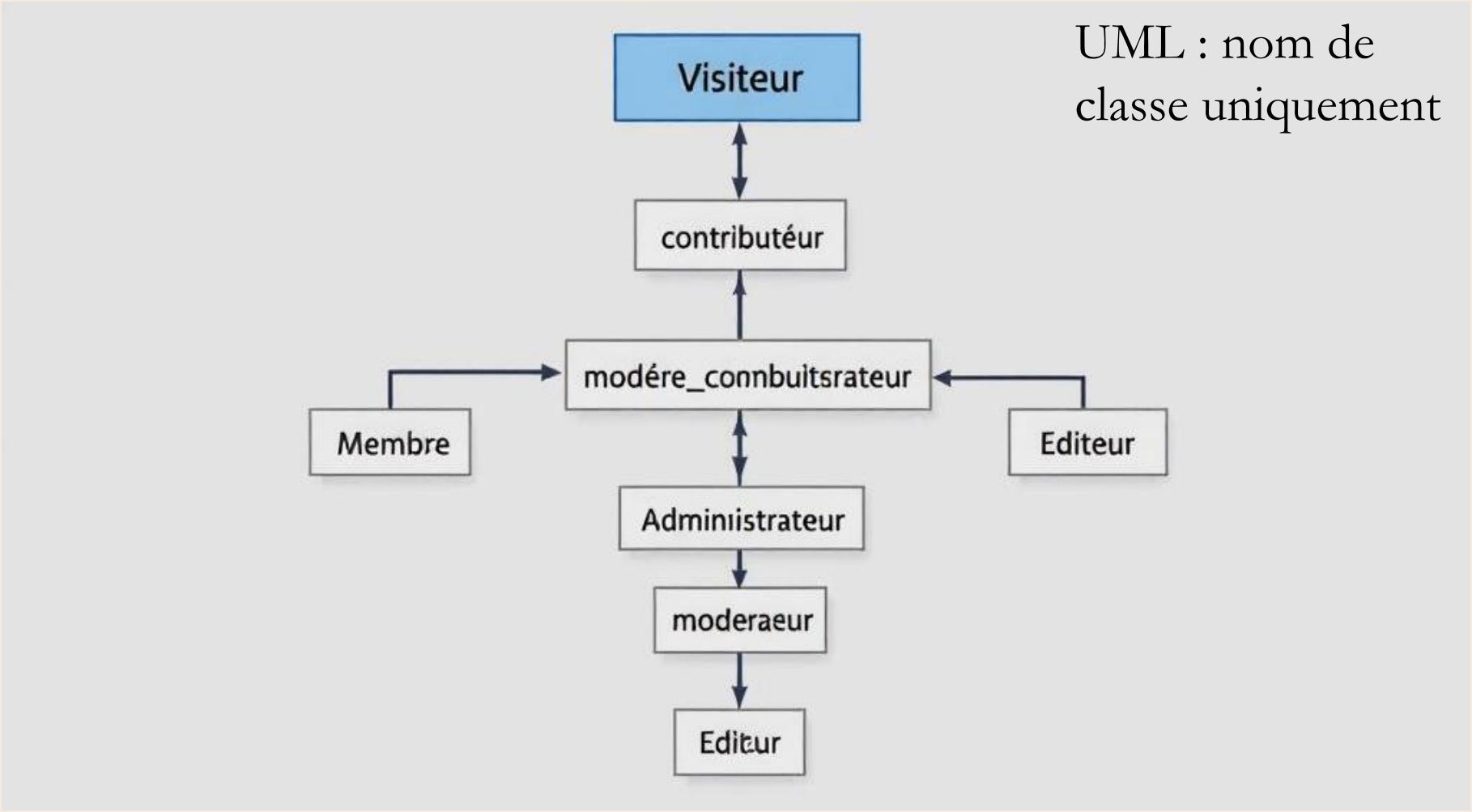
- Consulter les contenus.
- Publier des contenus.
- Ne pas commenter les contenus.

Analyste

- Consulter les statistiques
- Ne pas publier de contenu.

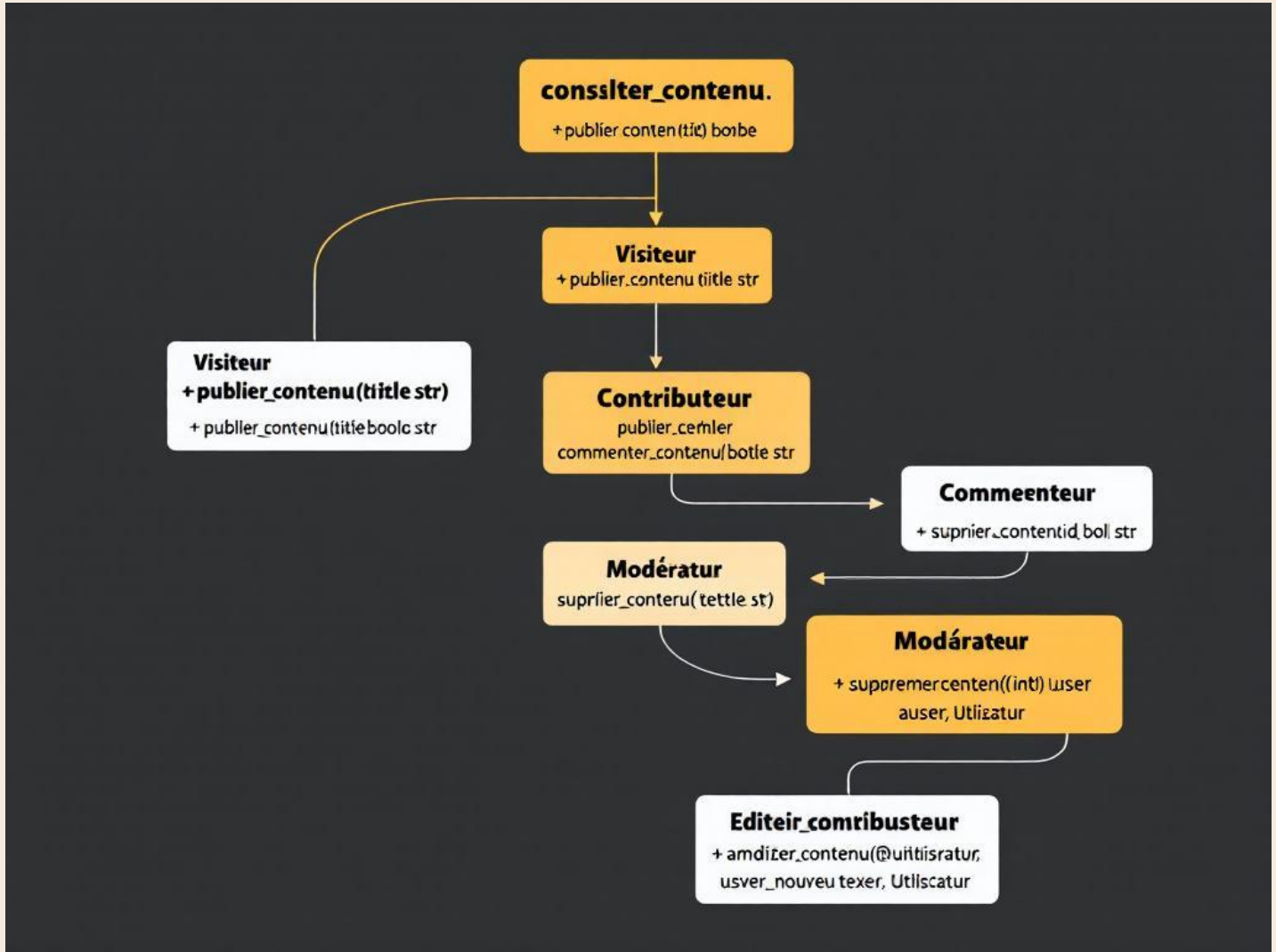
= BUROTIX ()

Exo 13-92 : esquisse de réponse (incorrecte)



UML : nom de classe uniquement

Exo 13-92 : esquisse de réponse (incorrecte)



UML :
nom de classe +
méthodes

= BUROTIX ()