Objets et classes

Arnaud Labourel arnaud.labourel@univ-amu.fr

Section 1

Types et structures de données

Types simples

Un même type basique peut correspondre à des données très différentes.

Par exemple, un entier peut correspondre à :

- un âge exprimé en année
- une température exprimée en Celsius
- un prix exprimé en €

```
int temperatureInCelsius = 8;
int age = 8;
int price = 8;
```

Fonction sur les types simples (1/2)

```java
public boolean isSenior(int age) {
  return age >= 70;
}
```

Techniquement, on peut utiliser cette fonction sur toutes les données entières :

```java
ageIsSenior(age);
ageIsSenior(temperatureInCelsius);
ageIsSenior(price);
```

Cela n'a pas de sens pour les entiers ne correspondant pas à un âge :

⇒ la fonction ne devrait pouvoir être appliquée qu'à des « âges »

Fonction sur les types simples (2/2)

```
public float toFahrenheit(int temperatureInCelsius) {
  return 9 / 5 * temperatureInCelsius + 32;
}
```

Techniquement, on peut utiliser cette fonction sur toutes les données entières :

```
toFahrenheit(temperatureInCelsius);
toFahrenheit(age);
toFahrenheit(price);
```

La fonction ne devrait pouvoir être appliquée qu'à des températures.

$\Rightarrow$ les types de base ne suffisent pas

On a besoin de distinguer âges et températures même si les deux données sont représentées par un entier

Types composites

Parfois, on a besoin de plusieurs données pour représenter des notions.

On souhaite travailler sur des personnes caractérisées par leur nom et année de naissance (données).

```
int a1 = 2004;
String a2 = "Bob";
int b1 = 2001;
String b2 = "Alice";
```

Des variables séparées ne conviennent pas, car on souhaiterait que les données soient regroupées.

Solution : types composites

Il faut un type composite (type contenant des éléments de différents types), par exemple en Python un dictionnaire

```
>>> alice = { "year" : 2001, "name" : "Alice" }
>>> bob = { "year" : 2006, "name" : "Bob" }
def is_adult(person) :
  return 2022 - person['year'] >= 18
>>> is_adult(alice)
True
>>> is_adult(bob)
False
```

Définir une fonction commune pour les variables `bob` et `alice` est possible, car elles partagent la même structure de clés.

Partager des clés n'est pas suffisant

On pourrait aussi considérer des livres caractérisés par leur année de parution et leur nom :

```
>>> lotr = { "year" : 1954,
             "name" : "Seigneur des Anneaux" }
>>> is_adult(lotr)
True
```

La fonction `is_adult` n'a pas de sens pour des livres et ne devrait pouvoir être appliqué qu'à des « personnes ». On a le même problème que pour les types simples.

$\Rightarrow$ partager les mêmes clés ne suffit pas

Bilan

- Les types de base (simples ou composites) ne suffisent pas

  $\Rightarrow$ il faut pouvoir distinguer un « int age » et un « int temperature »

- Des données composites de même nature doivent partager la même structure

- La possibilité d'appliquer une opération à une donnée doit être contrôlée selon la nature de la donnée

$\Rightarrow$ il faut pouvoir définir ses propres **types**

Type : besoin

Un type définit les valeurs possibles et les opérations autorisées.

Exemples :

- un type `Temperature`
  - valeurs possibles : la valeur en degré Celsius de type `int` ≥ -273
  - opérations autorisées : obtenir la valeur en Fahrenheit
- un type `Age`
  - valeurs possibles : une donnée de type `int` ≥ 0
  - opérations autorisées : savoir si l'âge correspond à un âge senior
- un type `Personne`
  - valeurs possibles : une année de naissance de type `int` et un nom de type `String`
  - opérations autorisées : savoir si la personne est majeure, obtenir l'âge

Section 2

Objet et classe

Principe programmation objet

- En programmation objet, les données sont représentées par des objets.
- La programmation objet consiste essentiellement à définir des types d'objets, c'est-à-dire des classes.

Objet

- un objet est composé de données (= d'autres objets)
- un objet peut exécuter des opérations
- un objet a un type et le type d'un objet définit :
  - la **structure** des données qui composent cet objet : **les attributs**
  - les **opérations** (= **traitements**) que peut exécuter cet objet : **les méthodes**

Classes

Une **classe** est un type objet.

Une classe définit :

- les **méthodes** et les traitements associés ⇒ le **comportement** des objets
- la structure des **attributs** nécessaires à la réalisation des traitements ⇒ l'**état** des objets

le **comportement** agit sur l'**état** et l'**état** influence le **comportement**

Méthodes et attributs

Méthode

Une **méthode** est une fonction qui appartient à une classe :
« *function member* »

- une méthode ne peut être utilisée que par les objets de la classe
- un objet ne peut exécuter que les méthodes de sa classe

Attribut

Un **attribut** est une variable qui appartient à un objet :
« *data member* » (synonymes : *property*, *field*, ...)

- les attributs sont définis par la classe de l'objet
- chaque attribut a un type

Instance

Une classe permet de **créer** des objets.

Ces objets sont du type de cette classe.

Définition d'instance

On appelle **instance** un objet créé par une classe.
Tout objet est l'instance d'une classe.

Exemples

- `"toto"` correspond à une instance de la classe `String` (un objet issu de la classe `String`).
- `new ArrayList()` construit une instance de la classe `ArrayList` (un objet issu de la classe `ArrayList`).

Définition d'une classe

1. **Définir les attributs** des objets de la classe
   - Tous les objets d'une même classe ont **la même structure d'attributs**
   - Les valeurs des attributs sont « personnelles » à chaque objet, elles peuvent être **différentes** d'un objet à l'autre.
2. **Définir les méthodes** que pourront exécuter les objets de la classe
   - Tous les objets d'un même classe peuvent exécuter les mêmes méthodes
   - Appeler (ou invoquer) une méthode sur un objet n'affecte que l'objet en question
   - Appeler une méthode sur un objet peut changer la valeur de ses attributs (effet de bord)

Exemple d'attributs d'une classe `Person`

On peut donc définir une classe correspondant à une personne et contenant deux attributs :

- une année de naissance : `int` nommé `yearOfBirth`
- un nom : `String` (chaîne de caractères) nommé `name`

On utilise la syntaxe suivante dans un fichier `Person.java`

```
public class Person {
  int yearOfBirth;
  String name;
}
```

Accès aux attributs

Pour accéder aux attributs d'un objet il faut utiliser la syntaxe suivante (notation pointée) :

`nomDeLaVariable.nomDeLAttribut`

Si on est dans le code d'une méthode, on peut seulement utiliser :

- `nomDeLAttribut` ou
- `this.nomDeLAttribut`

pour accéder à l'attribut de l'objet courant.

Exemples

- `person.name` pour accéder à l'attribut `name` d'une instance `person` de type `Person`
- `array.length` pour accéder à l'attribut `length` d'un tableau `array` de type `int[]`

Exemple de méthodes d'une classe `Person`

On peut définir des **méthodes** qu'on peut appeler à partir d'une personne.

```
public class Person {
  int yearOfBirth;
  String name;
  int getAge(){
    return 2023 - yearOfBirth;
  }
  boolean isAdult(){
    return getAge() >= 18;
  }
}
```

Les méthodes sont liées aux objets de la classe et se définissent sans le mot-clé `static`.

Accès aux méthodes

Une méthode s'appelle avec une instance de la classe.

Pour accéder aux méthodes d'un objet il faut utiliser la syntaxe suivante (notation pointée) :

`nomDeLaVariable.nomDeLaMéthode(arguments)`

Si on est dans le code d'une méthode, on peut seulement utiliser `nomDeLaMéthode(arguments)` ou `this.nomDeLaMéthode(arguments)` pour appeler une méthode sur l'objet courant.

Exemples

- `person.getAge()` pour appeler la méthode `getAge` sans argument sur l'objet `person` de type `Person`
- `string.charAt(i)` pour appeler la méthode `charAt` avec un argument `i` sur l'objet `string` de type `String`

Constructeurs

À l'exécution, il faut créer les objets conformément au modèle.

Constructeur

Pour créer un objet, il faut utiliser un **constructeur**.
Chaque appel à un constructeur crée un **nouvel** objet (instance) qui obéit au modèle défini par la classe du constructeur.

Rôles du constructeur

Un constructeur a deux rôles :

1. créer les attributs de l'objet (la structure de l'état) $\Rightarrow$ réserver l'espace mémoire (automatique en Java)
2. donner les valeurs initiales aux attributs (« initialiser l'objet » : code du constructeur)

Chaque classe doit définir comment sont initialisés les attributs
$\rightarrow$ il peut y avoir plusieurs manières de réaliser cette initialisation

Exemple de constructeur d'une classe Person

```java
public class Person {
  int yearOfBirth;
  String name;

  // Méthodes getAge et isAdult omises

  Person(int yearOfBirth, String name) {
    this.yearOfBirth = yearOfBirth;
    this.name = name;
  }
}
```

Accès au constructeur d'une classe

Appel du constructeur

`new nomDeLaClasse(arguments)`

Exemples

- `new Person("Alice", 2001)` crée un objet de type `Person`
- `new ArrayList()` crée un objet de type `ArrayList`

Constructeur implicite

En Java, si une classe ne définit pas de constructeur, alors il y a un constructeur par défaut (constructeur sans paramètre n'ayant pas d'instruction réservant juste l'espace mémoire pour l'objet)
$\rightarrow$ il existe seulement s'il n'y a pas d'autre constructeur déclaré

Références

- L'appel à un constructeur a pour résultat une référence vers l'objet créé.
- Cette référence = une adresse vers l'identité de l'objet.

  Elle peut être stockée dans une variable (de type objet).

Important

- La référence permet d'accéder à l'objet, mais n'est pas l'objet lui-même.
- Une variable objet contient l'information pour accéder à l'objet (et potentiellement le modifier).
- Deux variables ayant la même référence d'un objet (par exemple après une affectation `a = b;`) pointe vers un objet unique $\Rightarrow$ toute modification de l'objet est visible via les deux variables.

Ajout d'autres méthodes dans Person

```java
public class Person {
  int year;
  String name;

  //Code des transparents précédents

  boolean isOlderThan(Person person){
    return getAge() > person.getAge();
  }

  @Override
  public String toString() {
    return name + " (" + year + ")";
  }
}
```

Diagramme de classe

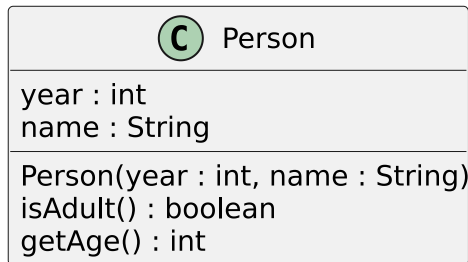
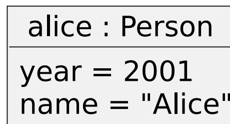


Diagramme de classe

Manière “simple” de représenter une classe avec :

- son nom
- ses attributs (avec leur type)
- ses constructeurs (types des paramètres)
- ses méthodes (types des paramètres et du retour de la méthode)

Diagramme d'objets



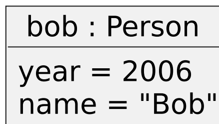


Diagramme d'objet

Manière “simple” de représenter un objet avec :

- son type et
- les valeurs de ses attributs.

Section 3

Appel de méthodes

Utilisation d'objet

On interagit avec un objet uniquement via :

- ses méthodes
- ses attributs

Appel/invocation de méthode

Appel/invocation de méthode = « envoi de message »
L'objet a le contrôle sur le message qui lui est envoyé :
$\Rightarrow$ seulement les méthodes définies dans la classe de l'objet sont autorisées.
Utilisation de la notation pointée :

```
aPerson.isAdult()
```

Exemple : classe `Thermometer`

Classe `Thermometer`

- structure attributs : `temperature` de type `float`
- méthodes : `getValue()`, `toFahrenheit()`, `toString()`

`t1` et `t2` deux instances de la classe `Thermometer`

$\Rightarrow$ même structure d'attributs, mais valeurs différentes

t1 : Thermometer	
temperature	12.0

t2 : Thermometer	
temperature	20.1

Exemple de code : classe Thermometer

```
public class Thermometer {
  private float temperature; // attribut
  public Thermometer(float initialTemp) {
    // constructeur
    this.temperature = initialTemp;
  }
  public float temperatureInCelsius() {
    // méthode
    return this.temperature;
  }
  public void changeTemperature(float newTemp) {
    // méthode
    this.temperature = newTemp;
  }
  //...
```

Exemple de code : classe Thermometer

```java
  // suite

  public String toString() { // méthode
    return
      + this.temperature + "°C";
  }
  public float temperatureInFahrenheit() { // méthode
    return (9.0f / 5.0f) * this.temperature + 32;
  }
}
```

Exemple : classe `Person`

Classe Person :

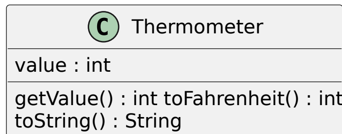
- structure attributs : `yearOfBirth` de type `int`, `name` de type `String`
- méthodes : `isAdult()`, `isOlderThan(Person)`, `getYearOfBirth()`, `getName()`, `toString()`

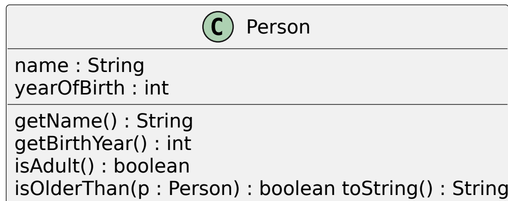
`alice` et `bob` deux instances de la classe `Person`

alice : Person	
name	"Alice"
yearOfBirth	2001

bob : Person	
name	"Bob"
yearOfBirth	2006

Exemples d'utilisations






```
t1.toFahrenheit() // -> 53.6
t2.toFahrenheit() // -> 68
t1.toString() // -> "18°C"
t1.isAdult() // impossible (erreur à la compilation)
alice.getBirthYear() // -> 19
alice.isAdult() // -> true
alice.toString() // -> "Alice is born in 2001"
alice.toFahrenheit()
// impossible (erreur à la compilation)
```

Envoi de message

Conformément à sa classe (son type), l'objet « contrôle » le message :

- sa légalité
- pas d'ambiguïté pour deux messages (méthodes) de même nom dans des classes différentes (cf. `toString()`)

Règle d'appel de méthode

- Une méthode ne peut pas être utilisée autrement qu'en étant appelée/invoquée sur un objet via un envoi de message à cet objet
- La validité du message pour le type de la référence est vérifié à la compilation

Receveur

L'objet invoquant = le **receveur** du message

Il fait partie du contexte d'exécution de la méthode

⇒ dans le code d'une méthode, on peut :

- appeler une méthode sur le receveur
- accéder aux attributs du receveur

Messages en cascade

Les attributs d'un objet étant eux-mêmes des objets : il peut y avoir des messages en cascade.
⇒ Appel d'une méthode d'un receveur peut entraîner l'appel d'une méthode sur un de ses attributs.

Exemple thermomètres, chaudières et thermostats

```
Boiler
--------------------------
power : int
on : boolean
--------------------------
Boiler(power : int)
isOn() : boolean
isOff() : boolean
turnOn() turnOff()
getPower() : int
```

```
Thermostat
--------------------------
thermo : Thermometer
boiler : Boiler
targetTemp : float
--------------------------
Thermostat(b:Boiler,target:float)
currentTemperature(): float
getTargetTemp() : float
setTargetTemp(target : float)
temperatureChange(newT:float)
update()
```

```java
Boiler b = new Boiler(1000);
Thermostat t = new Thermostat(b, 20);
t.temperatureChange(25);
// envoi de message en cascade t -> b
```

Classe et instance

Classe = définition/modèle

- décrit la structure de l'état des objets : les attributs et leurs types
- définit les envois de messages possibles : les méthodes
  $\Rightarrow$ **interface** d'une classe

Abstrait

Instance = objet conforme au modèle de la classe qui l'a créé

- son état correspond à la structure définie par sa classe
  $\rightarrow$ association de valeurs aux attributs
- n'accepte que les messages définis par la classe
  = n'exécute que les méthodes définies par sa classe

Concret

Section 4

Classes : à retenir absolument

Définition d'une classe

Une **classe** (d'objet) définit des :

- **constructeurs** : des façons de construire/instancier les objets (**instances** de la classe)
- **attributs** (champs, propriétés ou données membres) : la structure des objets de la classe
- **méthodes** : le comportement des objets de la classe

Syntaxe de définition d'une classe

```java
public class NameOfTheClass{
    Type1 attribute1;
    Type2 attribute2;

    NameOfTheClass(Type3 argument1, Type4 argument2){
        // constructor code
    }

    Type2 getAttribute2(){
        return this.attribute2;
    }
}
```

Syntaxe d'utilisation d'une classe

```
>>> Person p = new Person(2000, "Bob");
>>> p.year;
2000
>>> p.name = "Alice";
>>> p.name
Alice
>>> p.getAge();
22
>>> p.isAdult();
true
```