Types paramétrés et interfaces

Arnaud Labourel arnaud.labourel@univ-amu.fr

Section 1

Types paramétrés

Stack d'Object

Supposons que nous ayons la classe suivante :

```
public class Stack {
  private Object[] stack = new Object[100];
  private int size = 0;

  public void push(Object object) {
    stack[size] = object;
    size++;
  }
  public Object pop() {
    size--;
    Object object = stack[size];
    stack[size] = null; // Pour le Garbage Collector.
    return object;
  }
```

Problème de Stack d'Object

Nous rencontrons le problème suivant :

```
Stack stack = new Stack();
String string = "truc";
stack.push(string);
string = (String) stack.pop();
// Transtypage obligatoire !
```

Nous avons également le problème suivant :

```
Stack stack = new Stack();
Integer intValue = new Integer(2);
stack.push(intValue);
String string = (String) stack.pop();
// Erreur à l'exécution
```

La solution : types paramétrés

Par conséquent, on souhaiterait pouvoir préciser le type des éléments :

```
Stack<String> stack = new Stack<String>();
String string = "truc";
stack.push(string); // Le paramètre doit être un String.
String string = stack.pop(); // retourne un String.
```

Java nous permet de définir une classe `Stack` qui prend en paramètre un type. Ce type paramétré va pouvoir être utilisé dans les signatures des méthodes et lors de la définition des champs de la classe.

Lors de la compilation, Java va utiliser le type paramétré pour effectuer :

- des vérifications de type ;
- des transtypages automatiques ;
- des opérations d'emballage ou de déballage de valeurs.

Définition de classes paramétrées

La nouvelle version de la classe Stack :

```
public class Stack<T> {
    private Object[] stack = new Object[100];
    private int size = 0;
    public void push(T element) {
        stack[size] = element;
        size++;
    }
    public T pop() {
        size--;
        T element = (T) stack[size];
        stack[size] = null;
        return element;
    }
}
```

Emballage et déballage

Les types primitifs ne sont pas des classes :

Dans le cas d'un `int`, on doit utiliser la classe d'emballage (*wrapper class*) 'Integer" :

Interdit : ~~`Stack<int> stack = new Stack<>();`~~

Autorisé :

```
Stack<Integer> stack = new Stack<>();
int intValue = 2;
Integer integer = Integer.valueOf(intValue);
// → emballage du int dans un Integer.
stack.push(integer);
Integer otherInteger = stack.pop();
int otherIntValue = otherInteger.intValue();
// → déballage du int présent dans le Integer.
```

Types primitifs

type	classe d'emballage	taille	valeurs possibles
byte	Byte	8 bits	-128 à 127
short	Short	16 bits	-32768 à 32767
int	Integer	32 bits	-2^{31} à $2^{31}-1$
long	Long	64 bits	-2^{63} à $2^{63}-1$
float	Float	32 bits	
double	Double	64 bits	
char	Character	16 bits	caractère unicode
boolean	Boolean	non définie	`false` ou `true`

Classes d'emballage

La classe `Number` sert de base pour toutes les classes d'emballage.

Elle contient les méthodes suivantes :

- `public int intValue()`
- `public long longValue()`
- `public float floatValue()`
- `public double doubleValue()`
- `public byte byteValue()`
- `public short shortValue()`

Classes d'emballage

Les classes d'emballage étendent `Number` :

- `Byte` → `public static Byte valueOf(byte b)`
- `Short` → `public static Short valueOf(short s)`
- `Integer` → `public static Integer valueOf(int i)`
- `Long` → `public static Long valueOf(long l)`
- `Byte` → `public static Byte valueOf(byte b)`

Ils existent des constructeurs mais ils sont dépréciés (et donc pas à utiliser).

La classe Character

Les classes d'emballage ne contiennent pas que des méthodes liées aux instances :

- `public static Byte valueOf(byte b)`
- `public static char charValue()`
- `public static boolean isLowerCase(char ch)`
- `public static boolean isUpperCase(char ch)`
- `public static boolean isDigit(char ch)`
- `public static boolean isLetter(char ch)`
- `public static boolean isLetterOrDigit(char ch)`
- `public static char toLowerCase(char ch)`
- `public static char toUpperCase(char ch)`
- `public static char toTitleCase(char ch)`

Emballage et déballage automatique

Depuis Java 5, il existe l'emballage et le déballage automatique :

```
Stack<Integer> stack = new Stack<Integer>();
int intValue = 2;
stack.push(intValue);
// → emballage automatique du int dans un Integer.
int otherIntValue = stack.pop();
// → déballage automatique du int.
```

Attention

Il est important de noter que des allocations sont effectuées lors des emballages sans que des `new` soient présents dans le code.

Plusieurs paramètres de types

```
public class Pair<A, B> {
    public final A first;
    public final B second;
    public Pair(A first, B second) {
        this.first = first;
        this.second = second;
    }
    public String toString() {
        return "(" + first + ", " + second + ')';
    }
}
```

Utilisation:

```
Pair<String,Integer> pair = new Pair<>("toto", 12);
System.out.println(pair); // affiche (toto, 12)
```

Méthodes paramétrées

Il est possible de mettre de définir des paramètres de types pour une méthode.

Il faut les déclarer avant le type de retour.

```
public class Pair<A, B> {
  public static <C, D> Pair<C,D>
      makePair(C first, D second) {
    return new Pair<C,D>(first, second);
  }
}
```

Utilisation:

```
Pair<String,Integer> pair =
  Pair.<String,Integer>makePair("toto",12);
System.out.println(pair); // affiche (toto, 12)
```

Le *diamond* <>

Depuis Java 7, lors de l'appel d'un constructeur, il est possible de laisser les arguments de type vide (*diamond* <>) du moment que le compilateur peut déterminer (inférer) le type ou les types à partir du contexte.

```
Box<Integer> integerBox = new Box<>();
public static <C, D> Pair<C,D>
    makePair(C first, D second) {
  return new Pair<>(first, second);
}
```

De même pour une méthode paramétrée en ne mettant même pas le *diamond*.

```
Pair<String,Integer> pair = Pair.makePair("tot",12);
```

Convention de nommage pour les types paramétrés

Par convention, les noms des paramètres de type sont des lettres majuscules.

C'est contraire bonnes pratiques visant à donner des noms détaillée aux éléments du code, mais indispensable pour différencier d'un coup d'œil les paramètres de type des classes.

Les noms les plus utilisés sont :

- `E` : *Element* (très utilisé pour les `Collections` de Java)
- `K` : *Key*
- `N` : *Number*
- `T` : *Type*
- `V` : *Value*

Section 2

Interfaces : type commun à plusieurs classes

Problèmes de recyclage

- Papiers, bouteilles, piles électriques, cageots, ... sont des objets **différents** :

  $\Rightarrow$ *déchirer* du papier, *remplir* une bouteille

- Mais ces objets partagent tous la propriété d'être **recyclable**

  $\Rightarrow$ 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 des classes.

Comment recycler tous les objets d'une poubelles ?

Avec le code suivant ?

```
T[] trashCan = // ...
for(T trashItem : trashCan){
    trashItem.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 provenant de classes différentes

Homogénéité des collections en Java

Les collections (`List`, tableaux, ...) sont **homogènes** : tous les éléments contenus dans la collection sont du même type.

Raisons :

- Parce que cela aide à la lisibilité du code (on comprend ce que contient la collection) comme par exemple lorsqu'on définit `List<Rectangle> rectangles` qui est une liste de rectangles.
- Cela évite des erreurs à l'exécution arrivant avec des langages non-statiquement typés comme Python lorsqu'on appelle une méthode sur un objet sans savoir si l'objet a une implémentation pour cette méthode.

Question

Les éléments de `TrashCan` doivent avoir un type, lequel ?

La solution objet

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.

Dans notre cas, la seule méthode qu'on pourra appeler sur les éléments de `trashCan` sera `recycle` même si les objets disposait à la base d'autres méthodes (par exemple `burn()` pour les `Paper`).

Solution java : `interface`

- En java, une `interface` est un ensemble de déclaration de **signatures** de méthodes (type de retour, nom de la méthode et type des arguments) 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.
- Une référence du type d'une interface accepte uniquement les appels de méthodes définies dans l'interface.

Exemple d'utilisation (1/2)

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

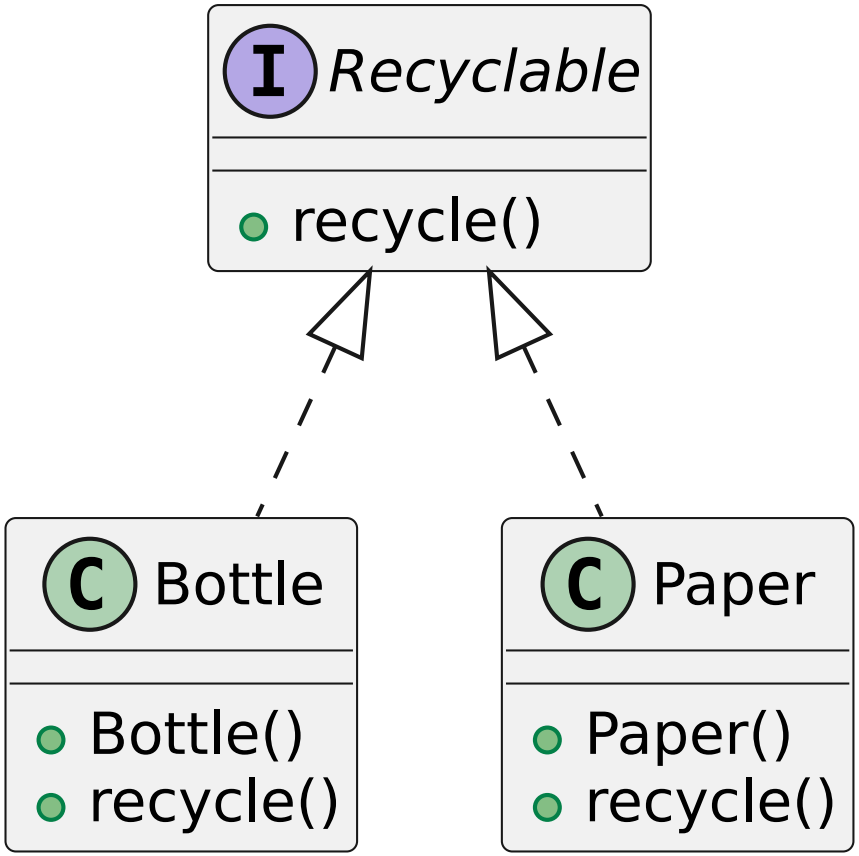
Exemple d'utilisation (2/2)

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

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

for(Recyclable trashItem : trashCan){
    trashItem.recycle();
}
```

La solution en diagramme



Implémentation d'une interface

Implémenter une interface

Une classe *implémente* une interface :

- si toute méthode déclarée dans l'interface est déclarée et programmée (codée) dans la classe.
- si la classe indique à sa déclaration qu'elle implémente l'interface.

- Une classe peut implémenter des méthodes en plus de l'interface
- Une classe peut implémenter plusieurs interfaces (séparées par des virgules à la déclaration)
- Pour implémenter une méthode d'une interface, la méthode de la classe doit avoir :
  - Le même nom
  - Les mêmes paramètres (les mêmes types) dans le même ordre
  - Le même type de retour
  - La même accessibilité publique (mot-clé `public`)

Interface en Java

```
public interface IntegerIterator {
  boolean hasNext();
  int next();
}
```

- Déclaration dans un fichier de même nom comme pour les classes.
- Liste de signatures de méthodes
- Pas d'instructions
- Pas d'attributs d'instances
- Pas de visibilité spécifiée pour les méthodes, car tout est public par défaut

Section 3

Interfaces : passer d'une implémentation à une autre

Interface

On vient de voir qu'une interface permet de définir un type commun correspondant à des instances de classes différentes.

$\Rightarrow$ cela permet de regrouper des objets partageant un même contrat (ensemble de méthode) dans une même collection (liste, tableau, ...) et d'utiliser ce service.

Une autre utilisation importante des interfaces est de séparer un contrat (déclaration des méthodes) de son implémentation (codes des méthodes) et donc de permettre de changer très facilement d'implémentation.

Inversion des dépendances (Dependency inversion principle)

Il faut dépendre des abstractions (les interfaces) et pas des implémentations (les classes).

Exemple : interface pour l'affichage

Supposons que des classes implémentent un service de façons différentes :

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

class BracePrinter {
    public 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`.

$\Rightarrow$ 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{
    /**
     * Print the {@code String} {@code document}
     * @param document the string to be printed
     */
    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 une `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]);
    }
}
```

Polymorphisme

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)
En programmation orientée objet, le polymorphisme permet à une classe d'être vue et traitée comme étant du type d'une des interfaces qu'elle implémente ou bien son propre type.

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}

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{
  void dial(string phoneNumber);
  void hangUp();
}
public interface TransmissionManager{
  void send(char c);
  char receive();
}
public class Modem
    implements ConnexionManager, TransmissionManager{
}
```

Implémentations multiples

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

```
public interface Printable {
    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.

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

Section 4

Extension d'interface

Extension d'interface

Supposons que nous ayons l'interface suivante :

```
public interface List<E> {
  public int size();
  public E get(int index);
}
```

Cette interface définit une liste permettant d'accéder à des éléments indexés.

Extension d'interface

En Java, on peut étendre une interface pour rajouter des services supplémentaires.

On peut donc étendre notre interface `List` pour ajouter des méthodes permettant de modifier les éléments de celle-ci.

```
public interface ModifiableList<E> extends List<E> {
  public void add(E value);
  public void remove(int index);
}
```

Une classe qui implémente l'interface `ModifiableList` doit implémenter les méthodes `size`, `get`, `add` et `remove`.

Extension d'interface

Supposons que la classe `ArrayModifiableList` implémente l'interface `ModifiableList`. Dans ce cas, nous pouvons écrire :

```
ModifiableList<Integer> modifiableList =
    new ArrayModifiableList<>();
modifiableList.add(2);
modifiableList.add(5);
modifiableList.remove(0);
List<Integer> list = modifiableList;
System.out.println(list.size());
```

En revanche, il n'est pas possible d'écrire :

```
  list.remove(0);
/* → Cette méthode n'existe pas */
/* dans l'interface List. */
```

Extension de plusieurs interfaces

Supposons que nous avons l'interface suivante :

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

En Java, une interface peut étendre plusieurs interfaces :

```
public interface ModifiablePrintableList
  extends ModifiableList, Printable {
}
```

Remarques

- Nous ne définissons pas de nouvelles méthodes dans l'interface `ModifiablePrintableList`.
  $\Rightarrow$ Cette interface ne représente que l'union des interfaces `ModifiableList` et `Printable`.
- De nouvelles méthodes auraient pu être définies dans `ModifiablePrintableList`.