

Polymorphisme

Programmation Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1



Université
Lille1
Sciences et Technologies

IEEA - FIL
Informatique

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

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
- ▶ mais sont tous *recyclables*
↪ on peut tous leur appliquer le traitement de *recyclage* (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, c-à-d ayant des interfaces différentes
- ▶ toutes leurs instances répondent à l’envoi de message `recycle()` (réponse adaptée à chacune)

- Peut on, et comment, coder :

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

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

Propositions ?

- 1 Si on a : `Paper[] trashcan = new Paper[12];`
↪ alors le compilateur refuse : `trashcan[3] = new Battery();`
- 2 Définir une classe `RecyclableObject` pour définir tous les objets ?
`RecyclableObject[] trashcan = new RecyclableObject[12];`
↪ comment dans ce cas prendre en compte les traitements de `recycle()` différenciés de `Paper` et `Bottle`.

Solution

mixer les 2 propositions !

- ▶ 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

Projeter les objets sur le type
“*obéissant à l’envoi de message `recycle()`*”

ne considérer 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/
- ▶ Une telle référence n'accepte dans ce cas **que les** envois de message acceptés/définis dans l'interface

Interface = type

(car définit des envois de messages)

Solution JAVA : interface

```

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 traitement différent

```

```

+---trace-----
| recyclage papier
| recyclage bouteille
+-----

```

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("recyclage papier");
    }
} // 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
```

Solution JAVA : interface

- ▶ on peut faire avec une interface tout ce que l'on peut faire avec une classe (sauf création d'instances)

↪ tous les deux des “types”

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

- ▶ une classe peut implémenter plusieurs interfaces à la fois, elle doit dans ce cas implémenter 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
```

NB : attention au risque de conflits entre interfaces

(↪ appliquer mêmes règles que pour le polymorphisme de méthodes)

Comment ça marche ?

Comment ça marche ?

```

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
}

```

+-----
 | recyclage papier
 | recyclage bouteille
 +-----

Comment la “bonne” méthode est-elle appelée ?

Late-binding et early binding

early binding compilateur génère un appel à une fonction en particulier et le code appelé est précisément déterminé 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.

Objets polymorphes

- ▶ Les objets peuvent être vus/considérés comme des instances de leur classe... mais aussi comme étant du type de chacune des interfaces implémentées par la classe.
- ▶ objets peuvent être considérés selon leurs différentes facettes `Object`

interface = contrat à respecter

par des objets afin d'en permettre une exploitation quelle que soit leur classe

- ▶ NB : tous les objets peuvent être vus selon la facette de la classe

Cast (= “fondre/mouler”)

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.

Cast (= “fondre/mouler”)

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

```

Cast et types primitifs

tous les cast sont possibles !
(sauf avec `boolean`)

- ▶ perte d'information possible (narrowing conversion), si le type d'arrivée est “plus petit”
- ▶ 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

Recyclable



stéréotype



UML : implémentation

