Programmation Orientée Objet en Python

Héritage et Abstraction

T. Dietenbeck
thomas.dietenbeck@sorbonne-universite.fr

SCIENCES SORBONNE UNIVERSITÉ

Plan

SCIENCES SORBONNE UNIVERSITÉ

Plan

SCIENCES SORBONNE UNIVERSITÉ

Rappels : Principes Orientés Objets

Principe 1 : Encapsulation

- Rapprocher données (attributs) et traitements (méthodes)
- Protéger les informations (privées vs publiques)

Principe 2 : Agrégation/Association

- Agrégation : Classe A Possède Classe B
- Association : Classe A Utilise Classe B

Principe 3 : Héritage

- Classe B EstUn Classe A

SCIENCES SORBONNE UNIVERSITÉ

Exercice

Exercice

Les relations suivantes sont-elles de type **Aggrégation**, **Association** ou **Héritage** ?

- Cercle et Ellipse
- Salle de Bains et Baignoire
- Piano et JoueurPiano
- Véhicule et Personne
- Voiture et Propriétaire
- Entier et Réel
- Personne, Enseignant et Etudiant

SCIENCES SORBONNE UNIVERSITÉ

Exemple de départ

Les véhicules

On veut représenter et traiter des véhicules

- 2 types de véhicules : Véhicule et Voiture
- Attributs :
  - Véhicules : Marque, Année de fabrication
  - Voiture : Marque, Année de fabrication, Puissance (chevaux fiscaux)
- Méthodes :
  - Calculer l'âge
  - Donner le type du véhicule (véhicule ou voiture ?)
  - Afficher toutes informations du véhicule
    - Véhicule : Type, Marque, Année
    - Voiture : Type, Marque, Année, Puissance
  - Voiture : calculer une taxe

Vehicule
-annee: integer -marque: String
+<<create>> Vehicule(a:integer,m:String) +age(): integer +who(): String +toString(): String

Voiture
-annee: integer -marque: String -puissance: integer
+<<create>> Voiture(a:integer,m:String,p:integer) +age(): integer +who(): String +toString(): String +taxe(): real

SCIENCES SORBONNE UNIVERSITÉ

Exemple de départ

Solution classique (**sans** héritage)

```
from datetime import datetime
class Vehicule :
  def __init__( self, marque, annee ) :
    self.marque = marque
    self.annee = annee

  def who( self ) :
    return "Vehicule"

  def __str__( self ) :
    res = self.who( ) + " - marque " + self.marque
    res += " construit en " + str( self.annee )
    return res

  def age( self ) :
    now = datetime.now()
    return now.year - self.annee
```

SCIENCES SORBONNE UNIVERSITÉ

Exemple de départ

Solution classique (**sans** héritage)

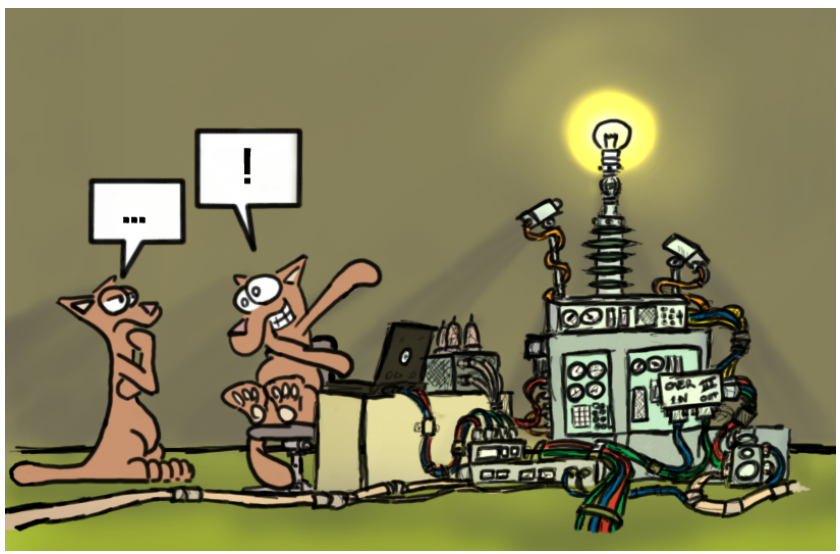
```
from datetime import datetime
class Voiture :
  def __init__( self, marque, annee, puissance ) :
    self.marque = marque
    self.annee = annee
    self.puissance = puissance

  def who( self ) :
    return "Voiture"

  def __str__( self ) :
    res = self.who( ) + " - marque " + self.marque
    res += " construit en " + str( self.annee )
    res += " de puissance " + str( self.puissance ) + " CV"
    return res

  def age( self ) :
    now = datetime.now()
    return now.year - self.annee

  def taxe( self ) :
    return self.puissance * 10 + 50.
```

SCIENCES SORBONNE UNIVERSITÉ

Exemple de départ

Limitations

- Solution inefficace
  - Duplication d'attibuts (ex : `annee`, `marque`)
  - Réécriture de méthodes (ex : `age`) : Si on veut changer une fonctionnalité, on doit parcourir **toutes** les classes l'implémentant pour y faire **les mêmes** changements
  - Gros overlap entre classes (11 lignes communes entre Vehicule (14 lignes) et Voiture (18 lignes))
- Problème pénalisant pour les applications réelles
  - Implémentation des classes `Moto`, `Avion`, `Bateau`, ...?
- Pas de hierarchie dans les classes : `Voiture` EstUn `Vehicule`



SCIENCES SORBONNE UNIVERSITÉ

Plan

SCIENCES SORBONNE UNIVERSITÉ

Définition

Définition

- Héritage = principe permettant de créer une nouvelle classe à partir d'une classe existante
- Relation unidirectionnelle la plus forte en POO. Elle traduit : B EstUn A
- La classe A est appelée "classe mère", B est la "classe fille"

Intérêts

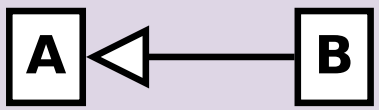
- Possibilité de créer une hiérarchie complète d'objets héritant les uns des autres.
- Construction : partir d'un objet relativement simple pour aller à des objets plus complexes ou plus spécialisés.

SCIENCES SORBONNE UNIVERSITÉ

Définition

Représentation UML

Relation d'héritage : 



Syntaxe Python

On donne la classe mère entre parenthèses

```
class B ( A ) :
```

SCIENCES SORBONNE UNIVERSITÉ

Propriétés

Réutilisation de ce qui est déjà réalisé

- Une classe B qui hérite d'une classe A possède tous les attributs et méthodes définis dans la classe A
- Si une méthode a la même fonctionnalité dans la classe mère et la classe dérivée, elle n'a pas besoin d'être réécrite.
- Si elle a une fonctionnalité différente, elle doit être re-déclarée et réécrite. On dit qu'elle **redéfinit** celle de son ancêtre.
  La déclaration de la méthode (paramètres et type de retour) ne doit pas changer dans la classe fille !

Spécialisation

- On peut ajouter des attributs et/ou méthodes dans la classe B pour traduire les nouvelles fonctionnalités.

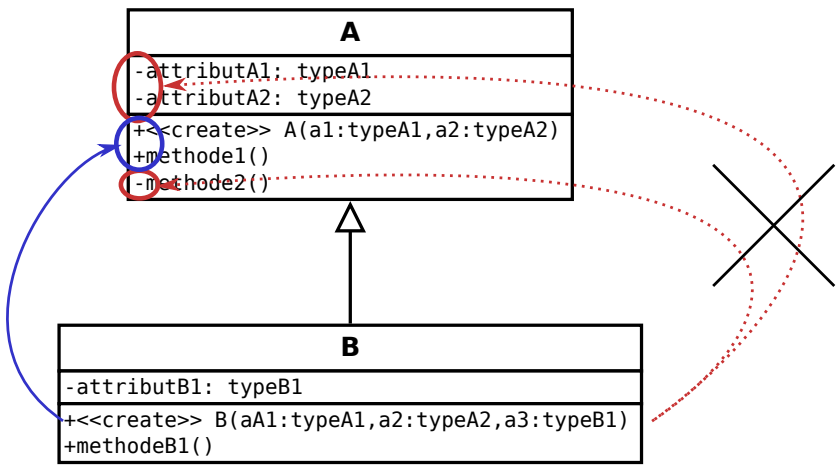
Propriétés

Visibilité et accès aux champs

- La visibilité des attributs et des méthodes de la classe mère (A) est appliquée à la classe fille (B)
  - La classe B a donc directement accès aux champs publics de la classe A
  - La classe B n'a pas accès aux champs privés de la classe A

Exemple

- B accède directement au constructeur de A et à `methode1` (car public)
- B n'a pas accès aux attributs et à `methode2` (car privé)



SCIENCES SORBONNE UNIVERSITÉ

Appel à la méthode ancêtre : `super`

Intérêt

- Pouvoir appeler une méthode de la classe ancêtre
- Dans le corps de la méthode de la classe dérivée
  - Appeler la méthode de la classe ancêtre pour traiter les attributs hérités
  - Traiter les nouveaux attributs (spécialisation)

Syntaxe

- Appel du constructeur de la classe ancêtre :

```
class B( A ) :
  def __init__( self, ... ) :
    super( ).__init__( ... )    # Appel du constructeur de A
    ...                 # Autres instructions du constructeur de B
```

- Appel d'une méthode de la classe ancêtre (redéfinie dans la classe dérivée) :

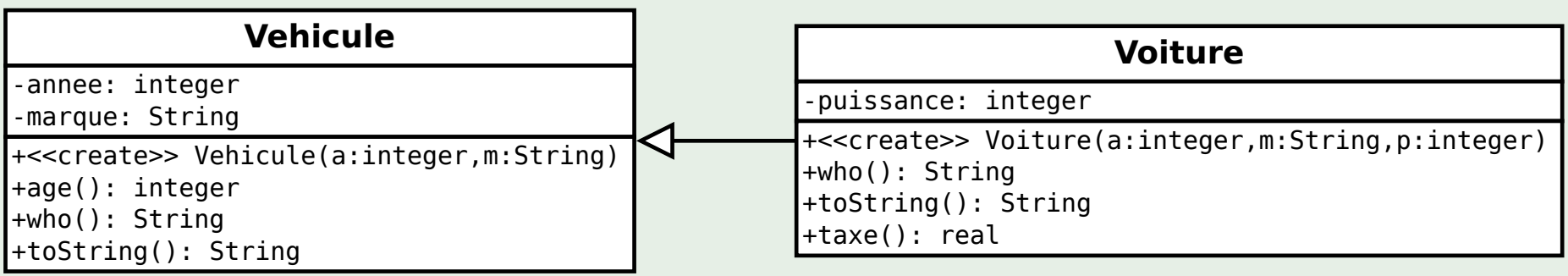
```
def nomMethode( self, ... ) :
  super( ).nomMethode(...)      # Appel de nomMethode de A
  ...                         # Specialisation pour la classe B
```

Exemple de départ

Analyse

- `Voiture` hérite de `Vehicule`
  - Inchangé : `marque`, `annee`, `age()`
  - Modifié : `who()`, `toString()` ⇒ à adapter selon le type
  - Nouveau : `puissance`, `taxe()`

Représentation UML



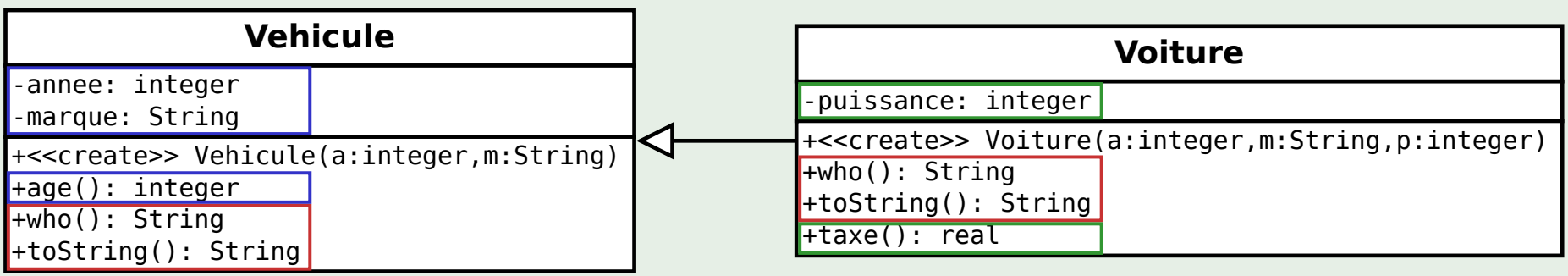
SCIENCES SORBONNE UNIVERSITÉ

Exemple de départ

Analyse

- `Voiture` hérite de `Vehicule`
  - Inchangé : `marque, annee, age()`
  - Modifié : `who(), toString()` ⇒ à adapter selon le type
  - Nouveau : `puissance, taxe()`

Représentation UML



SCIENCES SORBONNE UNIVERSITÉ

Exemple de départ

Solution avec héritage

```
from datetime import datetime
class Vehicule :
  def __init__( self, marque, annee ) :
    self.marque = marque
    self.annee = annee

  def who( self ) :
    return "Vehicule"

  def __str__( self ) :
    res = self.who( ) + " - marque " + self.marque
    res += " construit en " + str(self.annee)
    return res

  def age( self ) :
    now = datetime.now()
    return now.year - self.annee
```

SCIENCES SORBONNE UNIVERSITÉ

Exemple de départ

Solution **avec** héritage

```
class Voiture (Vehicule):
  def __init__( self, marque, annee, puissance ) :
    super( ).__init__( marque, annee )
    self.puissance = puissance

  def who( self ) :
    return "Voiture"

  def __str__( self ) :
    res = super( ).__str__( )
    res += " de puissance " + str( self.puissance ) + " CV"
    return res

  def taxe( self ) :
    return self.puissance * 10 + 50.
```

Exemple

Quelques erreurs

```
class Voiture ( Vehicule ) :
  def __init__( self, marque, annee, puissance ) :
    # Oubli de l'appel du constructeur Vehicule
    self.puissance = puissance

  # Mauvaise redefinition de la methode who
  def who( a ) :  # Pas les memes parametres
    return "Voiture"

  def taxe( self )
    # Acces a un attribut prive de Vehicule pour lequel le getter
        ↪ n'est pas defini
    return self.puissance * 10. + (self.__annee - 1900)


class Camion :  # Oubli de l'heritage
  def __init__( self, marque, annee, nbRoues ) :
    # Appel du constructeur Vehicule (inconnu)
    super( )__init__( marque, annee )
```

SCIENCES SORBONNE UNIVERSITÉ

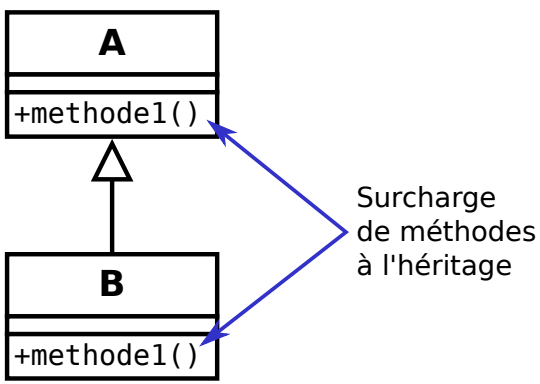
Plan

SCIENCES SORBONNE UNIVERSITÉ

Polymorphisme : Définition

Polymorphisme : qui peut prendre plusieurs formes

- Concept relatif à la surcharge ou à l'utilisation des méthodes de classes d'une même hiérarchie
  - La méthode appliquée à un objet est déterminée à l'exécution par le type de l'objet
    Ceci est automatique en Python, mais ce n'est pas le cas dans tous les langages !
- Extension des possibilités du mécanisme d'héritage :
  Une classe fille est du **même type** que sa classe mère



Surcharge de `methode1`

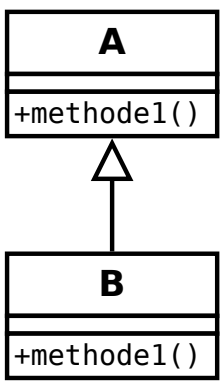
- Pour un objet de type B, `methode1` de la classe B remplace l'implémentation de la classe A

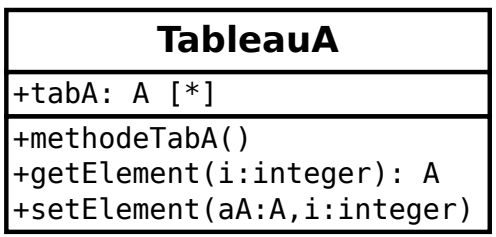
SCIENCES SORBONNE UNIVERSITÉ

Polymorphisme : Définition

Polymorphisme : qui peut prendre plusieurs formes

- Concept relatif à la surcharge ou à l'utilisation des méthodes de classes d'une même hiérarchie
  - La méthode appliquée à un objet est déterminée à l'exécution par le type de l'objet
    Ceci est automatique en Python, mais ce n'est pas le cas dans tous les langages !
- Extension des possibilités du mécanisme d'héritage :
  Une classe fille est du **même type** que sa classe mère





Compatibilité de types

Avec un objet de type `TableauA`, on peut gérer de la même manière des objets de classe A **ou** B

SCIENCES SORBONNE UNIVERSITÉ

Compatibilité de types : Définition

Conséquences

Si la classe B hérite de la classe A (qui hérite éventuellement de AA, *etc.*) :

- **Transitivité :** Toute méthode m de A peut-être appelée depuis une instance de B
  Ex : utilisation des méthodes de `Vehicule` à partir d'une `Voiture`
- **Subsomption :** Dans toute expression "qui attend" un A, je peux "placer" un B à la place
  Ex : ajout d'une `Voiture` dans un tableau de `Vehicule`

La réciproque est fausse

On ne peut pas invoquer une méthode spécifique de B depuis une instance de A

- Ex : On ne peut pas utiliser la méthode spécifique `taxe()` de `Voiture` sur un `Vehicule` car on ne connaît pas la puissance d'un `Vehicule`

SCIENCES SORBONNE UNIVERSITÉ

Compatibilité de types : Définition

Conséquences

Si la classe B hérite de la classe A (qui hérite éventuellement de AA, *etc.*) :

- **Transitivité :** Toute méthode m de A peut-être appelée depuis une instance de B
  Ex : utilisation des méthodes de `Vehicule` à partir d'une `Voiture`
- **Subsomption :** Dans toute expression "qui attend" un A, je peux "placer" un B à la place
  Ex : ajout d'une `Voiture` dans un tableau de `Vehicule`

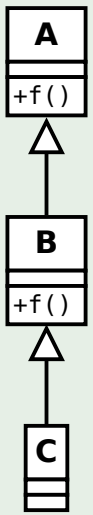
Intérêts

- Unification des traitements
- Une méthode réalisant une fonctionnalité donnée a le même nom et s'adapte en fonction du type de l'objet à traiter
- Si l'on ne connaît pas à l'avance le type d'un objet
  - on lui donne le **type le plus général** (ici `Vehicule`)
  - on choisit **à l'exécution** le type utilisé parmi les type dérivés

Exemples

Polymorphisme

```
class A :
  def f( self ) :
    print( "A.f" )

class B (A) :
  def f( self ) :
    print( "B.f" )

class C (B) :   # Classe sans
  pass          # definition
```




```
vA = A( );        vA.f( )
vB = B( );        vB.f( )
vC = C( );        vC.f( )
```

affiche :

```
A.f
B.f
B.f
```

Exemples

Polymorphisme

```
tab = [ ]     # Tableau vide
tab.append( Voiture( 2014, "Renault", 5 ) )
tab.append( Vehicule( 2005, "Rockrider" ) )
tab.append( Voiture( 2012, "Peugeot", 3 ) )
tab.append( Vehicule( 2015, "Airbus" ) )
tab.append( Vehicule( 2010, "MAN" ) )
for v in tab :
  print( v )
```

affiche :

```
Voiture - marque Renault construit en 2014 de puissance 5 CV
Vehicule - marque Rockrider construit en 2005
Voiture - marque Peugeot construit en 2012 de puissance 3 CV
Vehicule - marque Airbus construit en 2015
Vehicule - marque MAN construit en 2010
```

Adaptation au type de l'objet

La programmation sans héritage impliquerait un test du type de l'objet, 2 tableaux et 2 méthodes

SCIENCES SORBONNE UNIVERSITÉ

Limitations du polymorphisme

Ajout de méthodes

- Si les sous-classes ajoutent des méthodes, alors il est plus difficile d'exploiter le polymorphisme. On peut avoir besoin de vérifier les types avec `isinstance`

```
t = 0
if isinstance( v1, Voiture ) :  # Si on a une Voiture,
  t = v1.taxe( )                # on peut calculer la taxe
  # Sinon on utilise la valeur (par defaut) 0
```

SCIENCES SORBONNE UNIVERSITÉ

Limitations du polymorphisme

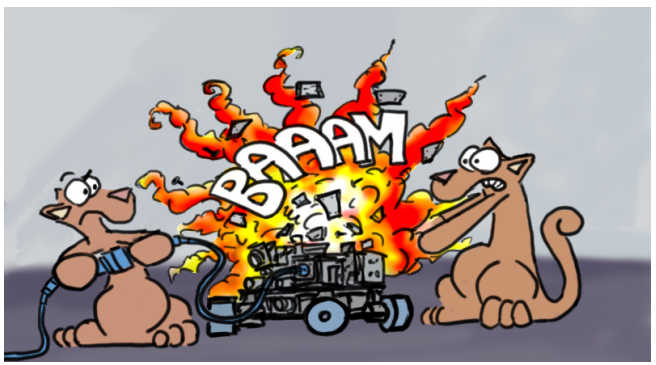
Enrichissement de la hiérarchie

- On ajoute 2 classes (ayant une méthode `taxe()`) : `VoitureElectrique` (EstUn `Voiture`) et `Bateau` (EstUn `Vehicule`)

Quel impact ?

Il faut toucher au code utilisateur !!!

```
t = 0
if isinstance( v1, VoitureElectrique )
    or isinstance( v1, Voiture )
    or isinstance( v1, Bateau ) :
  t = v1.taxe()
```



SCIENCES SORBONNE UNIVERSITÉ

Plan

SCIENCES SORBONNE UNIVERSITÉ

Définition

Idée

- Déclarer une méthode dans la classe mère (sans l'implémenter)
  - On décrit les besoins de la méthode (paramètres d'entrée)
- Implémenter cette méthode dans les classes filles
  - On code ce que fait la méthode

Une telle méthode est dite **abstraite**

Remarque :

- Toute classe ayant une méthode abstraite doit aussi être déclarée abstraite.

SCIENCES SORBONNE UNIVERSITÉ

Définition

Dangers :

- Une méthode abstraite doit pouvoir être vue et implémentée par les héritiers
  $\Rightarrow$ elle doit être soit publique soit protégée
- On ne peut pas instancier de classe abstraite
  $\Rightarrow$ cela évite d'appeler une méthode dont on ne connait pas le comportement

Conséquences

Les classes abstraites

- sont les classes les plus hautes dans la hiérarchie
- définissent une architecture et obligent les classes dérivées à la respecter

Définition

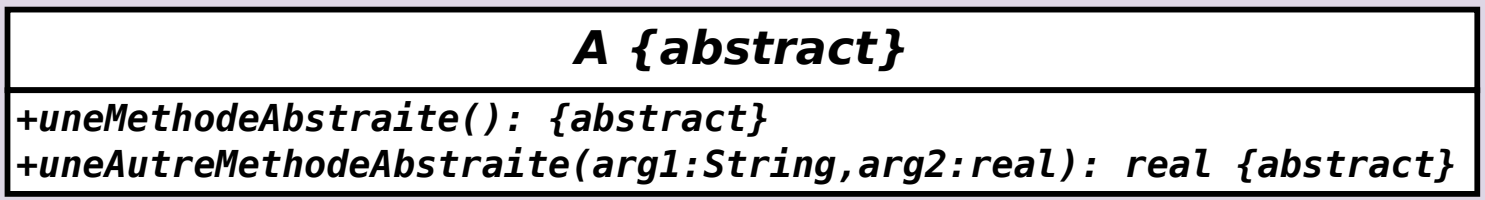
Contenu d'une classe abstraite

- Attributs publics ou privés
- Méthodes implémentées
  - publiques : invocables depuis l'extérieur
  - privées : invocables dans la classe uniquement
- Méthodes abstraites **publiques**
  - Pas d'implémentation !
  - ex : calculer la taxe d'un vehicule

Définition

Représentation UML

- Pour une classe : représentée en *italique* ou par {abstract}
- Pour une méthode : représentée en *italique* ou par {abstract}



Syntaxe en Python

- Il faut importer des modules d'abc (Abstract Base Class) :

```
from abc import ABC, abstractmethod
```

- Pour une classe : on hérite de ABC

```
class A (ABC) :  ...  # Classe abstraite A
```

- Pour une méthode :
  - on précède la méthode du décorateur @abstractmethod
  - la méthode ne fait (généralement) rien : pass

```
@abstractmethod
def nomMethode( self, ... ) :  pass
```

SCIENCES SORBONNE UNIVERSITÉ

Exemple : les véhicules revisitées

Classe Vehicule revisitée

```
from abc import ABC, abstractmethod   # Import des modules
class Vehicule ( ABC ) :  # Classe abstraite
  def __init__( self, annee, marque ) :
    self.annee = annee
    self.marque = marque

  # Code inchange pour ces methodes
  def who( self ) :        ...
  def __str__( self ) :    ...
  def age( self ) :        ...

  # Declaration d'une methode abstraite
  @abstractmethod
  def taxe( self ) :  pass
```

Vehicule {abstract}
-annee: integer -marque: String
#<<create>> Vehicule(a:integer,m:String) +age(): integer +who(): String +toString(): String ***+taxe(): real {abstract}***

SCIENCES SORBONNE UNIVERSITÉ

Exemple : les véhicules revisitées

Classe Voiture

```
class Voiture ( Vehicule ) :
  def __init__( self, annee, marque, puissance ) :
    # Appel du constructeur de Vehicule
    super( ).__init__( annee, marque)
    self.puissance = puissance

  def who( self ) :      ...
  def __str__( self ):   ...

  # Implementation de la fonction abstraite taxe (obligatoire)
  def taxe( self ) :     return self.puissance * 10 + 50.
```

Programme principal

```
v = Voiture( 2014, "Renault", 5 )
print( vo )
print( "Taxe:", vo.taxe( ) )
```

affiche :

```
Voiture - marque Renault construit en 2014 de puissance 5 CV
Taxe: 100
```

Erreurs classiques

Classe abstraite

```
# Oubli de l'import de abc

# Oubli de l'heritage d'ABC (methodes abstraites presentes)
class Vehicule :
  # Methode abstraite et privee: interdit
  @abstractmethod
  def __accelerer( self ) : pass

  # Methode sans implementation et non abstraite: dangereux
  def taxe( self ) :  pass
```

Erreurs classiques

Classe fille

```
class Voiture ( Vehicule ) :
  def __init__(self, annee, marque, puissance ) : ...
  def who( self ) : ...

  # Pas d'implementation de la methode abstraite taxe
  # => Voiture reste une classe abstraite!
```

Programme principal

```
# Construction d'une classe abstraite
ve = Vehicule( 2005, "Rockrider" )
```

SCIENCES SORBONNE UNIVERSITÉ

Plan

SCIENCES SORBONNE UNIVERSITÉ

Exercice

Énoncé

Que pourra-t'on lire à l'écran lors de l'exécution du programme suivant ?

```
class A :
  def fA( self, obj ) :
    if isinstance( obj, A ) :      print( "A.fA avec A" )
    elif isinstance( obj, B ) :    print( "A.fA avec B" )
    else :                         print( "A.fA avec ??" )

class B (A) :
  def fA( self, obj ) :
    if isinstance( obj, B ) :      print( "B.fA avec B" )
    elif isinstance( obj, A ) :    print( "B.fA avec A" )
    else :                         print( "B.fA avec ??" )

class C (B) :
  pass     # Classe sans definition

vA = A( );        vB = B( );        vC = C( )
vA.fA( vA );      vA.fA( vB );      vA.fA( vC )
vB.fA( vA );      vB.fA( vB );      vB.fA( vC )
vC.fA( vA );      vC.fA( vB );      vC.fA( vC )
```

SCIENCES SORBONNE UNIVERSITÉ

Étude de cas

Énoncé

On souhaite réaliser un logiciel de géométrie permettant de construire et manipuler des figures géométriques. Il faudra pouvoir intégrer différents types de figures :

- Figures simples : Point
- Polygones : Triangle, Carré, Losange, Rectangle, Parallélépipède
- Courbes : Segment, Béziers, Lignes brisées, Cercles, Ellipses

et le logiciel pourra être étendu pour en ajouter de nouvelles.
Nous souhaitons pouvoir transformer les figures : translater, mettre à l'échelle, calculer des distances et des surfaces (figures fermées), et bien sûr afficher les figures.

SCIENCES SORBONNE UNIVERSITÉ

Étude de cas

Identification des concepts

On souhaite réaliser un logiciel de géométrie permettant de construire et manipuler des **figures géométriques**. Il faudra pouvoir intégrer différents types de figures :

- **Figures simples** : **Point**
- **Polygones** : **Triangle**, **Carré**, **Losange**, **Rectangle**, **Parallélépipède**
- **Courbes** : **Segment**, **Béziers**, **Lignes brisées**, **Cercles**, **Ellipses**

et le logiciel pourra être étendu pour en ajouter de nouvelles.
Nous souhaitons pouvoir transformer les figures : translater, mettre à l'échelle, calculer des distances et des surfaces (**figures fermées**), et bien sûr afficher les figures.

Étude de cas

Identification des concepts concrets

On souhaite réaliser un logiciel de géométrie permettant de construire et manipuler des **figures géométriques**. Il faudra pouvoir intégrer différents types de figures :

- **Figures simples** : Point
- **Polygones** : Triangle, Carré, Losange, Rectangle, Parallélépipède
- **Courbes** : Segment, Béziers, Lignes brisées, Cercles, Ellipses

et le logiciel pourra être étendu pour en ajouter de nouvelles.
Nous souhaitons pouvoir transformer les figures : translater, mettre à l'échelle, calculer des distances et des surfaces (**figures fermées**), et bien sûr afficher les figures.

Rappel : Classes concrètes

Les classes concrètes sont les classes que l'on souhaitera effectivement **construire en mémoire**

SCIENCES SORBONNE UNIVERSITÉ

Étude de cas

Identification des concepts abstraits

On souhaite réaliser un logiciel de géométrie permettant de construire et manipuler des figures géométriques. Il faudra pouvoir intégrer différents types de figures :

- Figures simples : Point
- Polygones : Triangle, Carré, Losange, Rectangle, Parallélépipède
- Courbes : Segment, Béziers, Lignes brisées, Cercles, Ellipses

et le logiciel pourra être étendu pour en ajouter de nouvelles.
Nous souhaitons pouvoir transformer les figures : translater, mettre à l'échelle, calculer des distances et des surfaces (figures fermées), et bien sûr afficher les figures.

Rappel : Classes abstraites

Les classes abstraites sont les classes que l'on souhaitera ne pas **construire en mémoire**

SCIENCES SORBONNE UNIVERSITÉ

Étude de cas

Identification des traitements

On souhaite réaliser un logiciel de géométrie permettant de construire et manipuler des figures géométriques. Il faudra pouvoir intégrer différents types de figures :

- Figures simples : Point
- Polygones : Triangle, Carré, Losange, Rectangle, Parallélépipède
- Courbes : Segment, Béziers, Lignes brisées, Cercles, Ellipses

et le logiciel pourra être étendu pour en ajouter de nouvelles.
Nous souhaitons pouvoir transformer les figures : translater, mettre à l'échelle, calculer des distances et des surfaces (figures fermées), et bien sûr afficher les figures.

SCIENCES SORBONNE UNIVERSITÉ

Étude de cas

Résultat : modèle objet

- Concepts abstraits : Classes abstraites + héritage éventuel
  - Figure, Polygone hérite de Figure, Courbe hérite de Figure
- Concepts concrets : Classes concrètes (+ héritage)
  - Point
  - Segment hérite de Courbe (idem Beziers, etc.)
  - Triangle hérite de Polygone (idem Carre, Rectangle, etc.)
  - etc. (exercice)
- Traitements = Méthodes (privilégier les classes mères)
  - afficher, translater, mettre à l'échelle et calculer la distance dans Point
  - afficher, translater et mettre à l'échelle dans Figure
  - calculer la longueur dans Courbe
  - calculer la surface dans Polygone

SCIENCES SORBONNE UNIVERSITÉ

Étude de cas