Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

2021 - 2022
Ateliers Logiciels

Java Generics

La généricité en Java

Dans ce document vous trouverez un introduction à la notion de *Generics* en Java.

Le code est disponible via git à l'adresse :
`https://git.esi-bru.be/ATL/generics.git`

Version du document : 15 novembre 2021

1 Introduction

Le but de ce document est d'introduire à la notion de classe et de méthode générique en Java. En particulier nous voulons comprendre ce genre d'entête :

```
1  public static <T> void copy(List<? super T> dest,
2      List<? extends T> src)
```

En effet, à la lecture de cet en-tête de méthode statique de la classe `java.util.Collections`, il n'est pas évident de savoir quels types de `List` nous pouvons passer comme destination et comme source de la copie.

Il en est de même pour la méthode `sort` couramment utilisée :

```
1  public static <T> void sort(List<T> list, Comparator<? super T> c)
```

On trouve de nombreux exemples d'utilisation des classes génériques dans les librairies, et en particulier dans les librairies graphiques telles que `JavaFX`. Par exemple, la classe `javafx.scene.Node` définit la propriété suivante :

```
1  ObjectProperty<EventHandler<? super MouseEvent>> onMouseEntered
```

Une bonne compréhension des génériques est nécessaire pour éviter de tomber dans le culte du Cargo [1]

1. https://fr.wikipedia.org/wiki/Culte_du_cargo#En_informatique (consulté le 23 novembre 2020).

2 Classes génériques

La classe `Box` représente un enrobage d'un objet. Cette classe est une classe générique, elle est paramétrée par le type `T`, le type de l'objet contenu dans la boîte.

```
1  package esi.atl.generics;
2
3  public class Box<T> {
4      T element;
5
6      public Box(T element) {
7          this.element = element;
8      }
9      public T getElement() {
10         return element;
11     }
12     public void setElement(T element) {
13         this.element = element;
14     }
15     @Override
16     public String toString() {
17         return "[" + element + "]";
18     }
19     public static void main(String[] args) {
20         Box<Integer> box = new Box<Integer>(42);
21         Integer value = box.getElement();
22         System.out.println(box);
23         System.out.println(value);
24         box.setElement(43);
25         System.out.println(box);
26     }
27 }
```

La classe `Box` va pouvoir être utilisée avec différents types. Dans l'exemple donné, la variable `box` est de type `Box<Integer>` cela signifie que l'on pourra y mettre des entiers. L'utilisation d'une classe générique nous permet d'éviter un *cast* à la ligne 21

```
Integer value = box.getElement();
```

le compilateur sait que `box` ne contient que des entiers.

Cela nous permet aussi d'assurer que seules des entiers seront mis dans la boîte. Ainsi, nous ne pourrions pas écrire

```
box.setElement(new Double(43));
```

Question 1

Quelle erreur lève le compilateur si vous tentez d'ajouter la ligne ci-dessus ?

Question 2

Lors de l'instanciation et de l'assignation à la ligne
`Box<Integer> box = new Box<Integer>(new Integer(42));`
quel avertissement donne NetBeans ?

En effet, comme le compilateur peut déduire que le paramètre de la classe `Box` est `Integer`, car le nouvel objet (la nouvelle boîte) est assigné à une variable de type `Box<Integer>`, le paramètre `Integer` est redondant. On peut écrire en utilisant l'opérateur *diamond* :

```
Box<Integer> box = new Box<>(new Integer(42));
```

Notez finalement qu'il y a un *autoboxing* qui est effectué par le compilateur afin de convertir l'entier 42 du type `int` vers `Integer`. Il n'est donc pas utile d'instancier explicitement l'`Integer`.

Depuis JDK10, il est bien sûr possible d'utiliser l'inférence de type et d'écrire ce qui suit. Dans ce cas, il faut bien préciser le type à l'instanciation :

```
var box = new Box<Integer>(42);
```

3 Méthodes génériques

Les méthodes, statiques ou non, peuvent également être paramétrées. Par exemple, la méthode suivante est paramétrée par le type `T`.

```
1  package esi.atl.generics;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class TestMethod {
7
8      public static <T> List<T> arrayToList(T[] t) {
9          List<T> list = new ArrayList<>();
10         for(T element : t)  list.add(element);
11         return list;
12     }
13     public static void main(String[] args) {
14         Point[]  points = {new Point(), new Point(1,2)};
15         List<Point> list = TestMethod.<Point>arrayToList(points);
16     }
17 }
```

Cette méthode reçoit un tableau d'éléments de type `T` et le transforme en une liste. Pour l'utiliser dans la méthode `main` un appel classique suffit :

```
List<Point> list = arrayToList(points);
```

Le type effectif de `T` est inféré (calculé) par le compilateur, il ne faut donc pas le préciser. Nous pouvons cependant, si la lisibilité le demande, préciser ce type (il faut alors toujours mettre le nom de la classe) :

```
List<Point> list = TestMethod.<Point>arrayToList(points);
```

Nous verrons d'autres exemples plus loin.

4 Héritage et génériques

Mélanger l'héritage et la généricité est nécessaire mais parfois source de confusion.

Pour rappel, les classes `Integer` et `Double` sont des sous-classes de la classe abstraite `Number` qui est une sous-classe directe de la classe `Object`. Les classes `Double` et `Integer` n'ont pas de lien de sous-classe entre elles.

Question 3

À votre avis, le type `Box<Integer>` est-il un sous-type du type `Box<Object>` ? En d'autres mots, peut-on écrire
`Box<Object> box = new Box<Integer>(42);`

Prenez la classe test `TestBoxSubtyping`

```
1  package esi.atl.generics;
2
3  public class TestBoxSubtyping {
4
5      public static void main(String[] args) {
6          Box<Object> box = new Box<Integer>(new Integer(42));
7          //Box<Integer> box = new Box<Integer>(new Integer(42));
8          Object value = box.getElement();
9          System.out.println(box);
10         System.out.println(value);
11         box.setElement(new Double(43));
12         System.out.println(box);
13     }
14 }
```

Question 4

Quelle erreur lève le compilateur à la ligne 6 ?

Question 5

Remplacez la ligne 6 par
`Box<Integer> box = new Box<Integer>(new Integer(42));`
quelle erreur lève maintenant le compilateur et à quelle ligne ?

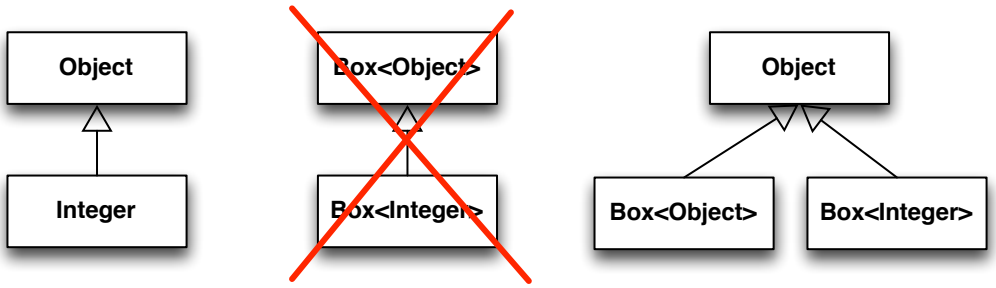
Si on pouvait écrire

```
Box<Object> box = new Box<Integer>(42);
```

alors la ligne

```
box.setElement(new Double(43));
```

ajouterait un `Double` dans une boîte d'entier. Cette dernière ligne doit compiler car ajouter un `Double` (ou tout autre objet) à une boîte d'`Object` doit-être légal.



5 Jokers

Dans la section précédente, nous avons vu que si `B` est une sous-classe de `A`, cela n'implique pas que le type `C<B>` est un sous-type de `C<A>`. Il est cependant parfois nécessaire de permettre une forme de polymorphisme sur les types génériques. Pour cela Java introduit les jokers, en anglais *wildcards*.

```
1  package esi.atl.generics;
2
3  public class TestBoxWildcards {
4      public static void main(String[] args) {
5          Box<?> box;
6          box = new Box<Integer>(new Integer(42));
7          box = new Box<Object>(new Integer(42));
8
9          Object value = box.getElement();
10         System.out.println(box);
11         System.out.println(value);
12         box.setElement(new Integer(43));
13         System.out.println(box);
14     }
15 }
```

Le variable `box` a le type `Box<?>` ce qui veut dire qu'elle peut référencer tout objet de type `Box<T>` ou `T` est un type quelconque. Dans l'exemple, on lui assigne successivement des objets de types `Box<Integer>` et `Box<Object>`.

Question 6

Quelle erreur lève le compilateur et à quelle ligne ?

En effet si `box` référence une `Box` contenant des objets de type quelconque, `box` peut par exemple référencer une boîte de type `Box<Point>` dans laquelle on ne peut mettre des `Integer`.

Le compilateur ne peut donc accepter de mettre quoi que ce soit comme élément dans `box`. Plus exactement, la méthode `setElement(T element)` est inutilisable, car elle ne peut recevoir qu'une référence compatible avec tous les types possibles de T. Dans ce cas-ci la seule référence acceptable est la référence `null` (mais cela n'est pas très utile).

On se demande dès lors quand est-ce utile d'utiliser un joker. Pour cela considérons l'exemple suivant.

```
1  package esi.atl.generics;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class TestWildcardsList {
7
8      public static void printList(List<?> list) {
9          for (Object n : list) {
10             System.out.println(n);
11         }
12     }
13     public static void main(String[] args) {
14         ArrayList<Integer> list = new ArrayList<>();
15         //ArrayList<Point> list = new ArrayList<>();
16         //ArrayList<Object> list = new ArrayList<>();
17         list.add(44); list.add(42); list.add(43);
18         printList(list);
19     }
20 }
```

La méthode `printList` accepte toute `List` générique quel que soit son paramètre de type. On peut par exemple lui passer un `ArrayList<Integer>`, un `ArrayList<Point>` ou un `ArrayList<Object>`.

Notez que dans la méthode `printList` la seule chose que l'on sait des objets dans `list` est qu'ils sont de type `Object`. On ne peut pas être plus précis, car le joker ne nous donne pas plus d'information.

6 Bornes supérieures et inférieures

Il est possible de donner une borne supérieure (ou de donner une borne inférieure) aux paramètres de type et aux jokers.

6.1 Bornes supérieures sur les paramètres de type

Considérons la classe `Pair` :

```
1  package esi.atl.generics;
2
3  public class Pair<T extends Comparable<T>> {
4      private final T first;
5      private final T second;
6
7      public Pair(T first, T second) {
8          this.first = first;
9          this.second = second;
10     }
11     public T getFirst() { return first; }
12     public T getSecond() { return second; }
13
14     public boolean ordered() {
15         return first.compareTo(second) < 0;
16     }
17     @Override
18     public String toString() {
19         return "("+first+", "+second+")";
20     }
21
22     public static void main(String[] args) {
23         Pair<Integer> pair = new Pair<>(32, 42);
24         System.out.println("la paire: "+pair
25                            + (pair.ordered()?" est":" n'est pas")+" ordonnée");
26     }
27 }
```

Question 7

Quelle erreur signale le compilateur si l'on retire `extends Comparable<T>` dans l'en-tête de la classe comme suit ?
`public class Pair<T> ...`

Cette classe est paramétrée par le type `T` qui doit être un sous-type de `Comparable<T>` [2], c'est-à-dire que la classe `T` doit avoir une méthode `compareTo(T t)` (javadoc [3]). Cela nous autorise à écrire la méthode `ordered()`.

Cette classe peut être instanciée avec comme paramètre toute classe qui implémente l'interface `Comparable`. Dans notre exemple on l'instancie avec la classe `Integer`.

2. Le mot clef `extends` est utilisé pour les classes mais aussi pour les interfaces, on n'utilise donc pas `implements`.
3. https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

Question 8

Quelle erreur lève le compilateur si l'on remplace la 1er ligne du `main` par la ligne suivante ? Pourquoi ?

```
Pair<Number> pair = new Pair<>(new Integer(32), new
Integer(42));
```

6.2 Bornes supérieures sur les jokers

Considérons l'exemple suivant avec la méthode `printGreaterThan` qui affiche les éléments de la liste qui sont plus grands qu'un élement passé en paramètre.

```
1  package esi.atl.generics;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class TestBoundedWildcards {
7      static <T> void printGreaterThan(List<? extends Comparable<T>> list, T e) {
8          for (Comparable<T> n : list) {
9              if(n.compareTo(e)>0) System.out.println(n);
10         }
11     }
12     public static void main(String[] args) {
13         ArrayList<Integer> list = new ArrayList<>();
14         list.add(44); list.add(42); list.add(43);
15         list.add(45); list.add(39); list.add(47);
16         printGreaterThan(list, 43);
17     }
18
19 }
```

Le joker est borné par l'interface `Comparable<T>`, cela permet de savoir que chaque élément de la liste est de type `Comparable<T>` (notez bien le type de la variable dans le `for`). Ces éléments peuvent donc être comparés à un élément de type `T` via la méthode `compareTo(T t)`.

6.3 Bornes inférieures

Lorsqu'il n'y a pas de borne supérieure, il est possible de spécifier une borne inférieure de manière similaire via le mot clef `super`.

Par exemple, dans la classe utilitaire `Collections` la méthode :

```
1      public static <T> boolean addAll(Collection<? super T> c,
2          T... elements)
```

permet d'ajouter les objets `elements` de type `T` à la collection `c` qui doit être de type `Collections<U>`, où `U` est de type `T` ou une super classe de `T`.

Question 9

Voici un en-tête de méthode générique de la classe `Collections` :
`static <T> void copy(List<? super T> d, List<? extends T> s)`
Cette méthode est plus générale que la méthode :
`static <T> void copy2(List<T> d, List<T> s)`
Pour montrer cela, donnez un exemple d'appel de la méthode `copy` tel que le type de `d` est différent du type de `s`.

Question 10

Voici un en-tête de méthode générique de la classe `Collections` :
`static <T> void sort(List<T> list, Comparator<? super T> c)`
Donnez un exemple d'appel de cette méthode en spécifiant le type de chaque argument.

6.4 Hiérarchie et sous-types

Le schéma suivant illustre les relations qu'implique l'utilisation de jokers combinée avec des bornes inférieures et supérieures.