

# Programmation 2 : troisième cours

Arnaud Labourel [arnaud.labourel@univ-amu.fr](mailto:arnaud.labourel@univ-amu.fr)

15 octobre 2019



# Interfaces

# Problèmes de recyclage

- Papiers, bouteilles, piles électriques, cageots, ... sont des objets **différents** :
    - ⇒ *déchirer* du papier, *remplir* un bouteille
  - Mais ces objets partagent tous la propriété d'être **recyclable**
    - ⇒ tous peuvent être *recyclés* (même si le processus peut varier)
- On peut *recycler* tous les objets d'une poubelle.

## En programmation objet

- Paper, Bottle, Battery, Crate, ... sont des classes d'objets **différentes**
- Toutes ont un méthode `recycle()` avec une implémentation **adaptée** à chacune.

# Comment recycler tous les objets d'une poubelles

Avec le code suivant ?

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

## Problème

Comment définir le tableau Trashcan ?

Quel est le type T de ces éléments ?

## Remarques

- les objets de type T implémentent la méthode recycle
- Trashcan doit pouvoir contenir des objets de types différents

Conserver les classes différentes et créer un type commun.

- on doit conserver les classes différenciées : Paper, Bottle, ...
- on doit traiter les objets sans les différencier par leur classe.
- il faut pouvoir considérer les instances des classes comme des objets de type implémente la méthode recycle.

## Projection

On va projeter les objets sur un type commun qui ne gardera que la partie commune des fonctionnalités. On ne considère qu'une facette de l'objet.

# Solution java : interface

- En java, une interface est un ensemble de déclaration de signatures de méthodes et définit un type.
- Une classe peut **implémenter** une interface et doit dans ce cas définir le comportement (code) pour **chacune** des méthodes de l'interface.
- les instances d'une classe pourront être vues comme étant du **type de l'interface**, manipulées comme telles et avoir leur référence stockée dans une variable du type de l'interface.
- Un référence du type d'une interface accepte uniquement les appels de méthodes définies dans l'interface.

## Exemple d'utilisation (1/2)

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

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

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

## Exemple d'utilisation (2/2)

```
Recyclable[] trashcan = new Recyclable[2];

trashcan[0] = new Paper();
trashcan[1] = new Bottle();

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

# Interface pour l'affichage

Supposons que des classes implémentent un service de façons différentes :

```
class SimplePrinter {  
    void print(String document){  
        System.out.println(document);  
    }  
}
```

```
class BracePrinter {  
    void print(String document){  
        System.out.println("{ " + document + " }");  
    }  
}
```

Les instances de ces deux classes possèdent une méthode `print` avec la même signature (types des arguments et du retour).

# Code facilement modifiable

Nous souhaiterions pouvoir facilement passer du code suivant :

```
Printer printer = new SimplePrinter();  
printer.print("truc"); // → truc
```

au code suivant :

```
Printer printer = new BracePrinter();  
printer.print("truc"); // → {truc}
```

Il nous faudrait définir un type `Printer` qui oblige la variable à contenir des références vers des objets qui implémentent la méthode `print`.

⇒ définition d'une interface `Printer`.

# Abstraction

On peut vouloir traiter les objets en utilisant les services qu'ils partagent :

```
for (int index = 0; index < printers.length; index++)  
    printers[index].print(document);
```

On peut aussi vouloir écrire un programme en supposant que les objets manipulés implémentent certains services (comme le fait de pouvoir les comparer) :

```
boolean isSorted(Comparable[] array) {  
    for (int i = 0; i < array.length - 1; i++)  
        if (array[i].compareTo(array[i+1]) > 0)  
            return false;  
    return true;  
}
```

# Description d'une interface

Description d'une interface en Java :

```
public interface Printer{  
    /**  
     * Affiche la chaîne de caractères document.  
     * @param document la chaîne à afficher  
     */  
    public void print(String document);  
}
```

Une interface :

- définit la signature (types des arguments et du retour) d'une ou plusieurs méthodes
- est un contrat
- définit un type : référence vers un objet implémentant les méthodes de l'interface

# Implémentation d'une interface

Le mot-clé `implements` permet d'indiquer qu'une classe implémente un interface :

```
class SimplePrinter implements Printer {  
    void print(String document){  
        System.out.println(document);  
    }  
}
```

```
class BracePrinter implements Printer {  
    void print(String document){  
        System.out.println("{ " + document + " }");  
    }  
}
```

Java vérifie à la compilation que toutes les méthodes de l'interface sont implémentées.

# Références et interfaces

Déclaration d'une variable de type référence vers une instance d'une classe qui implémente l'interface `Printer` :

```
Printer printer;
```

## Important

Une interface ne définit pas de constructeurs.

Interdit : `printer = new Printer()`

## Compatibilité classe et instance

Pour affecter une référence à une variable d'un type défini par une interface, on doit instancier une classe implémentant l'interface.

# Références et interfaces

Il est donc possible d'instancier une classe implémentant l'interface,

```
SimplePrinter simplePrinter = new SimplePrinter();
```

puis de stocker la référence de l'objet dans une variable de type de l'interface :

```
Printer printer1 = simplePrinter;
```

On parle alors d'**upcasting** (transtypage vers le haut). On peut aussi directement mettre un tel objet sans passer par une variable intermédiaire :

```
Printer printer2 = new BracePrinter();
```

Par contre, cela ne fonctionne pas dans le cas où la classe n'implémente pas l'interface :

```
Printer printer3 = new String("Hello!"); //interdit
```

# Références et interfaces

```
class Utils {
    static void printString(Printer[] printers,
                            String doc) {
        for (int i = 0; i < printers.length; i++)
            printers[i].print(doc);
    }

    static void printArray(String[] array,
                            Printer printer) {
        for (int i = 0; i < array.length; i++)
            printer.print(array[i]);
    }
}
```

L'existence des méthodes est vérifiée à la compilation.

Le code suivant ne compilera pas car l'interface `Printer` n'a pas de méthode `println` :

```
Printer printer = new SimplePrinter();  
printer.println("Hello!"); // Impossible
```

## Définition polymorphisme

Du grec ancien *polús* (plusieurs) et *morphê* (forme), concept consistant à fournir une interface unique à des entités pouvant avoir différents types.

# Polymorphisme

Le choix de la méthode à exécuter ne peut être fait qu'à l'exécution :

```
Printer[] printers = new Printer[2];
printers[0] = new SimplePrinter();
printers[1] = new BracePrinter();
Random random = new Random(); // générateur aléatoire
int index = random.nextInt(2); // 0 et 1
printers[index].print("mon message");
```

L'affichage dépend du tirage aléatoire pour index :

- index=0 → méthode de la classe SimplePrinter → mon message
- index=1 → méthode de la classe BracePrinter → {mon message}

# Résumé des interfaces

- Une interface est un ensemble de signatures de méthodes.
- Une classe peut implémenter une interface : elle doit préciser le comportement de chacune des méthodes de l'interface.
- Il est possible de déclarer une variable pouvant contenir des références vers des instances de classes qui implémentent l'interface.
- Java vérifie à la compilation que toutes les affectations et les appels de méthodes sont corrects.
- Le choix du code qui va être exécuté est décidé à l'exécution (en fonction de l'instance pointée par la référence).

# Exemple d'interfaces en Java

- `Comparable<T>` : objets qu'on peut comparer à des objets de type `T`.
- `List<T>` : liste d'objets de type `T`.
- `Stack<E>` : pile d'objet de type `T`.
- `Iterable<T>` : collection d'objet de type `T` qu'on peut parcourir avec un boucle.

## Types paramétrés/générique

Types dont la définition contient un autre type.

# Définition de Comparable<T>

```
public interface Comparable<T>{  
    /**  
     * Compares this object with the specified object for  
     * order. Returns a negative integer, zero, or a  
     * positive integer as this object is less than,  
     * equal to, or greater than the specified object.  
     * @param other the objet to be compared  
     * @return a negative integer, zero, or a positive  
     * integer as this object is less than, equal to, or  
     * greater than the specified object.  
     */  
    int compareTo(T other);  
}
```

# Utilisation de Comparable<T>

```
// Utilisation typique
```

```
public class MyOrderedClass
    implements Comparable<MyOrderedClass>{

    int compareTo(MyOrderedClass other){
        // ...
    }
}
```

Sert à définir une relation d'ordre entre les objets d'une classe (par exemple pour les trier).

# Utilisation de Comparable<T> pour la classe Student

```
public class Student
    implements Comparable<Student>{

    public final String lastName;
    public final int idNumber;
    //...
    public int compareTo(Student other){
        return lastName.compareTo(other.lastName);
    }
}

List<Student> students; //...
Collections.sort(students);
// Tri par nom de famille
```

# Utilisation de Comparable<T> pour la classe Student

```
public class Student
    implements Comparable<Student>{

    public final String lastName;
    public final int idNumber;
    //...
    public int compareTo(Student other){
        return idNumber - other.idNumber;
    }
}

Student[] students; //...
Arrays.sort(students);
// Tri par numéro d'étudiant
```

# Interface Iterable

```
public interface Iterable<T>{  
    Iterator<T> iterator();  
    void forEach(Consumer <? super T> action);  
    Spliterator<T> spliterator();  
}
```

- Définition complexe
- Utilisation facile et utile

# Utilisation d'Iterable

**Utilisation** : si une classe implémente `Iterable<T>`, ces instances contiennent une collection d'objets de type `T`.

On peut parcourir les objets de la collection à l'aide d'une boucle `for`.

```
public class Order{
    Iterable<Items> items;
    //...
    public void printAllItems(){
        for (Item item : this.items){
            System.out.println(item.toString());
        }
    }
}
```

# Interface Collection

```
public interface Collection<T> extends Iterable<T>{  
    boolean add(T element);  
    boolean contains(T element);  
    boolean isEmpty();  
    boolean remove(Object o);  
    //...  
}
```

## Mot-clé extends

Quand une interface “fille” étend une interface “mère”, elle hérite de toutes les méthodes de sa “mère”.

Une classe implémentant la classe “fille” doit donc définir les méthodes des deux interfaces.

# Interfaces étendant Collection

Il existent de nombreuses interfaces qui sont une extension de l'interface `Collection` :

- `Set` : collection d'objet dans laquelle chaque objet ne peut apparaitre qu'une fois. Implémenté par `HashSet` et `TreeSet`.
- `List` : séquence d'éléments. Implémenté par `ArrayList` et `LinkedList`.
- `Deque` (**d**ouble **e**nded **q**ueue) : séquence d'éléments avec accès qu'au début et à la fin (file d'attente). Implémenté par `ArrayDeque` et `LinkedList`.
- ...

# Implémentations multiples

Il peut être utile d'avoir une classe implémentant plusieurs interfaces.

Par exemple, une classe Modem pourrait implémenter les deux interfaces suivantes :

```
public interface ConnexionManager
{
    public void dial(string phoneNumber);
    public void hangUp();
}
```

```
public interface TransmissionManager
{
    public void send(char c);
    public char receive();
}
```

# Implémentations multiples

Une classe `Printable` avec une méthode `print` qui permet d'afficher l'objet.

```
interface Printable {  
    public void print();  
}
```

Une classe `Stack` avec deux méthodes :

- `push` qui permet d'empiler un entier.
- `pop` dépile et retourne l'entier en haut de la pile.

```
interface Stack {  
    public void push(int value);  
    public int pop();  
}
```

# Implémentations multiples

Implémentation des deux interfaces précédentes :

```
public class PrintableArrayStack
    implements Stack, Printable {
    private int[] array; private int size;
    public PrintableArrayStack(int capacity) {
        array = new int[capacity]; size = 0;
    }
    public void push(int v) { array[size] = v; size++; }
    public int pop() { size--; return array[size]; }
    public void print() {
        for (int i = 0; i < size; i++)
            System.out.print(array[i]+" ");
        System.out.println();
    }
}
```

# Implémentations multiples

Implémentation d'une des deux interfaces :

```
public class PrintableString implements Printable {
    private String string;
    public PrintableString(String string) {
        this.string = string;
    }
    public void print() {
        System.out.println(string);
    }
}
```

# Implémentations multiples

Exemple d'utilisation des classes précédentes :

```
Printable[] printables = new Printable[3];
printables[0] = new PrintableString("bonjour");
PrintableArrayStack stack = new PrintableArrayStack(10);
printables[1] = stack;
printables[2] = new PrintableString("salut");
stack.push(10);
stack.push(30); System.out.println(stack.pop());
stack.push(12);
for (int i = 0; i < printables.length; i++)
    printables[i].print();
```

Qu'écrit ce programme sur la sortie standard ?

# Vérification des types

Vérification des types à la compilation :

```
Stack[] arrayStack = new Stack[2];  
arrayStack[0] = new PrintableStack();  
arrayStack[1] = new PrintableString("t"); // Erreur !
```

PrintableString n'implémente pas Stack.

```
Stack stack = new PrintableStack();  
Printable printable = stack; // Erreur !
```

Le type Stack n'est pas compatible avec le type Printable.