

Types génériques

Conception Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1



Les collections

Rappel : les collections sont typées : utilisation des types génériques.

- Collection<E>, List<E>, Set<E>

tye paramétré

On fixe le type des éléments à la construction :

```
List<String> l = new ArrayList<String>();
Collection<Hobbit> c = new HashSet<Hobbit>();
```

renforcement du typage : contrôle à la compilation

(à partir de java 1.5)

Autre exemple

L'interface java.lang.Comparable :

```
java.lang.Comparable<T>
    public int compareTo(T o)
```

où **T** représente le type des éléments comparés.

Implémentation :

```
public class String implements Comparable<String>, ...
```

Effacement de type

de java.sun.com

*Generics are implemented by **type erasure**: generic type information is present **only at compile time**, after which it is erased by the compiler. The main advantage of this approach is that it provides total interoperability between generic code and legacy code that uses non-parameterized types (which are technically known as raw types). The main disadvantages are that parameter type information is not available at run time, and that automatically generated casts may fail when interoperating with ill-behaved legacy code.*

Différence avec C++ où les templates produisent des nouvelles classes.

Créer un type générique

```
package value;
public class Value<T> {
    private T v;
    public Value (T v){
        this.v = v;
    }
    public T getValue() {
        return this.v;
    }
} // Value
```

Usage

```
Value<Integer> v = new Value<Integer>(12);
Value<List<Chou>> v = new Value<List<Chou>>(new ArrayList<Chou>());
```

- On veut définir une classe permettant de lier des identifiants (**String**) à des valeurs (**Value**).
- Le type des valeurs est inconnu a priori.
 - ↔ comment le déclarer ?
 - ↔ comment signifier “une valeur quelconque” ?

Utilisation de **joker** (*wildcard*) : **?. Value<?>**

```
package value;
public class Context {
    private Map<String,Value<?>> lesVariables;
    public Context() {
        this.lesVariables = new HashMap<String,Value<?>>();
    }
    public void addVariable(String id, Value<?> var) {
        this.lesVariables.put(id, var);
    }
    public Value<?> getVariable(String id) {
        return this.lesVariables.get(id);
    }
}

//=====
public static void main(String[] args) {
    Context c = new Context();
    c.addVariable("v1", new Value<Integer>(12));           v1: class value.Value
    c.addVariable("v2", new Value<Boolean>(true));         v2: class java.lang.Boolean
    S.o.p("v1: " +c.getVariable("v1").getClass());
    S.o.p("v2: " +c.getVariable("v2").getValue().getClass());
}
} // Context
```

Méthode générique

méthode paramétrée par un type

Dans la classe Context ajouter une méthode addValue qui prend l'identifiant et une valeur du type qui paramètre Value<T> :

```
public <T> void addValue(String id,T value) {
    Value<T> v = new Value<T>(value);
    this.addVariable(id, v);
}
//-----
Context c = new Context();
c.addValue("var3",10);
c.addValue("var4",true);
```

Problèmes liés au typage

- ArrayList<String> est un sous-type de Collection<String>
- Collection<String> n'est pas un sous-type de Collection<Object>

Conséquence,

```
Collection<Hobbit> colHob = new ArrayList<Hobbit>(); // ok
Collection<Object> c = new ArrayList<Hobbit>(); // ne compile pas !!!
Collection<Object> c = colHob; // idem : incompatible types
```

et donc, soit :

```
public void dump(Collection<Object> c) {
    for (Object o : c) {
        System.out.println(o);
    }
}
```

ne peut pas prendre pour paramètre autre chose que Collection<Object>. xxx.dump(new ArrayList<Hobbit>()) ne compile pas !

- Collection<Object> ne signifie pas "n'importe quelle collection pourvue qu'elle contienne des objets" mais bien "collection d'Objects"

- Comment exprimer "n'importe quelle collection" ?
càd le type qui réunit toutes les collections

Collection<?>

```
package value;
import java.util.*;
public class Context {
    private Map<String,Value<?>> lesVariables;
    public Context() {
        this.lesVariables = new HashMap<String,Value<?>>();
    }
    public void addVariable(String id, Value var) {
        this.lesVariables.put(id, var);
    }
    public Value<?> getVariable(String id) {
        return this.lesVariables.get(id);
    }
}
} // Context
```

Collection<?> (collection d'inconnus, ? = joker)

mais la seule garantie sur les éléments c'est que ce sont des Objects

```
public void dump(Collection<?> c) {
    for (Object o : c) {
        System.out.println(o);
    }
}
```

xxx.dump(new ArrayList<Hobbit>()) est légal. Mais :

```
Collection<?> c = new ArrayList<Hobbit>();
c.add(new Hobbit(...)); // ne compile pas
```

```
public void drawAll(Collection<Shape> c) {
    for (Shape o : c) {
        o.draw();
    }
}
```

permet :

```
List<Shape> theShapes = new ArrayList<Shape>();
xxx.drawAll(theShapes); // ok
```

mais pas :

```
// Circle implements Shape
List<Circle> theCircles = new ArrayList<Circle>();
xxx.drawAll(theCircles); // ne compile pas, même raison
```

Comment exprimer :

une collection de *n'importe quoi du moment que c'est une Shape*
 càd du moment que c'est un **sous-type** de Shape

```
Collection<? extends Shape>
```

On a alors :

```
public void drawAll(Collection<? extends Shape> c) {
    for (Shape s : c) {
        s.draw();
    }
}
```

et alors xxx.drawAll(new ArrayList<Circle>()) est légal.

super

Il existe **super** pour réclamer un type *plus général*.

Syntaxe	interprétation
?	n'importe quel type
? <u>extends</u> T	n'importe quel sous-type de T
? <u>super</u> T	n'importe quel super-type de T

Exemples :

- dans List<E> :
 boolean addAll(Collection<? extends E> c)
- dans Collections :
 static <T> void sort(List<T> list, Comparator<? super T> c)
- dans Collections :
 static <T> void copy(List<? super T> dest, List<? extends T> src)

Combinaison de types

Possibilité de préciser “plusieurs bornes” aux jokers à l’aide de l’opérateur **&**.

Par exemple, si avec addVariable, on veut que les valeurs ajoutées soient **à la fois** des Shape et Clonable, alors on peut écrire :

```
public <T extends Shape & Clonable> void addVariable(String id,T value)
```

Conversion de code

“In general, if you have an API that only uses a type parameter T as an argument, its uses should take advantage of lower bounded wildcards (? super T). Conversely, if the API only returns T, you’ll give your clients more flexibility by using upper bounded wildcards (? extends T).”

pas toujours trivial :
 cf. dans Collection<E> :

```
static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```