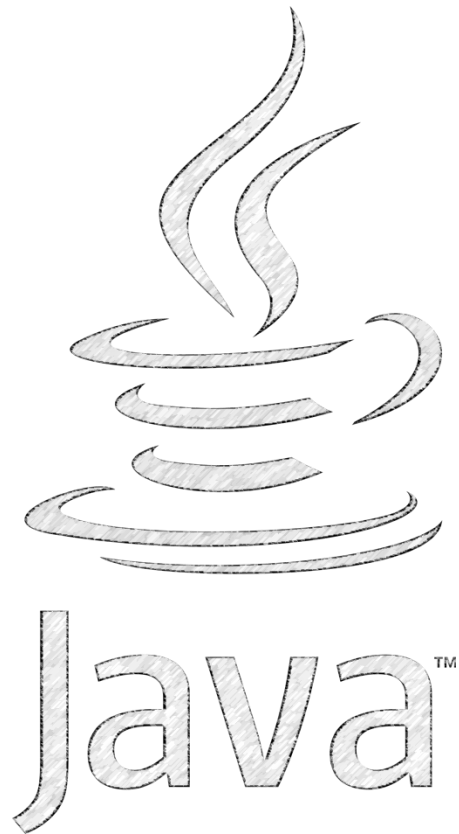


Programmation Orientée Objets avec Java

Stéphane Malandain, Yassin Rekik



Chapitre 7

Les types génériques

- Terminologie
- Utilité
- Raw type
- Mise en œuvre d'une Box
- Paramètres bornés
- Conventions

Les types génériques

La programmation générique

4

La généricité permet de concevoir des algorithmes qui peuvent être utilisés pour des objets de types différents. Il est donc possible de réduire la duplication de code en offrant une même implémentation tout en étant indépendant d'un type.

Nous avons déjà employé à plusieurs reprises des types génériques issus de la librairie standard (`List<E>`, `Map<K, V>`, `Iterable<E>`...). Pour une liste par exemple, nous pouvons facilement comprendre que l'ajout, le décompte, la récupération d'éléments est identique, qu'importe le contenu de la liste.

Nous allons donc voir comment réaliser nos propres types génériques. Nous verrons qu'il est possible de borner les paramètres d'un type générique

Si nous prenons l'exemple des listes :

- `List<E>` est un type générique.
- `E` est un **type paramétrique** ou un paramètre du type `List`
- Une fois le ou les paramètres renseignés, un type générique devient un type concret :
 - `List<Integer>` est un type concret, il s'agit d'une liste d'entiers.

Cela correspond à un type de polymorphisme : **le polymorphisme paramétrique**

La principale utilité est de réduire la duplication de code. Une autre utilité est de bénéficier d'un code plus sûr.

Sans générique, Java utilisait des `Object` pour n'importe quel type :

```
1  List ls = new ArrayList(); // équivaut à ArrayList<Object>
2  ls.add("Hello");
3  Object o = ls.get(0);
4  /* ne compile pas. get() retournait un Object
5  String s = ls.get(0);
6  */
7  String s = (String)o; // Cast /\
8
```

La dernière ligne est problématique et obligeait l'utilisateur à réaliser une conversion. Il devenait même important de vérifier la référence au préalable :

```
1  List ls = new ArrayList();  
2  ls.add("Hello");  
3  Object o = ls.get(0);  
4  if (o instanceof String) {  
5      String s = (String)o;  
6  }  
7
```

Grâce aux génériques, une fois le paramètre spécifié, le code devient plus robuste. La méthode `get` retourne le type spécifié et le cast devient inutile.

```
1    List<String> ls = new ArrayList<>();  
2    ls.add("Hello");  
3    String s = ls.get(0);
```


L'interface de `List` permet de bien comprendre le mécanisme :

```
1  interface List<E> {  
2      boolean add(E e);  
3      E get(int index);  
4      E remove(int index);  
5      E set(int index, E element);  
6      int size();  
7      ...  
8  }
```

Pour une liste de `String`, il devient possible d'interpréter, par substitution, l'interface ainsi :

```
1  interface List<String> {  
2      boolean add(String e);  
3      String get(int index);  
4      String remove(int index);  
5      String set(int index, String element);  
6      int size();  
7      ...  
8  }
```

Raw type

11

Les génériques ont été introduits avec Java 5. L'objectif était de rester compatibles avec les anciennes versions. Il est donc toujours possible d'utiliser une telle syntaxe sans les crochets, sans les paramètres :

```
List rawType = new ArrayList();
```

Un `rawType` ("type brut") est un type générique sans paramètre. Des alertes sont cependant levées à la compilation lors de manipulation dangereuse:

```
List rawType = new ArrayList();  
List<String> strings = rawType; // warning: unchecked conversion
```

Raw type

12

La référence `rawType` peut contenir n'importe quel objet. Du coup, récupérer un élément de `strings` ne retourne pas forcément un `String` !

```
List<String> strings = List.of("hello");  
List rawType = strings;  
rawType.add(42); // warning: unchecked call to add(E) ...
```

Dans l'exemple ci-dessus, `rawType` référence une liste de `String` mais peut ajouter n'importe quel type d'objets, ce qui viole le contrat de `strings`. Un *warning* est généré à la compilation et une exception serait levée lors de l'exécution.

Mise en oeuvre d'une Box

13

Réalisons maintenant notre propre classe générique selon le cahier des charges ci-dessous :

- Une boîte `Box` peut contenir une valeur de n'importe quel type
- Il est possible de récupérer sa valeur ou de modifier sa valeur
- Elle redéfinit la méthode `toString()`, `equals()` et `hashCode()`
- Il ne doit pas être possible d'hériter de `Box`

Mise en oeuvre d'une Box

14

```
1  import java.util.Objects;
2
3  public final class Box<T> {
4
5      private T value;
6
7      public Box(T value) { this.value = value; }
8      public T get() { return this.value; }
9      public void set(T newValue) { this.value = newValue; }
10
11     @Override
12     public String toString() { return "[" + this.value.toString() + "]"; }
13
14     @Override
15     public boolean equals(Object o) {
16         if (this == o) {
17             return true;
18         }
19         if (o == null || o.getClass() != this.getClass()) {
20             return false;
21         }
22         Box<?> other = (Box<?>)o;
23         return this.value.equals(other.value);
24     }
25
26     @Override
27     public int hashCode() {
28         return Objects.hash(this.value);
29     }
30 }
```

Mise en oeuvre d'une Box

15

```
1  import java.util.Objects;
2
3  public final class Box<T> {
4
5      private T value;
6
7      public Box(T value) { this.value = value; }
8      public T get() { return this.value; }
9      public void set(T newValue) { this.value = newValue; }
10
11     @Override
12     public String toString() { return "[" + this.value.toString() + "]; }
13
```

Mise en oeuvre d'une Box

16

```
13
14 @Override
15 public boolean equals(Object o) {
16     if (this == o) {
17         return true;
18     }
19     if (o == null || o.getClass() != this.getClass()) {
20         return false;
21     }
22     Box<?> other = (Box<?>)o; // utilise le jocker pour convertir la référence en une Box
23     return this.value.equals(other.value); // utilise le equals de l'objet qui se trouve dans la boîte
24 }
25
26 @Override
27 public int hashCode() {
28     return Objects.hash(this.value);
29 }
30 }
```


Mise en oeuvre d'une Box

17

Exemple d'utilisation :

```
1  Box<Integer> box1 = new Box<Integer>(1);
2  Box<Integer> box2 = new Box<>(2); // Diamond : Le type est inféré
3
4  System.out.println( box1 );
5  System.out.println( box2 );
6  System.out.println( box1.equals(box2) );
7  System.out.println( new Box<>(3) );
8  System.out.println( box1.get() );
9
```

Mise en oeuvre d'une Box

18

Affichage :

[1]

[2]

False

[3]

1

Methode d'instance générique

19

Ajoutons une fonction permettant de transformer une boîte que nous appelons `map`. Nous souhaitons par exemple transformer une boîte contenant un entier en une boîte contenant une chaîne de caractères : `Box<Integer> -> Box<String>`. Cette méthode retourne une nouvelle boîte.

Par exemple :

```
1  Box<Integer> bi = Box<>(42);  
2  Box<String> bs = bi.map( i -> "La boite contient: " + String.valueOf(i) + " !" );  
3  System.out.println(bs);  
4
```

Doit afficher :

[La boite contient: 42 !]

Methode d'instance générique

20

Voici la méthode à ajouter à notre `Box` :

```
1  // (Box<T>, T -> R) -> Box<R>
2  public <R> Box<R> map(Function<T, R> f) {
3      /*      ^
4      *      |
5      *      \- - - - permet de retourner une boîte d'un autre type */
6
7      return new Box<R>(f.apply(this.t));
8  }
```

Cette méthode est générique, car sans la déclaration du `public <R> Box...`, le compilateur rechercherait un type `R` existant.

Methode d'instance générique

21

Déclaration complète de la classe avec cette nouvelle méthode :

```
1  import java.util.Objects;
2  import java.util.function.Function;
3
4  public final class Box<T> {
5
6      private T value;
7      public Box(T value) { this.value = value; }
8      public T get() { return this.value; }
9      public void set(T newValue) { this.value = newValue; }
10
11     @Override
12     public String toString() { return "[" + this.value.toString() + "]"; }
13
14     @Override
15     public boolean equals(Object o) {
16         if (this == o) {
17             return true;
18         }
19         if (o == null || o.getClass() != this.getClass()) {
20             return false;
21         }
22         Box<?> other = (Box<?>)o;
23         return this.value.equals(other.value);
24     }
```

Methode d'instance générique

22

Suite :

```
25
26  /* TODO:
27   *      /- - - enlever le <R> ici et compilez !
28   *      |
29   *      v                                     */
30  public <R> Box<R> map(Function<T, R> f) {
31      return new Box<R>(f.apply(this.value));
32  }
33
34  @Override
35  public int hashCode() {
36      return Objects.hash(this.value);
37  }
38  }
```

Essayez le TODO pour comprendre pourquoi cette méthode doit être générique !

Méthode statique générique

23

Dans cet exemple, il est important de déclarer le type `T` comme étant un paramètre. Si nous l'omettons, le compilateur chercherait un type `T` connu.

La méthode `areEquals` permet ici de contraindre les boîtes à comparer à avoir le même type.

```
1  class Util {  
2      public static <T> boolean areEquals(Box<T> b1, Box<T> b2) {  
3          return b1.equals(b2);  
4      }  
5  }
```

Methode statique générique

24

Exemple d'utilisation :

```
1  Util.<Integer>areEquals(new Box<>(10), new Box<>(10)); // ==> true
2
3  Util.<Integer>areEquals(new Box<>(10.0), new Box<>(10)); // ==> Erreur de compilation,
4  |           |           |           |           |           |           |           |           |           |
5  |           |           |           |           |           |           |           |           |           |
6  Util.areEquals(new Box<>(10.0), new Box<>(10)); // ==> false (compile et s'exécute)
7  // en l'occurrence, le paramètre T serait inféré
8  // comme Object (ou plus probablement Serializable)
```


Paramètres bornés

25

Un paramètre peut être borné lors de sa déclaration. Il est donc possible de préciser qu'un paramètre doit être d'un type ou d'un sous-type particulier.

Imaginons que nous souhaitons créer uniquement des boîtes numériques. Une borne se déclare à l'aide du mot-clé `extends`.

```
1  public final class Box<T extends Number> {  
2      ...  
3  }  
4  
5      ...  
6  Box<Integer> bi = ... // ok, Integer hérite de Number  
7  Box<Double> bd = ... // ok, Double hérite de Number  
8  Box<String> bs = ... // erreur de compilation, String n'est pas un Number
```

Paramètres bornés

26

Exemple d'une méthode générique bornée (en l'occurrence statique)

```
1  class Util {
2      public static <T extends Number> Box<T> toBox(T v) {
3          return new Box<T>(v);
4      }
5  }
6  ...
7  Box<Integer> bi = toBox(22); // ==> ok
8  Box<Integer> bi = toBox(22.0); // ==> ERROR:
9      // incompatible types: inference variable T has incompatible bounds
10 Box<Double> bd = toBox(22.0); // ok
```

La borne peut être une classe ou une interface. Il est possible de contraindre le paramètre à plusieurs types :

```
<T extends I1 & I2 & I3> // T doit respecter plusieurs interfaces  
<T extends C & I1 & I2> // T doit être une classe C ou hériter de C  
                        // et respecter les interfaces I1 et I2
```

Si une des bornes est une classe, elle doit être placée en premier,

Nommage des paramètres

28

Le paramètre, en Java, est généralement une lettre indiquant parfois la nature de celui-ci :

- E - Element
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types