

Polymorphisme

Programmation Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1



des méthodes : méthodes de même nom avec signatures différentes
↔ cf. constructeurs

des objets : pouvoir exploiter une “facette” d’un objet indépendamment des autres

- ▶ “multi-typage” des objets
- ▶ permettre une “*projection*” de l’objet sur un type
- ▶ aspect orienté objet non présent dans prog. modulaire

Polymorphisme des méthodes

▶ possibilité d’avoir dans la définition d’une même classe des méthodes de **même nom** mais de **signatures différentes**
↔ variation du nombre et/ou du type/classe des arguments

```
public void someMethod(int i) {...}
public void someMethod(int i, String name) {...}
public void someMethod(String name, int i) {...}
public void someMethod(Livre l) {...}
```

Usage possible : valeur par défaut des paramètres

```
public void someMethod(int i, String name) {
    ... traitement de someMethod
}
public void someMethod(int i) {
    this.someMethod(i, "valeur par défaut"); // invoque la méthode ci-dessus
}
```

▶ cas des constructeurs : un autre usage de this

```
public class AClass {
    public AClass(int i, String name) {
        ... gestion de la construction d'une instance
    }
    public AClass(String name) {
        this(12, name); // appel du constructeur ci-dessus
    }
}
```

Pas sur les valeurs de retour

▶ Polymorphisme des valeurs de retour interdit (refusé à la compilation)

```
public int someMethod(String name) {...}
public String someMethod(String name) {...} // interdit !!!

public String someMethod(String name, int i) {...}
// autorisé car args différents
```

▶ problème d’ambiguïté

```
String stringRes = anObject.someMethod("abcdef"); // pas d'ambiguïté
int intRes = anObject.someMethod("abcdef"); // ici

anObject.someMethod("abcdef"); // mais dans ce cas, si !
// quelle méthode invoquée ?
```

Polymorphisme des objets

première approche par les interfaces JAVA

▶ Un des “problèmes” principaux :
↔ Quels messages sont acceptés par les objets ?
↔ Quelles interfaces publiques pour leurs classes ?

Abstraction

▶ Des objets de natures très différentes peuvent parfois subir les mêmes traitements
↔ càd répondre aux mêmes envois de messages
↔ càd avoir une **partie** de leur interface commune

▶ Pouvoir les traiter globalement en ne considérant que leur interface commune
↔ projection de leur interface sur la partie commune
↔ projection sur une restriction de leur type.

Exemple

Vie quotidienne

- ▶ Papiers, bouteilles, piles électriques, Cageots, etc. sont des objets **différents**, ayant des comportements **différents**
 ↳ *déchirer* du papier, *remplir* une bouteille
- ▶ mais sont tous *recyclables*
 ↳ tous peuvent être recyclés (même si processus **différents**)
 On peut : “recycler tous les objets d’une poubelle”

Programmation

- ▶ Paper, Bottle, Battery, Crate, etc sont des classes d’objets **différentes**, elles proposent donc des fonctionnalités (méthodes) **différentes**
 ↳ *tear()* pour Paper, *empty()* pour Bottle
- ▶ **mais** elles proposent toutes *recycle()*
 ↳ avec réponse **adaptée** à chacune

- ▶ Peut on, et comment, programmer :

“recycler tous les objets d’une poubelle”

```
for (int i = 0; i < trashcan.length; i++) {
    trashcan[i].recycle();
}
```

- ▶ Quelle est la définition du tableau `trashcan` ?
- ▶ Quel est le type `T` de ses éléments ?

`T[] trashcan`

Seule information :

`T` accepte le message `recycle()`

et on veut pouvoir mettre un papier et une bouteille dans la poubelle...

Pistes ?

1 Conserver les classes différentes

on veut mettre des papiers donc `T = Paper...`
 Alos on a : `Paper[] trashcan = new Paper[12];`
 ↳ `trashcan[0] = new Paper();` est possible
 ↳ **mais** le compilateur refuse : `trashcan[3] = new Bottle();`

2 Créer un type commun.

On définit une classe `RecyclableObject` pour modéliser tous les objets indifférent : papiers, bouteilles, etc. : `T = RecyclableObject`
`RecyclableObject[] trashcan = new RecyclableObject[12];`
 ↳ `trashcan[0] = new RecyclableObject("paper");` (par exemple)
 ↳ **mais**

- ▶ comment dans ce cas prendre en compte les traitements de `recycle()` différents pour les papiers et les bouteilles ?
- ▶ quel sens a la méthode `empty()` pour `new RecyclableObject("paper")` ?

Solution

mixer les 2 propositions :

Conserver les classes différentes et créer un type commun

- ▶ il faut conserver les classes différenciées Paper, Bottle, etc.
- ▶ il faut pouvoir considérer leurs instances comme des objets
 “obéissant à l’envoi de message `recycle()`”
- ▶ il faut alors pouvoir traiter les objets sans les différencier par leur classe

Pouvoir considérer les objets comme étant du type
 “obéissant à l’envoi de message `recycle()`”
 Réaliser une **projection** sur ce type.

Ne considérer dans ce cas qu’une **facette** de l’objet

Solution Java : interface

- ▶ En JAVA, on appelle **interface**, un ensemble de déclarations de signatures de méthodes publiques.
- ▶ Une classe peut **implémenter** une interface, dans ce cas elle **doit** définir un comportement pour **chacune** des méthodes qui y sont définies.
- ▶ Les instances de la classe pourront alors être vues comme **étant du type de l’interface** et manipulées comme telles, et initialiser une référence de ce type “**projection**” sur le type de l’interface
- ▶ Une telle référence accepte dans ce cas **uniquement** les envois de message définis dans l’interface

Interface = type

(définit les envois de messages autorisés)

```
public interface Recyclable {
    public void recycle();
} // Recyclable

public class Paper implements Recyclable {
    ...
    public void recycle() {
        System.out.println("recyclage papier");
    } // Paper

public class Bottle implements Recyclable {
    ...
    public void recycle() {
        System.out.println("recyclage bouteille");
    } // Bottle

...
Recyclable[] trashcan = new Recyclable[2];
trashcan[0] = new Paper(); // projection des instances
trashcan[1] = new Bottle(); // sur le "type" Recyclable
for (int i = 0; i < trashcan.length; i++) {
    trashcan[i].recycle(); // message indifférencié
} // mais traitements différents
```


Cast et types numériques

tous les casts sont possibles!

- ▶ perte d'information possible ("narrowing conversion", "loss of precision"), si le type d'arrivée est "plus petit"
- ▶ exemple : pour byte, char, short, tous les opérateurs arithmétiques retournent un int, il est donc nécessaire de "caster" le résultat vers le type souhaité

```
short s1 = 12;
short s2 = 25;
short s3 = (short) s1+s2;
```

Retour sur types énumérés

les enum implémente l'interface java.lang.Comparable

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

où T représente le type des éléments à comparer

Pour le type énuméré Saison, T vaut Saison, on a donc la méthode :

```
public int compareTo(Saison o)
```

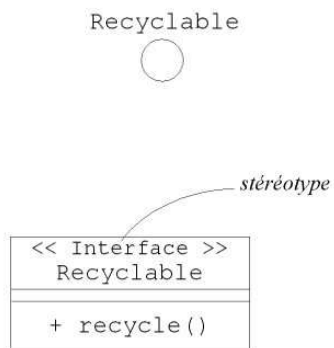
L'enum Saison correspond donc à la classe

```
public class Saison implements Comparable<Saison> {
    ... // voir cours précédent
    public int compareTo(Saison o) {
        return this.ordinal() - o.ordinal();
    }
}
```

en java ≤ 1.4, le "T" n'existe pas et doit être remplacé par Object :

```
public class Saison implements Comparable {
    ... // voir cours précédent
    // pour satisfaire l'interface comparable
    public int compareTo(Object o) {
        if (o instanceof Saison) {
            Saison lAutre = (Saison) o;
            return this.ordinal() - lAutre.ordinal();
        } else {
            throw new ClassCastException("argument should be a Saison"); // voir plus tard
        }
    }
} // Saison
```

Notations UML : interface



UML : implémentation

