Paradigmes de programmation

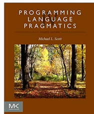
Cours 6 : Éléments de programmation orientée objet

Benoît Montagu — benoit.montagu@inria.fr

Préparation à l’agrégation d’informatique — Automne 2022

Programmation orientée objet : références ?

- Concepts généraux de POO :



Michael Scott (déc. 2015). Programming Language Pragmatics. Fourth Edition. San Francisco, CA : Morgan Kaufmann Pub. 992 p. ISBN : 9780124104099

Chapitre 9 : Data Abstraction and Object Orientation

- Peu de ressources de bonne qualité qui soient *indépendantes* d'un langage de programmation...
- À défaut : un bon livre sur Python ou Java
- Je suis preneur si vous avez des bonnes références
- Python est un choix *étonnant* pour parler de POO : certains concepts n'y existent tout simplement pas
  (type dynamique/statique, méthodes abstraites, classes abstraites, interfaces, mots-clefs pour la visibilité...)

Présentation générale

Qu'est-ce que la programmation orientée objet ?

- POO/OOP (Object Oriented Programming)
- La POO est un *style* de programmation où on fournit données et opérations sur ces données dans un même *objet*
- Ce style peut être imposé ou encouragé par un langage de programmation
- La POO est orthogonale au concept de typage statique ou dynamique
- On peut faire de la POO *sans* utiliser de système de classe
- Exemples de langages supportant la POO :
  Java, Scala, C++, Objective-C, C#, Eiffel, Python, Lua, Ruby,
  Smalltalk, JavaScript, OCaml…
- Les traits objet d'OCaml sont hors programme (typage très avancé) et ne sont pas traités dans ce cours

Concepts importants en POO

1. L'encapsulation
   - Objet = données + fonctions de manipulation
   - But : contrôler l'interface offerte par un objet pour cacher des détails d'implémentation
   - Types de données abstraits

Concepts importants en POO

1. L'encapsulation
   - Objet = données + fonctions de manipulation
   - But : contrôler l'interface offerte par un objet
     pour cacher des détails d'implémentation
   - Types de données abstraits
2. L'héritage
   - Technique pour réutiliser du code existant
   - Permet d'étendre/raffiner/modifier le comportement de code existant
   - Utilisation des classes comme un moyen d'organiser le code
   - Les classes sont des « fabriques à objets »
   - Les classes servent de support à l'héritage
   - Moins fréquent : des langages pour la POO *sans* classes

Concepts importants en POO

1. L'encapsulation
   - Objet = données + fonctions de manipulation
   - But : contrôler l'interface offerte par un objet pour cacher des détails d'implémentation
   - Types de données abstraits
2. L'héritage
   - Technique pour réutiliser du code existant
   - Permet d'étendre/raffiner/modifier le comportement de code existant
   - Utilisation des classes comme un moyen d'organiser le code
   - Les classes sont des « fabriques à objets »
   - Les classes servent de support à l'héritage
   - Moins fréquent : des langages pour la POO *sans* classes
3. La liaison dynamique de méthodes (ou liaison tardive, dispatch dynamique)
   - C'est l'ingrédient-clef pour l'extension de code *a posteriori*
   - Nous l'avons déjà vu : c'est de la récursion ouverte
   - ⚠ Peut conduire à un flot de contrôle complexe
   - ⚠ C'est différent de la *surcharge* (overloading) et de la *redéfinition* (overriding)
   - ⚠ C'est différent de la *portée dynamique* des variables

Objets

Qu'est-ce qu'un objet ? Rappels de vocabulaire

- Un objet est une structure (enregistrement, record…)
- Les éléments de cette structures sont appelés :
  - champs
  - propriétés
  - membres
  - attributs
- Il y a deux sortes d'attributs :
  - Des variables d'instance : ces champs contiennent des valeurs, qui définissent *l'état interne* de l'objet
  - Des méthodes : ces champs contiennent des fonctions, qui vont créer d'autres objets, modifier l'état de l'objet courant…
- Un objet regroupe donc en une même entité :
  - Des données
  - Les fonctions qui opèrent sur ces données
- Un objet peut être mutable ou immuable

Méthodes d'un objet

Considérons un objet `obj` implémentant un compteur

- `obj` possède un état interne contenant un entier
- `obj` expose 2 méthodes :
  - Une méthode `get` pour récupérer l'état interne de l'objet
    On dit que `get` est un *accesseur*
  - Une méthode `add` qui ajoute un entier à son état interne
    On dit que `add` est un *modifieur*

L'expression `obj.get()` est appelée :

- Appel de la méthode `get` de l'objet `obj`
- Envoi du message `get` à l'objet `obj`

L'objet `obj` est appelé l'objet *receveur* du message `get`

Méthodes d’un objet

Considérons un objet `obj` implémentant un compteur

- `obj` possède un état interne contenant un entier
- `obj` expose 2 méthodes :
  - Une méthode `get` pour récupérer l’état interne de l’objet
    On dit que `get` est un *accesseur*
  - Une méthode `add` qui ajoute un entier à son état interne
    On dit que `add` est un *modifieur*

L’expression `obj.get()` est appelée :

- Appel de la méthode `get` de l’objet `obj`
- Envoi du message `get` à l’objet `obj`

L’objet `obj` est appelé l’objet *receveur* du message `get`

ℹ Vocabulaire anthropomorphique introduit par Smalltalk

- Les objets sont vus comme des « agents » qui réagissent à des messages

⚠ C’est en fait assez loin de la réalité…

Méthodes et récursion ouverte (1/2)

- Les méthodes d'un objet prennent comme premier argument l'objet lui-même
- Dans certains langages, c'est implicite (exemple : Java)
  On utilise le mot-clef **`this`** pour référer à l'objet courant
- Dans d'autres langages, c'est explicite (exemple : Python)

```
def add(self, n):
    self.state += n
```

  On a *choisi* `self` comme nom de variable, mais ce n'est qu'une convention
- Lors d'un l'appel de méthode, l'objet est donné comme premier argument à la fonction qui est associée à la méthode
- Le code `obj.add(42)` effectue réellement l'appel `C.add(obj, 42)`
  où `C` est la *classe* de l'objet `obj`
  (c-à-d la classe qui a été utilisée pour *créer* l'objet)

Méthodes et récursion ouverte (2/2)

```
1  class C:
2      # Constructeur
3      def __init__(self):
4          self._state = 0
5
6      # Accesseur
7      def get(self):
8          return self._state
9
10     # Modifieur
11     def add(self, n):
12         if n > 0:
13             self._state += n // 2
14             if n % 2 == 1:
15                 self._state += 1
16             self.add(n // 2)
```

- Convention de nommage : le champ `self._state` est privé à l'objet

Méthodes et récursion ouverte (2/2)

```
class C:
    # Constructeur
    def __init__(self):
        self._state = 0

    # Accesseur
    def get(self):
        return self._state

    # Modifieur
    def add(self, n):
        if n > 0:
            self._state += n // 2
            if n % 2 == 1:
                self._state += 1
            self.add(n // 2)
```

- Convention de nommage : le champ `self._state` est privé à l'objet
- Les méthodes ne sont pas définies récursivement
- L'appel de méthode `obj.add(n)` est implémenté par l'appel de la fonction `C.add(obj, n)` où `C` est la classe de l'objet `obj`
- ```
  obj = { "_state": 0,
          "get": lambda : C.get(obj),
          "add": lambda n : C.add(obj, n)
   } # Encore un point fixe!
  ```
- La récursion est nouée lors de l'appel
- `self` est donc résolu à l'appel des méthodes, pas lors de leur définition
  ```

Des objets sans classes

En Python, les objets sont implémentés par des dictionnaires

```
obj.__class__.__dict__["add"](obj, 42) # identique à obj.add(42)
obj.__dict__["_state"]                 # identique à obj._state
```

Écrivons le même exemple de compteur, encodé avec des dictionnaires, sans utiliser les classes :

```
1  def create():  # Constructeur
2      _obj = {}
3      _obj["_state"] = 0
4      _obj["add"] = add
5      _obj["get"] = get
6      return _obj
```

```
7  def get(self):  # Accesseur
8      return self["_state"]
9
10
11 def add(self, n):  # Modifieur
12     if n > 0:
13         self["_state"] += n // 2
14         if n % 2 == 1:
15             self["_state"] += 1
16         self["add"](self, n // 2)
```

Classes

Classes

- Une classe est une « fabrique à objets »
- Les objets créés à partir d'une classe `C` sont appelés des instances de `C`
- Un classe définit :
  - Quel sera « l'état » d'une instance (c-à-d ses variables d'instance)
  - Quelles seront les méthodes d'une instance
  - Un constructeur : définit l'état initial d'une instance fraîchement créée
  - Certains langages permettent de définir plusieurs constructeurs

Classes

- Une classe est une « fabrique à objets »
- Les objets créés à partir d'une classe C sont appelés des instances de C
- Un classe définit :
  - Quel sera « l'état » d'une instance (c-à-d ses variables d'instance)
  - Quelles seront les méthodes d'une instance
  - Un constructeur : définit l'état initial d'une instance fraîchement créée
  - Certains langages permettent de définir plusieurs constructeurs
- Les objets créés à partir d'une même classe :
  - Présentent la même interface : ils ont les mêmes méthodes
  - Ont les mêmes variables d'instance (mais chaque objet a son état propre)
  - ⚠ En Python, un constructeur peut choisir de définir certaines variables d'instance ou d'autres, en fonction de la valeur d'un paramètre...
  - ⚠ En Python, on peut ajouter dynamiquement des champs à un objet, après qu'il a été créé...

Classes

- Une classe est une « fabrique à objets »
- Les objets créés à partir d'une classe `C` sont appelés des instances de `C`
- Un classe définit :
  - Quel sera « l'état » d'une instance (c-à-d ses variables d'instance)
  - Quelles seront les méthodes d'une instance
  - Un constructeur : définit l'état initial d'une instance fraîchement créée
  - Certains langages permettent de définir plusieurs constructeurs
- Les objets créés à partir d'une même classe :
  - Présentent la même interface : ils ont les mêmes méthodes
  - Ont les mêmes variables d'instance (mais chaque objet a son état propre)
  - ⚠ En Python, un constructeur peut choisir de définir certaines variables d'instance ou d'autres, en fonction de la valeur d'un paramètre...
  - ⚠ En Python, on peut ajouter dynamiquement des champs à un objet, après qu'il a été créé...
- Une classe peut aussi définir :
  - Des « méthodes statiques » : c-à-d des fonctions qui ne dépendent pas de `self`
  - Des variables de classe : ce sont des variables globales du programme

Classes : un exemple

```python
class Complex:
    __nb_instances = 0

    @staticmethod
    def number_of_instances():
        return Complex.__nb_instances

    def __init__(self, x, y):
        self.real = x
        self.imag = y
        Complex.__nb_instances += 1

    def __str__(self):
        x = str(self.real)
        y = str(self.imag)
        return x + " + " + y + " i"

    def add(self, cplx):
        self.real += cplx.real
        self.imag += cplx.imag

    def conjugate(self):
        x = self.real
        y = self.imag
        return Complex(x, -y)

    @staticmethod
    def basis():
        one = Complex(1, 0)
        i = Complex(0, 1)
        return (one, i)
```

Types abstraits

Encapsulation et types abstraits

Les méthodes et variables d'instance d'un objet définissent son interface :

- Quelles opérations un client peut-il effectuer sur un objet ?
- À quelles parties de l'état d'un objet un client a-t-il accès ?

Encapsulation et types abstraits

Les méthodes et variables d'instance d'un objet définissent son interface :

- Quelles opérations un client peut-il effectuer sur un objet ?
- À quelles parties de l'état d'un objet un client a-t-il accès ?

Encapsulation : c'est le fait de regrouper des données et les fonctions sur ces données afin de définir une *interface*, de manière à ce qu'un client ne puisse pas contourner cette interface

👉 En pratique, cela demande de restreindre l'accès aux variables d'instance, pour ne pas révéler des détails d'implémentation (variables privées)

Encapsulation et types abstraits

Les méthodes et variables d'instance d'un objet définissent son interface :

- Quelles opérations un client peut-il effectuer sur un objet ?
- À quelles parties de l'état d'un objet un client a-t-il accès ?

Encapsulation : c'est le fait de regrouper des données et les fonctions sur ces données afin de définir une *interface*, de manière à ce qu'un client ne puisse pas contourner cette interface

👉 En pratique, cela demande de restreindre l'accès aux variables d'instance, pour ne pas révéler des détails d'implémentation (variables privées)

Type abstrait : un type abstrait est un objet dont l'interface ne permet pas à un client de connaître son implémentation

👉 **Avec un type abstrait, on sépare interface et implémentation :** un client ne dépend que de l'interface. C'est un bon principe de programmation !

En Python, c'est illusoire... à cause de l'introspection

Types abstraits et types abstraits

Côté Python :

```
class Counter:

    def __init__(self):
        ...
    def add(self, n):
        ...
    def get(self):
        ...
```

- Type abstrait et interface sont confondus
- La valeur abstraite est l'objet complet
- Les variables d'instance sont privées

Côté OCaml :

```
module Counter : sig
  type t
  val create: unit -> t

  val add: t -> int -> unit

  val get: t -> int
end = struct
  ...
end
```

- Type abstrait et interface sont séparés
- La valeur abstraite est distincte de ses fonctions de manipulation
- L'implémentation est cachée (type opaque)

Exercice

- Réimplémenter en Python une classe pour les nombres complexes, qui utilise une autre représentation (par exemple : coordonnées polaires)
- Votre implémentation respectera la même interface que celle vue précédemment
- Un client ne devra pas être capable de distinguer ces deux implémentations

  (on suppose un client « bien discipliné », qui ne fera pas d'introspection)

Héritage

Classes et héritage

L'héritage (simple) permet de définir une nouvelle classe à partir d'une ancienne.

Avec l'héritage, il est possible de :

- Réutiliser du code existant
- Étendre l'état des instances d'une classe (ajouter des variables d'instance)
- Étendre les fonctionnalités d'une classe (ajouter des méthodes)
- Redéfinir le comportement de méthodes existantes (*overriding*)

Classes et héritage

L'héritage (simple) permet de définir une nouvelle classe à partir d'une ancienne.
Avec l'héritage, il est possible de :

- Réutiliser du code existant
- Étendre l'état des instances d'une classe (ajouter des variables d'instance)
- Étendre les fonctionnalités d'une classe (ajouter des méthodes)
- Redéfinir le comportement de méthodes existantes (*overriding*)

En Python :

```
class B(A):
# B hérite de A
    ...
```

Vocabulaire :

- B hérite de A
- B dérive de A
- B étend A
- B spécialise A
- B est une sous-classe de A
- A est une super-classe de B
- B est une classe fille de A
- A est une classe mère de B

- Le mot-clef `super` permet de référer au code de la classe mère :
  `super` désigne la version de `self` dans la classe mère

Exemple d’héritage : compteur avec histoire

```
1  class CHist(C):
2      def __init__(self):
3          # inherit the initialization of instance variables
4          super().__init__()
5          # store the "history" of updates in a list
6          self._hist = []
7
8      # all calls to `add' in the super-class are replaced with
9      # this version of `add', including the "recursive" calls!
10     def add(self, n):
11         # record previous state
12         self._hist.append(self.get())
13         # then, perform the update
14         super().add(n)
15
16     def history(self):
17         # return the history list
18         return self._hist
```

Même exemple, sans classes

```
1  def create_with_hist():
2      _obj = create()  # inherit the initialization of instance variables
3      _obj["_hist"] = []  # store the "history" of updates in a list
4      # add/redefine methods
5      _obj["add"] = add_with_hist
6      _obj["history"] = history
7      return _obj
8
9
10 def add_with_hist(self, n):
11     self["_hist"].append(self["get"](self))  # record previous state
12     add(self, n)  # then, perform the update
13
14
15 def history(self):
16     return self["_hist"]  # return the history list
```

- Toutes les classes dérivent (transitivement) de la classe `object`

```
>>> class A:
...     pass
...
>>> issubclass(A,object)
True
```

En savoir plus sur les classes de Python

- Toutes les classes dérivent (transitivement) de la classe `object`

```
>>> class A:
...     pass
...
>>> issubclass(A,object)
True
```

- Certains noms de méthodes sont réservés :
  - La méthode `__init__` est appelée lors de l'appel au constructeur de la classe
  - `obj.__str__()` est appelée lors d'un appel à `print(obj)`
  - `obj1.__add__(obj2)` est appelée lors d'un appel à `obj1 + obj2`
  - Les itérateurs `for x in ...` sont aussi des appels dissimulés à des méthodes

- Les exceptions dérivent de la classe `Exception` :

```
>>> class E(Exception):
...     pass  # ici, E ne contient pas de valeur
...
>>> try:
...     raise E
... except E:  # "as exc": exc est l'instance de E qui est attrapée
...     print("E was raised")
...
E was raised
```

En savoir plus sur les classes de Python

- Les exceptions dérivent de la classe `Exception` :

```
>>> class E(Exception):
...     pass  # ici, E ne contient pas de valeur
...
>>> try:
...     raise E
... except E:  # "as exc": exc est l'instance de E qui est attrapée
...     print("E was raised")
...
E was raised
```

- La valeur **`None`** de la classe `NoneType` est l'équivalent de `( )` en OCaml
  On peut utiliser **`None`** comme le **`null`** de Java pour initialiser des champs

Implémentation de l'héritage : 2 grandes familles

- Par concaténation/agrégation :
  - Chaque objet *pointe vers* toutes ses méthodes
  - La création d'objet peut être coûteuse (en temps et en mémoire) :
    on crée un dictionnaire comprenant les méthodes définies par la classe
    (et par ses classes parentes)
  - L'accès à une méthode se fait en cherchant directement dans le dictionnaire qui implémente l'objet
  - C'est l'approche de l'exemple précédent

Implémentation de l'héritage : 2 grandes familles

- Par concaténation/agrégation :
  - Chaque objet *pointe vers* toutes ses méthodes
  - La création d'objet peut être coûteuse (en temps et en mémoire) : on crée un dictionnaire comprenant les méthodes définies par la classe (et par ses classes parentes)
  - L'accès à une méthode se fait en cherchant directement dans le dictionnaire qui implémente l'objet
  - C'est l'approche de l'exemple précédent
- Par délégation :
  - Chaque objet *pointe vers* sa classe
  - Une structure de données décrit cette classe (méthodes, classes parentes...)
  - La création d'objet est peu coûteuse : on n'ajoute pas les méthodes à l'objet, mais seulement sa classe
  - L'accès à une méthode peut être coûteuse : indirection via le dictionnaire de la classe, parcours dans la hiérarchie de classes
  - C'est l'approche utilisée en Python

Implémentation de l'héritage : 2 grandes familles

- Par concaténation/agrégation :
  - Chaque objet *pointe vers* toutes ses méthodes
  - La création d'objet peut être coûteuse (en temps et en mémoire) : on crée un dictionnaire comprenant les méthodes définies par la classe (et par ses classes parentes)
  - L'accès à une méthode se fait en cherchant directement dans le dictionnaire qui implémente l'objet
  - C'est l'approche de l'exemple précédent
- Par délégation :
  - Chaque objet *pointe vers* sa classe
  - Une structure de données décrit cette classe (méthodes, classes parentes...)
  - La création d'objet est peu coûteuse : on n'ajoute pas les méthodes à l'objet, mais seulement sa classe
  - L'accès à une méthode peut être coûteuse : indirection via le dictionnaire de la classe, parcours dans la hiérarchie de classes
  - C'est l'approche utilisée en Python

? Exercice : réécrire l'exemple précédent en utilisant l'approche par délégation

Exercice d'héritage

1. Définir en Python une classe `LinkedList` qui implémente des listes chaînées (mutables) d'entiers (sans utiliser les `list` Python). Méthodes requises :
   - Ajout d'un élément en tête
   - Concaténation
   - Conversion en chaîne de caractères
2. Dans un second temps, définir une classe `LinkedListWithLength` :
   - Qui dérive de `LinkedList`
   - Qui implémente une méthode qui calcule la longueur d'une liste en temps constant
   - Les objets de cette nouvelle classe auront une variable d'instance supplémentaire, qui contient la longueur de la liste
   - On spécialisera les méthodes d'insertion en tête et de concaténation sans dupliquer de code de la classe mère. On utilisera `super()` a cet effet
   - Il pourra s'avérer utile de changer la classe mère et d'exporter de nouvelles méthodes…

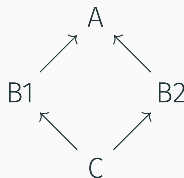
Faites vraiment cet exercice : il est intéressant et assez difficile

Héritage multiple : quelques notions

- Certains langages de programmation supportent l'héritage multiple : une classe peut hériter de plusieurs autres
- C'est par exemple possible en C++, Python, OCaml…
- Héritage simple : la relation d'héritage entre classes est un **arbre**
- Héritage multiple : la relation d'héritage est un **graphe acyclique**

Héritage multiple : quelques notions

- Certains langages de programmation supportent l'héritage multiple : une classe peut hériter de plusieurs autres
- C'est par exemple possible en C++, Python, OCaml…
- Héritage simple : la relation d'héritage entre classes est un **arbre**
- Héritage multiple : la relation d'héritage est un **graphe acyclique**
- ⚠ L'héritage multiple pose des problèmes sémantiques :
  - Supposons deux classes `B1` et `B2` qui définissent une méthode `m`
  - Supposons que `C` hérite de `B1` et `B2`
  - Supposons que l'objet `c` est une instance de `C`
  - Quel est le sens de `c.m()` ?

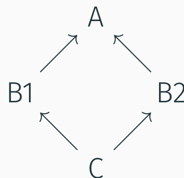


Héritage multiple : quelques notions

- Certains langages de programmation supportent l'héritage multiple : une classe peut hériter de plusieurs autres
- C'est par exemple possible en C++, Python, OCaml...
- Héritage simple : la relation d'héritage entre classes est un **arbre**
- Héritage multiple : la relation d'héritage est un **graphe acyclique**

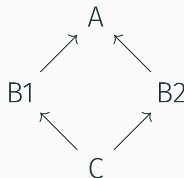
⚠ L'héritage multiple pose des problèmes sémantiques :

- Supposons deux classes `B1` et `B2` qui définissent une méthode `m`
- Supposons que `C` hérite de `B1` et `B2`
- Supposons que l'objet `c` est une instance de `C`
- Quel est le sens de `c.m()` ?
- Et si `B1` et `B2` ont hérité la méthode `m` depuis une même classe `A`, mais que `B2` l'a redéfinie ?



Héritage multiple : quelques notions

- Certains langages de programmation supportent l'héritage multiple : une classe peut hériter de plusieurs autres
- C'est par exemple possible en C++, Python, OCaml…
- Héritage simple : la relation d'héritage entre classes est un **arbre**
- Héritage multiple : la relation d'héritage est un **graphe acyclique**
- ⚠ L'héritage multiple pose des problèmes sémantiques :
  - Supposons deux classes `B1` et `B2` qui définissent une méthode `m`
  - Supposons que `C` hérite de `B1` et `B2`
  - Supposons que l'objet `c` est une instance de `C`
  - Quel est le sens de `c.m()` ?
  - Et si `B1` et `B2` ont hérité la méthode `m` depuis une même classe `A`, mais que `B2` l'a redéfinie ?



- En pratique, les langages de programmation
  - Soit choisissent une méthode de résolution (C++, Python)
  - Soit demandent au programmeur d'être explicite (OCaml)
- ⚠ Pas de panique : l'héritage multiple est *explicitement* hors programme

Sous-typage et héritage

Classes, types, typage nominal ou structurel

- En Python, le « type » d'un objet est la classe dont il est instance :

```
>>> type("Hi!")
<class 'str'>
>>> type(str)
<class 'type'>
```

- C'est aussi le cas en Java : le type d'un objet est sa classe
- En Java, le typage est nominal :
  - Deux types sont égaux ssi ils ont le même nom
  - Une définition de classe définit systématiquement un *nouveau* type
  - Deux classes qui ont le même contenu sont nécessairement différentes car elles ont un nom différent

Classes, types, typage nominal ou structurel

- En Python, le « type » d'un objet est la classe dont il est instance :

```
>>> type("Hi!")
<class 'str'>
>>> type(str)
<class 'type'>
```

- C'est aussi le cas en Java : le type d'un objet est sa classe
- En Java, le typage est nominal :
  - Deux types sont égaux ssi ils ont le même nom
  - Une définition de classe définit systématiquement un *nouveau* type
  - Deux classes qui ont le même contenu sont nécessairement différentes car elles ont un nom différent
- Dans le cas où le nom n'importe pas, on parle de typage structurel :
  - Deux types sont équivalents s'ils ont des *définitions* équivalentes
- Par exemple, en OCaml :
  - Le typage des enregistrements et des types algébriques est nominal
  - Le typage des tuples est structurel (et des variants polymorphes, et des objets...)

Sous-typage structurel

- Avec un système de types structurel, le *nom* des types ou des classes n'importe pas
- Les règles de sous-typage sont définies sur la structure des types (c-à-d sur l'interface des objets)

Width

$$\frac{n \leq m}{\{f_1 : \tau_1, \ldots, f_n : \tau_n, f_{n+1} : \tau_{n+1}, \ldots, f_m : \tau_m\} \leq \{f_1 : \tau_1, \ldots, f_n : \tau_n\}}$$

Depth

$$\frac{\tau_i \leq \tau_i' \qquad 1 \leq i \leq n}{\{f_1 : \tau_1, \ldots, f_i : \tau_i, \ldots, f_n : \tau_n\} \leq \{f_1 : \tau_1, \ldots, f_i : \tau_i', \ldots, f_n : \tau_n\}}$$

Trans

$$\frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

Sous-typage : covariance et contravariance

Le sous-typage est :

- Covariant à droite de la flèche (type du résultat)
- Contravariant à gauche de la flèche (type de l'argument)

$$\textsc{Arrow}\quad \frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

Exemple : $\{a : \tau_a\} \to \{b : \tau_b, c : \tau_c\} \leq \{a : \tau_a, d : \tau_d\} \to \{c : \tau_c\}$

C'est cohérent avec le polymorphisme de sous-typage de la POO

Et avec la règle de *subsumption*

$$\textsc{Subsumption}\quad \frac{\Gamma \vdash t : \tau_1 \qquad \tau_1 \leq \tau_2}{\Gamma \vdash t : \tau_2}$$

Sous-typage par héritage (1/2)

Dans les langages où le type d'un objet est (le nom de) sa classe :

- La relation de sous-typage est la relation d'héritage entre classes
- $\tau_1 \leq \tau_2$: « τ_1 est sous-type de τ_2 »
- On a $\tau_1 \leq \tau_2$ lorsque $\tau_1 = \tau_2$ ou τ_1 hérite (transitivement) de τ_2
- Si $\tau_1 \leq \tau_2$, alors un objet `obj1` de type τ_1 possède plus de méthodes ou de variables d'instances qu'un objet `obj2` de type τ_2
  C'est le sous-typage en largeur (« *width subtyping* »)
- Si `obj1` de type τ_1 et `obj2` de type τ_2 ont une méthode f qui retourne un objet, alors : le type de retour de `obj1.f` est aussi un sous-type du type de retour de `obj2.f`
  C'est le sous-typage en profondeur (« *depth subtyping* »)

Sous-typage par héritage (2/2)

- Dans les langages typés comme Java, cette contrainte de sous-type est vérifiée statiquement par le compilateur lors de la déclaration d'héritage :

```
class A {}
class B extends A {}

class C {
    A foo() { return new A(); }
}
class D extends C {
    @Override
    B foo() { return new B(); } // depth subtyping
    // return type B must be a subtype of A

    void bar() { return; } // width subtyping
}
```

Polymorphisme de sous-typage (1/2)

- Dans le contexte de la programmation objet :
  - « Si une fonction/méthode accepte un argument de type τ, alors elle peut aussi accepter un argument qui a pour type un sous-type de τ »

```
1   class A {}
2   class B extends A {}
3
4   class C {
5       int foo(A a) { return 0; }
6       int bar() {
7           return foo(new B()); // subtype polymorphism
8       }
9   }
```

  - C'est une conséquence de la règle de « *subsumption* » :
    « Si un objet a le type τ, alors il a aussi n'importe quel super-type de τ »

Polymorphisme de sous-typage (2/2)

⚠ <u>Dans le contexte de la théorie des langages de programmation :</u>

- Le polymorphisme de sous-typage est du « polymorphisme borné » (« *bounded polymorphism* », « *bounded quantification* »)
- C'est utilisé pour modéliser les langages orientés objet
- Utilisable en Java, Scala… par des experts
- Par exemple :
  $\forall(\alpha \leq \tau).\ (\alpha \times \alpha) \rightarrow \text{bool}$
  est le type d'une fonction de comparaison pour des valeurs de type τ

ℹ C'est plus fin que la notion de POO de polymorphisme de sous-typage, et c'est *largement* hors programme…

Polymorphisme de sous-typage en Python ? (1/4)

- En Java, la relation de sous-typage est imposée lors des appels de fonctions
  Pour que l'appel `f(x)` soit accepté par le compilateur, il faut que :
  - La fonction `f` ait pour type $A \to B$
  - et que l'argument `f` ait pour type A'
  - Avec $A' \leq A$

  C'est une contrainte sur les relations d'héritage entre classes

Polymorphisme de sous-typage en Python ? (1/4)

- En Java, la relation de sous-typage est imposée lors des appels de fonctions
  Pour que l'appel `f(x)` soit accepté par le compilateur, il faut que :
  - La fonction `f` ait pour type $A \to B$
  - et que l'argument `f` ait pour type A'
  - Avec $A' \leq A$

  C'est une contrainte sur les relations d'héritage entre classes
- En Python : le type est vérifié dynamiquement, lorsque l'argument est utilisé, et sans se baser sur la classe des objets
  - C'est la présence des méthodes dans l'argument qui est testée
  - ⚠ La classe de l'argument n'importe pas
  - Ce qui importe, c'est le fait qu'une classe définisse une méthode ou non
  - Toutefois, la fonction *peut choisir* de tester la classe de son argument

Polymorphisme de sous-typage en Python ? (1/4)

- En Java, la relation de sous-typage est imposée lors des appels de fonctions
  Pour que l'appel `f(x)` soit accepté par le compilateur, il faut que :
  - La fonction `f` ait pour type $A \to B$
  - et que l'argument `f` ait pour type A'
  - Avec $A' \leq A$

  C'est une contrainte sur les relations d'héritage entre classes
- En Python : le type est vérifié dynamiquement, lorsque l'argument est utilisé, et sans se baser sur la classe des objets
  - C'est la présence des méthodes dans l'argument qui est testée
  - ⚠ La classe de l'argument n'importe pas
  - Ce qui importe, c'est le fait qu'une classe définisse une méthode ou non
  - Toutefois, la fonction *peut choisir* de tester la classe de son argument
- En Python : en pratique, le polymorphisme de sous-typage se réduit à
  « si une méthode accepte un objet **`obj1`** en argument, alors cette méthode acceptera tout objet **`obj2`** qui possède plus de méthodes que **`obj1`** »

Polymorphisme de sous-typage en Python ? (2/4)

Exemple en Java :

```
1   class A {
2       void f() { System.out.println("A.f"); }
3   }
4
5   class B { // not a subclass of A
6       void f() { System.out.println("B.f"); }
7   }
8
9   class C {
10      void g(A a) { a.f(); }
11      void h() {
12          g(new A());
13          // g(new B()); would not typecheck
14          // because B is not a subclass of A
15      }
16  }
```

Exemple en Python :

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

class B:  # not a subclass of A
    def f(self):
        print("B.f")

def g(a):  # no check is performed on the class of `a'
    a.f()
    if isinstance(a, A):  # check the class of the argument
        print("Argument is of class A")

g(A())  # prints "A.f" "Argument is of class A"
g(B())  # prints "B.f"
```

Polymorphisme de sous-typage en Python ? (4/4)

En Python, on peut dire que :

- C'est l'interface offerte par un objet qui est importante, pas sa classe
- Interface ≈ quelles méthodes sont disponibles ≈ type structurel

Polymorphisme de sous-typage en Python ? (4/4)

En Python, on peut dire que :

- C'est l'interface offerte par un objet qui est importante, pas sa classe
- Interface ≈ quelles méthodes sont disponibles ≈ type structurel
- « *Duck typing* » ? ⚠ Notion non scientifique et mal définie...

Polymorphisme de sous-typage en Python ? (4/4)

En Python, on peut dire que :

- C'est l'interface offerte par un objet qui est importante, pas sa classe
- Interface ≈ quelles méthodes sont disponibles ≈ type structurel
- « *Duck typing* » ? ⚠ Notion non scientifique et mal définie…
- On peut faire plus en utilisant des fonctions d'introspection :
  - Tester dynamiquement la présence d'un champ :

```
if hasattr(obj, "f"):
    print("f est présent")
else:
    print("f est absent")
```

  - Réagir à l'exception levée par l'accès à un champ absent :

```
try:
    obj.f()
    print("f est présent")
except AttributeError:
    print("f est absent")
```

Héritage, interfaces, sous-typage

- Le sous-typage est une relation entre les types des valeurs
  (c-à-d entre les interfaces qu'offrent ces valeurs)
- L'héritage est une relation entre des classes
  (c-à-d entre des fragments de programmes)
- ⚠ Il n'y a pas de lien entre ces deux relations…

  … bien que certains langages les confondent

Héritage, interfaces, sous-typage

- Le sous-typage est une relation entre les types des valeurs (c-à-d entre les interfaces qu'offrent ces valeurs)
- L'héritage est une relation entre des classes (c-à-d entre des fragments de programmes)

⚠ Il n'y a pas de lien entre ces deux relations…

… bien que certains langages les confondent

- Exercice : trouver des classes S1 et S2 telles que
  1. S1 est sous-classe de S2, et leurs instances ont des interfaces qui sont bien sous-types l'une de l'autre
  2. S1 n'est pas sous-classe de S2, et leurs instances ont des interfaces qui ne sont pas non plus sous-types l'une de l'autre
  3. S1 n'est pas sous-classe de S2, mais leurs instances ont des interfaces qui sont pourtant sous-types l'une de l'autre
  4. S1 est sous-classe de S2, mais leurs instances ont des interfaces qui ne sont pourtant pas sous-types l'une de l'autre (Indice : pensez au type de `self`)

Programmation orientée objet et programmation fonctionnelle

Exercice : comment écrire un code utilisant des classes, et qui offre les mêmes fonctionnalités que le code OCaml suivant ?

```
type expr =
| Num of int
| Plus of expr * expr

let rec interp = function
| Num n -> n
| Plus (e1, e2) -> interp e1 + interp e2
```

Donner des solution dans les langages suivants :

1. En Java
2. En Python
3. Quelle serait la façon idiomatique de faire en C (sans classes bien sûr) ?

Types de données algébriques à base de classes : discussion

Comparer les modifications à faire en OCaml, et celles à faire en Python pour effectuer les changements suivants :

1. Ajouter une nouvelle fonction `size`

Comparer les modifications à faire en OCaml, et celles à faire en Python pour effectuer les changements suivants :

1. Ajouter une nouvelle fonction `size`
2. Ajouter un nouveau cas `mult` au type des expressions

Types de données algébriques à base de classes : discussion

Comparer les modifications à faire en OCaml, et celles à faire en Python pour effectuer les changements suivants :

1. Ajouter une nouvelle fonction `size`
2. Ajouter un nouveau cas `mult` au type des expressions
3. Ajouter une fonction `reassoc` qui réassocie les additions à droite

Visiteurs (1/3)

- Un visiteur est similaire à une opération `fold`, mais qui est extensible, c-à-d qui est écrit en style de récursion ouverte

Visiteurs (1/3)

- Un visiteur est similaire à une opération `fold`, mais qui est extensible, c-à-d qui est écrit en style de récursion ouverte
- Un visiteur est un objet qui comprend autant de méthodes que de cas dans le type algébrique que nous avons représenté

  Classe abstraite (pour mimer Java) :

```
class Visitor:
    def visit_num(self, num):
        pass

    def visit_plus(self, plus):
        pass
```

Visiteurs (1/3)

- Un visiteur est similaire à une opération `fold`, mais qui est extensible, c-à-d qui est écrit en style de récursion ouverte
- Un visiteur est un objet qui comprend autant de méthodes que de cas dans le type algébrique que nous avons représenté

  Classe abstraite (pour mimer Java) :

```
class Visitor:
    def visit_num(self, num):
        pass

    def visit_plus(self, plus):
        pass
```

- Chaque classe correspondant à un cas du type algébrique explique comment évaluer un visiteur

  Exemple pour la classe `Num` :

```
    def accept(self, visitor):
        return visitor.visit_num(self)
```

Visiteurs (2/3)

- Ensuite, on peut écrire un visiteur qui interprète notre expression arithmétique :

```
class InterpVisitor(Visitor):
    def visit_num(self, num):
        return num.get_value()

    def visit_plus(self, plus):
        n1 = plus.get_left().accept(self)
        n2 = plus.get_right().accept(self)
        return n1 + n2
```

- On peut enfin interpréter une expression, en utilisant notre visiteur :
  `expr.accept(InterpVisitor())`
- Notre visiteur est extensible !
  Écrire une version qui trace les évaluations des `Plus`, en utilisant l'héritage

Visiteurs (3/3)

Exercice :

- Écrire une classe `CopyVisitor` qui dérive de `Visitor`, dont les instances vont créer une copie de l'expression arithmétique visitée
- Écrire une classe `SimplifyVisitor` qui dérive de `CopyVisitor`, et qui remplace les sous-expressions de la forme `Plus(Num(0), e)` et `Plus(e, Num(0))` par l'expression `e`

Conclusion

Conclusion

Bilan :

- Un rapide tour d'horizon de la programmation orientée objet
- Rappel des notions principales de la POO
- Un aperçu de comment les programmes objets sont implémentés
- Extensibilité par récursion ouverte
- POO ≈ fonctions de 1re classe + état mutable + récursion ouverte
- On l'a vu dans les cours précédents : il est difficile d'utiliser tous ces ingrédients simultanément
- Et pourtant, la POO est un style de programmation très répandu
- Programmer pour l'extensibilité est *très* délicat…

Conclusion

Bilan :

- Un rapide tour d'horizon de la programmation orientée objet
- Rappel des notions principales de la POO
- Un aperçu de comment les programmes objets sont implémentés
- Extensibilité par récursion ouverte
- POO ≈ fonctions de 1re classe + état mutable + récursion ouverte
- On l'a vu dans les cours précédents : il est difficile d'utiliser tous ces ingrédients simultanément
- Et pourtant, la POO est un style de programmation très répandu
- Programmer pour l'extensibilité est *très* délicat…

Notion non abordées dans ce cours :

- *Design patterns* autres que le pattern visiteur (*singleton*, *factory*…)
- *Inner classes*
- Exemples d'héritage multiple