

Polymorphisme

Programmation Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1



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
```

polymorphisme des objets

posons le problème...

Notion clef

Où l'on aborde ce qui fait que les langages objet diffèrent des langages impératifs...

première approche par les interfaces JAVA

- peut on, et comment, programmer :

"recycler tous les objets d'une poubelle"

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

Problème

quelle est la définition du tableau trashcan ?
ou : quel est le type T de ses éléments ?

T[] trashcan

Informations :

- T accepte le message recycle()
- on veut pouvoir mettre à la fois un papier et une bouteille dans la poubelle...

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

Pistes ?

A vous...

Quelles sont les possibilités ?
Quels avantages/inconvénients ont-elles ?

- 1 Conserver les classes **différentes**
- 2 Créer un type **commun**

solution objet

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 traiter les objets sans les différencier par leur classe
- il faut pouvoir considérer leurs instances comme des objets du type
"accepte l'envoi de message `recycle()`"

Ne considérer dans ce cas qu'une **facette** de l'objet.
Réaliser une **projection** sur ce type.

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
- 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("recycling paper");
    }
} // Paper
```

```
public class Bottle implements Recyclable {
    ...
    public void recycle() {
        System.out.println("recycling bottle");
    }
} // 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

+---trace-----
| recycling paper
| recycling bottle
+-----
```

Important

La référence n'est pas l'objet !

Le type de la référence définit les envois de message autorisés.

La classe de l'objet définit le traitement exécuté.

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

public class Paper implements Recyclable {
    public void dechire() { ... }
    public void recycle() {
        System.out.println("recycling paper");
    }
} // Paper

...
Paper p = new Paper();
p.dechire();           // ok
p.recycle();           // ok
Recyclable r = p;      // ok : p est aussi de type Recyclable : 2 références sur le même objet
r.recycle();           // ok : code de recycle dans classe Paper exécuté
r.dechire();           // NON, envoi de message interdit sur type Recyclable
```

- on peut utiliser une interface là où on utilise une classe (sauf création d'instances)
 ↪ tous les deux sont des "types"

```
public void aMethod(Recyclable r) {
    ... traitement en n'invoquant sur r que des méthodes de Recyclable
}
...
Recyclable aRecyclableObject = new Paper();
someObject.aMethod(aRecyclableObject);
someObject.aMethod(new Bottle()); // projection implicite sur l'interface
```

- une classe peut implémenter plusieurs interfaces, elle doit dans ce cas fournir un comportement pour chacune des méthodes de chaque interface.

```
public interface Flammable {
    public void burn();
} // Flammable
...
public class Paper implements Recyclable, Flammable {
    public void recycle() { ... traitement ... }
    public void burn() { ... traitement ... }
} // Paper
```

```
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(); // traitement indifférencié
}
```

+-----+
 | recycling paper
 | recycling bottle
 +-----+

Comment la « bonne » méthode est-elle appelée ?

Late-binding et early binding

- early binding** (C) compilateur génère un appel à une fonction en particulier et le code appelé est précisément déterminé **à la compilation** lors de l'édition de liens
- late binding** (POO) le code appelé lors de l'envoi d'un message à un objet n'est déterminé qu'**au moment de l'exécution** (run time)
 ↪ le compilateur ne vérifie « que » l'acceptation du message et la validité des types d'arguments et de retour.

Late-binding

Mécanisme fondamental de la programmation objet

```
public class RecyclingUnit {
    /** applies the recycling process to r
     * @param r the object to be recycled
     */
    public void doIt(Recyclable obj) {
        obj.recycle();
    }
}
```

?

quelle méthode est invoquée lors de l'exécution de
`obj.recycle()`

Late-binding

Mécanisme fondamental de la programmation objet

```
public class RecyclingUnit {
    /** applies the recycling process to r
     * @param r the object to be recycled */
    public void doIt(Recyclable obj) {
        obj.recycle();
    }
}

public class RecyclingUnitMain {
    public static void main(String[] args) {
        RecyclingUnit recycler = new RecyclingUnit();
        Recyclable ref;
        if (args[0].equals("paper")) {
            ref = new Paper();
        }
        else { ref = new Bottle(); }
        recycler.doIt(ref);
    }
}
```

Ce code compile,
résultat dépend de `args[0]`.

```
...> java RecyclingUnit paper
recyclage papier
...> java RecyclingUnit other
recyclage bottle
```

méthode recycle in-
voquée non connue a
priori.

Objets polymorphes

- Les objets sont instances d'une classe et donc du type de cette classe... mais sont **aussi** du type de chacune des interfaces implémentées par la classe.
- différents points de vue possible sur un même objet
un objet présente différentes facettes

interface = contrat à respecter

Cast (= “fondre/mouler”)

Transtypage

Une référence n'a qu'un seul type, un objet peut en avoir plusieurs.
“caster”/transtyper : à partir d'une référence sur un objet, en créer **une autre** d'un **autre type**, vers le **même objet**.

UpCast changer vers une classe moins spécifique (toujours possible vers `Object`) : **généralisation**

- naturel et implicite,
- vérifié à la compilation,
- “safe”.

DownCast changer vers une classe plus spécifique **spécialisation**

- explicite,
- vérifié à l'exécution,
- à risque.

Illustration

```
Recyclable[] trashcan = new Recyclable[2];
trashcan[0] = new Paper();           // upcast      (Paper → Recyclable)
trashcan[1] = new Bottle();          // implicite   (Paper → Recyclable)
for (int i = 0; i < trashcan.length; i++) {
    ((Paper) trashcan[i]).burn();     // downcast explicite
}                                     //             (Recyclable → Paper)
                                     // erreur à l'exécution qd i=1
```

Notations UML : interface

