Programmation orientée objet. Cours 4

Marie-Pierre Béal

Interface.
Polymorphisme.

Interface

Le problème

Supposons que l'on a plusieurs sortes de véhicules

```
public record Car(int seats) {}
```

```
public record Bus(long weight) {}
```

Et que l'on veut les afficher

```
var list = List.of(new Car(3), new Bus(5_000));
for(var vehicle: list) {
  IO.println(vehicle.toString());
}
// Car[seats=3]
// Bus[weight=5000]
```

Pourquoi ce code marche ?

La liste est typée `List<Object>`

- `Car` est un `Object`
- `Bus` est un `Object`
- et `Object` possède une méthode `toString()`

```
List<Object> list = List.of(new Car(3), new Bus(5_000));
for(Object vehicle: list) {
  IO.println(vehicle.toString()); // Object.toString()
}
```

Et s'il y a une méthode commune ?

Ajoutons une façon de calculer les taxes différentes.

```
public record Car(int seats) {
  public long computeTax() { return seats * 50; }
}
```

```
public record Bus(long weight) {
  public long computeTax() { return weight * 2; }
}
```

Et on veut faire la somme des taxes.

```
var list = List.of(new Car(3), new Bus(5_000));
var tax = 0L;
for(var vehicle: list) {
  tax = tax + vehicle.computeTax();
              // ne compile pas ! Problème de typage
}             // il n'y a pas de méthode Object.computeTax()
IO.println(tax);
              // il nous faut un type équivalent à  "Car | Bus"
              // qui possède une méthode long computeTax()
```

Interface

Une interface permet de d'écrire un super-type commun à plusieurs classes. Les classes `Car` et `Bus` sont sous-types de `Vehicle`.

```
// dans fichier Vehicle.java
public interface Vehicle {}
```

```
// dans fichier Car.java
public record Car(int seats) implements Vehicle {
  public long computeTax() { return seats * 50; }
}
```

```
// dans fichier Bus.java
public record Bus(long weight) implements Vehicle {
  public long computeTax() { return weight * 2; }
}
```

"implements" indique que l'on est "une sorte de"

Mais pas suffisant ...

Le code ne marche toujours pas

```
List<Vehicle> list = List.of(new Car(3), new Bus(5_000));
var tax = 0L;
for(Vehicle vehicle: list) {
  tax = tax + vehicle.computeTax();
                // ne compile pas !
                // pas de méthode Vehicle.computeTax()
}
IO.println(tax);
```

Il faut aussi déclarer une méthode `computeTax()` dans l'interface `Vehicle` !

Méthode abstraite

Une méthode abstraite

- est une méthode sans code
- force une classe qui implémente l'interface contenant une méthode abstraite à fournir un code pour cette méthode

On ajoute une méthode abstraite `computeTax()` à Vehicle

```
public interface Vehicle {
  public abstract long computeTax();
}
```

On demande à ce que la méthode abstraite soit implantée

Interface + méthode abstraite

On obtient le code suivant

```
public interface Vehicle {
  long computeTax(); // dans une interface les méthodes
                     // sont public abstract par défaut
}
```

```
public record Car(int seats) implements Vehicle {
  @Override  // @Override marche aussi avec l'implantation
             // de méthode abstraite, l'annotation n'est pas
             // obligatoire mais elle rend le code plus lisible
  public long computeTax() { return seats * 50; }
}
```

```
public record Bus(long weight) implements Vehicle {
  @Override
  public long computeTax() { return weight * 2; }
}
```

Et ça marche !

Avec l'interface et la méthode abstraite, le code compile ET marche

```
List<Vehicle> list = List.of(new Car(3), new Bus(5_000));
var tax = 0L;
for(Vehicle vehicle: list) {
  tax = tax + vehicle.computeTax();
                // appelle Vehicle.computeTax()
                // Il y a de la magie ici, car la méthode
                // Vehicle.computeTax() n'a pas de code !!
}
IO.println(tax);
```

Dynamic dispatch / Late binding

Liaison tardive en français

Lorsque l'on appelle une méthode sur un type

- À l'exécution, la machine virtuelle regarde la classe de l'objet sur lequel la méthode est appelée et appelle la méthode de cette classe

À la compilation

- L'appel `vehicle.computeTax()` est typé `Vehicle::computeTax()`

À l'exécution

- Si la variable `vehicle` contient
  - un objet de la classe `Car`, `Car::computeTax()` est appelée
  - un objet de la classe `Bus`, `Bus::computeTax()` est appelée

Pourquoi c'est intéressant ?

Le code qui utilise l'interface est un code générique

```
public static long sumAllTaxes(List<Vehicle> vehicles) {
  var tax = 0L;
  for(var vehicle: vehicles) {
    tax = tax + vehicle.computeTax();
  }
  return tax;
}
```

Et en même temps, spécialisé, car la "bonne" méthode `computeTax()` est appelée.

En résumé

Une interface permet de créer un super-type commun.

- Un sous-type doit déclarer qu'il implémente l'interface avec le mot clé implements

Une méthode abstraite "foo" dans une interface force à avoir une méthode commune à tous les sous-types.

La notation `o.foo()` appelle la méthode de la classe de la référence "o" à l'exécution automatiquement (dynamic dispatch).

Cela permet d'écrire des méthodes génériques, spécialisées et extensibles.

En terme de design

Lorsque l'on écrit une librairie/application, on raisonne souvent dans l'autre sens.

- Créer une interface et essayer de tous rentrer dedans ne marche pas comme approche

On regarde les endroits où l'on va gérer des objets différents de la même façon

- Pour cet endroit, on va créer une interface que l'on passe en paramètre
- Le code qui utilise l'interface est plus important que l'interface en elle-même

Sous-typage multiple

Implanter plusieurs interfaces

Une classe peut implanter plusieurs interfaces. Cela permet d'utiliser une même classe dans plusieurs contextes.

```
public interface Displayable {
  void display(Screen screen);
}
```

```
public interface Collidable {
  boolean collideWith(Collidable c);
}
```

```
public class SpaceShip implements Displayable, Collidable {
  ...
}
```

Sous-typage multiple

Une même instance peut alors être vue comme une instance d'une interface ou une instance de l'autre interface.

```
public class Game {
  private final ArrayList<Displayable> displayables;
  private final ArrayList<Collidable> collidables;
  ...
  public void add(SpaceShip spaceShip) {
    Objects.requireNonNull(spaceShip);
    displayables.add(spaceShip);
    collidables.add(spaceShip);
  }
}
```

Membre d'une interface

Une interface, comme une classe ou un record, peut contenir des membres :

- les membres sont public par défaut et peuvent être déclarés private (mais pas d'autre visibilité)

Elle contient

- des champs, mais toujours static
- des méthodes d'instance, abstract par défaut
  - méthodes abstract : elles doivent être implantées
  - méthodes default (par défaut) : peuvent être remplacées
  - méthodes private
- des méthodes static

Méthodes par défaut

Dans une interface, une méthode par défaut (default) est une méthode d'instance pas abstraite qui est utilisée si une sous-classe ne déclare pas la méthode

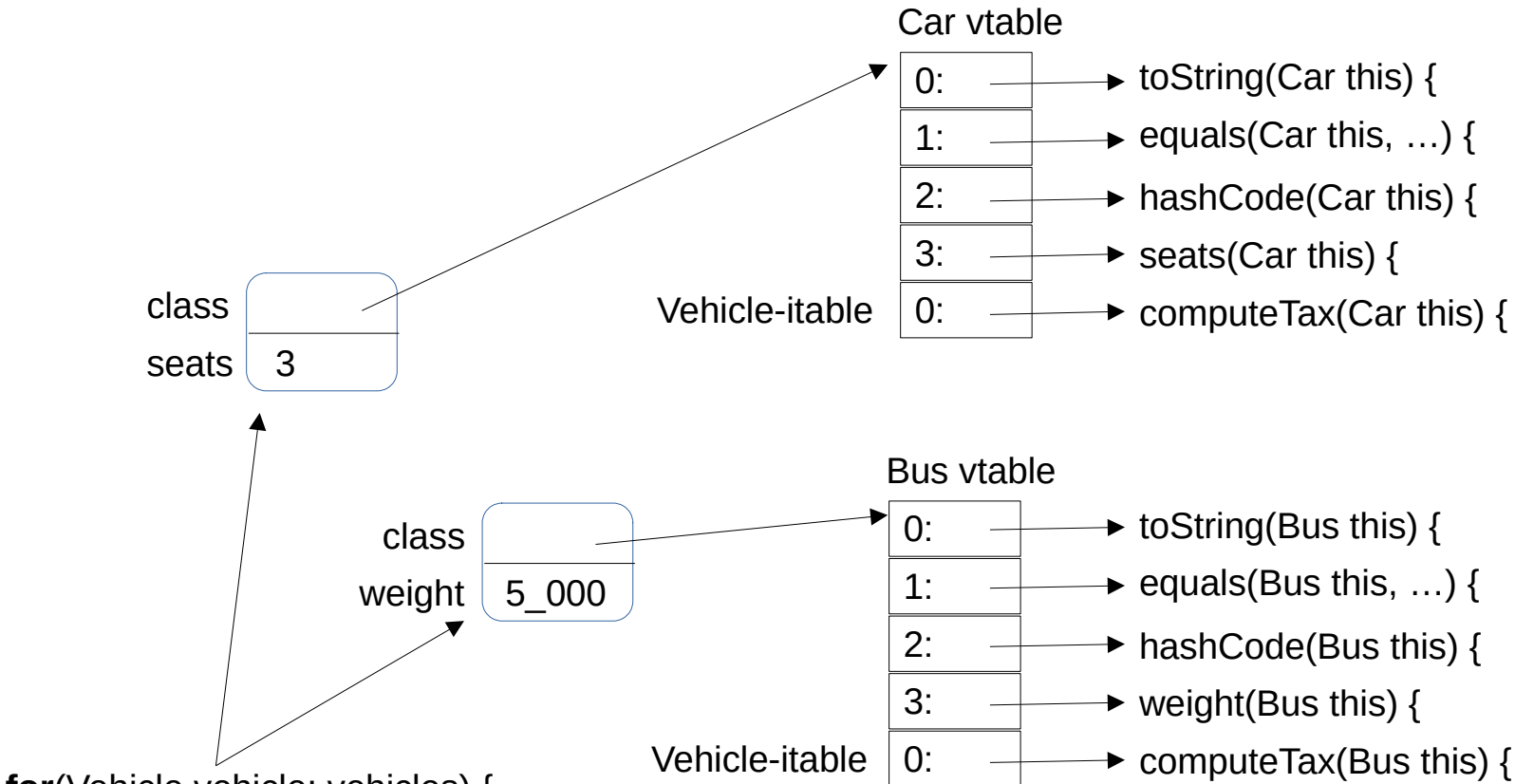
```
public interface Investment {
  default boolean gambling() { return true; }
}
```

```
public record House() implements Investment {
  @Override
  public boolean gambling() { return false; }
                              // remplace la méthode
}
```

```
public record BitCoin() implements Investment {
// pas de méthode gambling,
// donc celle de Investment est utilisée
}
```

Comment ça marche en mémoire

public interface Vehicle { long computeTax(); }
public record Car(int seats) **implements** Vehicle { **long** computeTax(Car this) { … } }
public record Bus(long weight) **implement** Vehicle { **long** computeTax(Bus this) { … } }




```
for(Vehicle vehicle: vehicles) {
  vehicle.computeTax()  // vehicle.Vehicle_itable[0]
}
```

Le type enum

Le type enum est un type de données spécial qui permet d'assigner à une variable une valeur choisie dans un ensemble de constantes prédéfinies.

```
public enum WeekDay {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

```
public record Day(WeekDay weekDay) {
  @Override
  public String toString() {
    return switch (weekDay) {
      case MONDAY -> "Monday";
      case TUESDAY -> "Tuesday";
      case WEDNESDAY -> "Wednesday";
      case THURSDAY -> "Thursday";
      case FRIDAY -> "Friday";
      case SATURDAY -> "Saturday";
      case SUNDAY -> "Sunday";
    };
  }
}
```

Le type enum

```
public class Main {
  public void main() {
   var day = new Day(WeekDay.MONDAY);
   IO.println(WeekDay.MONDAY);
   IO.println(day);
   IO.println(WeekDay.MONDAY.ordinal());
   for(var weekDay : WeekDay.values()) {
     IO.println(weekDay.ordinal() + " : " + weekDay);
   }
  }
}
// MONDAY
// Monday
// 0
// 0 : MONDAY
// 1 : TUESDAY
// 2 : WEDNESDAY
// 3 : THURSDAY
// 4 : FRIDAY
// 5 : SATURDAY
// 6 : SUNDAY
```

En résumé

En résumé

Une interface est un type abstrait qui permet de considérer plusieurs classes de la même façon.

- Une interface définit des méthodes abstraites qui doivent être implémentées par les sous-classes
- Une sous-classe indique qu'elle implante l'interface avec le mot-clé implements
  - Implémenter une interface implique le sous-typage, le fait de pouvoir utiliser une sous-classe à tous les endroits où l'on demande l'interface.

Lorsqu'on appelle une méthode abstraite sur une interface, à l'exécution le dynamic dispatch regarde quelle est la classe de l'objet pour appeler la bonne méthode.

Exercices

Exercice 1

On reprend l'exercice sur le magasin d'articles de sport du cours 3.

On souhaite définir un type `ShoePair` qui modélise une paire de chaussures. Chaque paire de chaussures a

- une couleur `color` de type `String`.
- une marque (`brand`) de type `String`
- une taille `size` (un `int`)
- un prix `price` (un `int`).

Les tailles devront être comprises entre 35 et 45, et le prix doit être positif ou nul. Les champs ne seront pas modifiés.

1. Écrire un record `ShoePair` avec le constructeur compact qui teste les pré-conditions.

Exercice 2

Le magasin d'articles de sport doit pouvoir maintenant contenir des vêtements et des chaussures de sport.
On aura besoin d'un type commun pour les articles de sport, `Sportswear`.

1. Écrire l'interface `Sportwear` et changer les records pour qu'ils implémentent l'interface. Dans un premier temps, on ne met aucune méthode dans l'interface.
2. Modifier la classe `SportsShop` pour qu'elle contienne une liste de `Sportswear`.
3. Modifier la méthode `add`.

Exercice 3

1. La méthode `totalPrice` marche-t-elle ? Pourquoi ?

   Faites les modifications nécessaires pour qu'elle marche.

Exercice 4

1. Modifier la méthode `onSale` de `SportsShop` pour qu'elle renvoie une liste des articles soldés.

   Les articles soldés sont
   - les vêtements dont la taille est supérieure ou égale à 3.
   - les chaussures de pointures inférieures ou égales à 36 ou supérieures ou égales à 44.

Exercice 5

Le code suivant devra fonctionner :

```
static void main() {
  var polo = new Clothing("polo", "Devred", 2, 40);
  IO.println(polo);
  var polo2 = new Clothing("popo", "Devred", 40);
  IO.println(polo2);
  var shirt1 = new Clothing("shirt", "Burton", 4, 50);
  var shirt2 = new Clothing("shirt", "Burton", 4, 50);
  var shop1 = new SportsShop("Italie2");
  shop1.add(polo);
  shop1.add(shirt1);
  shop1.add(shirt2);
  var shop2 = new SportsShop("Jaude");
  shop2.add(shirt2);
  shop2.add(polo);
  shop1.add(new ShoePair("red", "Nike", 40, 300));
  IO.println(shop1);
  IO.println(shop2);
  IO.println(shop1.totalPrice());
  IO.println(shop1.onSale());
  // IO.println(SportsShop.sameItems(shop1, shop2));
 }
```

Exercice 5

Sortie attendue

```
Clothing[category=polo, brand=Devred, size=2, price=40]
Clothing[category=popo, brand=Devred, size=1, price=40]
Italie2
Clothing[category=polo, brand=Devred, size=2, price=40]
Clothing[category=shirt, brand=Burton, size=4, price=50]
Clothing[category=shirt, brand=Burton, size=4, price=50]
ShoePair[color=red, brand=Nike, size=40, price=300]
Jaude
Clothing[category=shirt, brand=Burton, size=4, price=50]
Clothing[category=polo, brand=Devred, size=2, price=40]
440
[Clothing[category=shirt, brand=Burton, size=4, price=50],
 Clothing[category=shirt, brand=Burton, size=4, price=50]]
```