

ESIEE. Programmation orientée objet. Cours 5

Marie-Pierre Béal (Cours de Rémi Forax)

Interfaces. Polymorphisme.



Interface

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.

... mieux encore

Le code est extensible, on peut ajouter des nouveaux sous-types sans changer le code de `sumAllTaxes()`

```
public record SpaceShuttle(int passengers) implements Vehicle {  
    @Override  
    public long computeTax() {  
        return passengers * 5_000_000;  
    }  
}
```

```
List<Vehicle> vehicles =  
    List.of(new Car(5), new SpaceShuttle(7));  
IO.println(sumAllTaxes(vehicles));
```

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.

Polymorphisme par sous-typage (un poil de théorie)

Polymorphisme par sous-typage (un poil de théorie)

Polymorphisme

C'est la responsabilité de l'objet de sélectionner le code à exécuter

- Si l'on veut afficher un objet, on demande à l'objet de s'afficher

C'est pour cela que la notation '.' (dot) a été inventée

- Au lieu de `f(o)` on écrit `o.f()` car on demande à l'objet de sélectionner le code à exécuter

Langages typés dynamiquement

JavaScript

```
class A {  
  constructor(value) {  
    this.value = value;  
  }  
  m() { // implémentation  
    return this.value;  
  }  
}  
  
class B {  
  m() { // implémentation  
    return 13;  
  }  
}  
  
...  
let i = (cond)?  
new A(42): new B();  
print(i.m()); // Dynamic dispatch
```

Python

```
class A:  
    def __init__(self, value):  
        self.value = value  
    def m(self):  
        // implémentation  
        return self.value  
  
class B:  
    def m(self):  
        // implémentation  
        return 13  
  
...  
i = new A(42) if (cond) else new B()  
print(i.m()) // Dynamic dispatch
```

Langages typés statiquement

Go

```
type I interface { // Type abstrait
    M() int
}
type A struct {
    value int
}
type B struct {}
func (a A) M() int { // implém.
    return a.value
}
func (b B) M() int { // implém.
    return 13
}
...
var i I
if cond {
    i = A { value: 12 }
} else {
    i = B {}
}
fmt.Println(i.M()) // Dyn. dispatch
```

Rust

```
trait I { // Type abstrait
    fn m(&self) -> i32;
}
struct A {
    value: i32
}
struct B {}
impl I for A { // implémentation
    fn m(&self) -> i32 {
        self.value
    }
}
impl I for B { // implémentation
    fn m(&self) -> i32 {
        13
    }
}
...
let i: &dyn I = if cond
{ &A { value: 42 } } else { &B{} };
println!("{}", i.m()); // Dyn. dispatch
```

Sous-typage

Dans les langages statiquement typés, pour pouvoir faire le polymorphisme, on a besoin

- d'un type abstrait déclarant les méthodes abstraites
- de pouvoir voir un type plus spécifique (sous-type) comme un type moins spécifique (super type)

Le fait de pouvoir assigner/passer en paramètre un type plus spécifique est appelé sous-typage

```
I i = new A(42);  
I i = new B();
```

Principe de substitution de Liskov

Barbara Liskov (https://fr.wikipedia.org/wiki/Barbara_Liskov)
1988 :

*Let $\Phi(x)$ be a property provable about objects x of type T .
Then $\Phi(y)$ should be true for objects y of type S where S is a
subtype of T .*

Avec Φ méthode appellable

- Si on peut appeler une méthode m sur T , alors on peut appeler m sur S si S est un sous-type T

Interface et principe de Liskov

En Java, pour qu'une classe "implements" une interface, elle doit fournir un code pour toutes les méthodes abstraites.

- Donc toutes les méthodes que l'on appelle sur l'interface sont appelables sur la classe
 - La classe est un sous-type de l'interface
 - On peut utiliser la classe A à tous les endroits où l'on attend l'interface I

A l'exécution, on utilisera le dynamic dispatch pour qu'un appel `i.m()` appelle le code de `m` de la classe `A`.

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

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);  
    }  
}
```

Type scellé (sealed)

Type scellé (sealed)

Interface Open / Close

Du point de vue d'une librairie, on a deux sortes d'interfaces

- Les interfaces ouvertes où l'utilisateur de la librairie va fournir une implantation
 - Une interface qui réagit au clic de souris
- Les interfaces fermées où l'on connaît toutes les implantations
 - On peut payer par chèque, espèces ou carte bleue

Par défaut les interfaces sont ouvertes mais on peut les fermer.

Interface scellée

Le mot clé **sealed** indique que l'on connaît tous les sous-types d'une interface et permits liste ces sous-types.

```
public sealed interface Download permits UriDownload,  
                                         FileDownload {  
}
```

```
public record UriDownload(URI uri) implements Download {  
}
```

```
public final class FileDownload implements Download {  
}
```

Dans le cas où le sous type est une classe, il faut qu'elle soit marquée **final**, on verra plus tard pourquoi (les record sont final par défaut).

Switch sur les types

On peut faire un switch sur des objets

```
Download download = new UriDownload(  
    new URI("http://www.example.com/"));  
switch(download) {  
    case UriDownload d -> ... ;  
    case FileDownload d -> ... ;  
}
```

Il n'y a pas besoin de **default** car tous les cas sont couverts.

Sum Type et Product Type

Dans les langages fonctionnels comme Caml ou F#, on a deux sortes de types

- Les types produit : $P = A \times B \times C$
- Les types somme : $S = A \mid B \mid C$

En Java,

- Un record est un type produit
- Une interface scellée est un type somme

Membre d'une interface

Membre d'une interface

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  
}
```

main() dans une interface

Comme une interface peut avoir des méthodes static, elle peut déclarer un main().

```
public interface HelloMain {  
    static void main() {  
        IO.println("hello from an interface");  
    }  
    // Pas nécessaire de déclarer main() public,  
    // les membres d'une interface sont public par défaut  
}
```

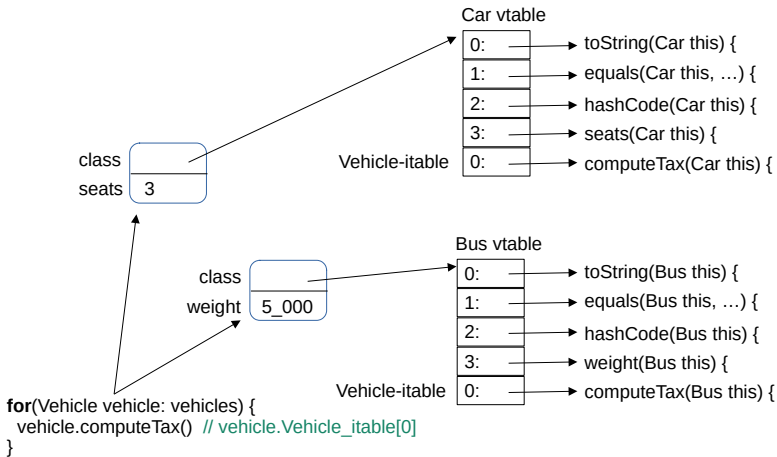
Comment ça marche en mémoire

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) { ... } }

```



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

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.