

Cours 6 : Types et méthodes paramétrés

Types paramétrés

Wrappers

Erasure

Wildcard

Redéfinition

Type paramétré

Ajouté en 2004 à Java 5

Permettre au compilateur de suivre/tracker les types des éléments des collections.

Exemple avant Java 5

```
ArrayList list = new ArrayList();
```

```
list.add("hello");
```

```
String s = list.get(0); // ne compile pas
```

```
String s = (String) list.get(0); // Ok, mais dangereux
```

Cast d'objets en Java

Les casts d'objets sont vérifiés à l'exécution par la machine virtuelle

Toujours avant Java 5

```
ArrayList list = new ArrayList();
```

```
list.add("hello");
```

```
list.add(3);
```

```
// et plus tard dans le code
```

```
String s = (String) list.get(1); // plante avec CCE
```

ClassCastException

CCE et Maintenance

Avoir des casts dans un programme veut dire que le programme peut planter à un endroit si on n'ajoute pas les bons objets à un autre Endroit

Et les deux endroits peuvent être éloignés

Le but des types paramétrés:


supprimer ces casts d'objet en ajoutant le type des objets stockés au type de la collection

Types paramétrés - vocabulaire

- ▶ On appelle **type paramétré** un type (classe, interface..) possédant des paramètres de type.
 - ▶ Ex : `List<T>` est un type paramétré dont `T` est le paramètre de type.
- ▶ Les types qui prennent les paramètres de type lors de l'utilisation du type paramétré s'appellent types arguments.

Type paramétré


A partir de Java 5,
on ajoute le type des éléments



```
ArrayList<String> list = new ArrayList<String>();  
list.add("hello");
```

```
list.add(3); // ne compile pas
```

```
String s = list.get(0); // ok
```



plus besoin de cast

Déclaration de type

On déclare les variables de type (E) entre “<” et “>” après le nom de la classe/record (séparées par des virgules s’il y en a plusieurs)

```
public record Holder<E>(E element) {  
    public E value(E defaultValue) {  
        return element != null? element: defaultValue;  
    }  
}
```

déclaration

utilisation

Après la déclaration, E est une variable de type que l’on peut utiliser là où habituellement on utilise un type (déclaration de champs, de variable, de paramètre, etc)

Déclaration de type

- Un type (ici Pair) peut déclarer une ou plusieurs variables de type (après son nom) et les utiliser dans ses membres.

```
1 public class Pair<T,U> {  
2     private final T first ;  
3     private final U second ;  
4  
5     public Pair(T first, U second) {  
6         this.first = first ;  
7         this.second = second ;  
8     }  
9 }
```

- A l'instanciation, les types arguments sont associés à T et U pour chaque instance !
 - Integer associé à T, String associé à U

```
1 Pair<Integer,String> p = new Pair<Integer, String>(3, "boo") ;  
2 Pair<Date,Double> p2 = new Pair<Date,Double>(null, 2.2) ;
```


Déclaration de méthode paramétrée

Les méthodes sont paramétrées pour indiquer des relations entre le type des paramètres et le type de retour

```
public class Utils {  
    public static <T> List<T> from(T one, T two) {  
        return List.<T>of(one, two);  
    }  
}
```

déclaration

utilisation

On déclare les variables de type après les modificateurs de visibilité et avant le type de retour

```
public static <E> void copy(List<E> src, List<E> dst) {  
    ...  
}
```

Utilisation d'une variable de type dans une méthode paramétrée

Dans une méthode paramétrée, la variable de type n'est accessible que dans cette méthode

```
public class Utils {  
    public static <T> List<T> from(T one, T two) {  
        T t; // ok  
    }  
    private final T t; // ne compile pas  
}
```

Utilisation d'une méthode paramétrée

Pour appeler une méthode paramétrée, il faut mettre les “<” et “>” après le ‘.’ et avant le nom de la méthode

```
Utils.<String>from("foo", "bar")
```

Attention, ne pas écrire

```
Utils.from<String>("foo", "bar")
```

le “<” est considéré comme le inférieur ($2 < 3$), pas comme le début d'un type argument

Utilisation d'une variable de type dans une méthode paramétrée

Dans une méthode paramétrée, la variable de type n'est accessible que dans cette méthode

```
public class Utils {  
    public static <T> List<T> from(T one, T two) {  
        T t; // ok  
    }  
    private final T t; // ne compile pas  
}
```

Utilisation d'une méthode paramétrée

Pour appeler une méthode paramétrée, il faut mettre les “<” et “>” après le ‘.’ et avant le nom de la méthode

```
Utils.<String>from("foo", "bar")
```

Attention, ne pas écrire

```
Utils.from<String>("foo", "bar")
```

le “<” est considéré comme le inférieur ($2 < 3$), pas comme le début d'un type argument

Bornes

- Déclaration d'une variable de type **avec une borne**.
- A l'instanciation, les types paramétrés doivent être un sous-type des bornes.

```
1 public class Pair<T extends Number,U extends Object> {
2     private final T first ;
3     private final U second ;
4
5     public Pair(T first, U second) {
6         this.first = first ;
7         this.second = second ;
8     }
9
10    public static void main(String[] args) {
11        Pair<Integer,String> p = new Pair<Integer, String>(3, "boo") ;
12        Pair<String, Integer> p2 = new Pair<String, Integer>("boo", 3) ; //compile pas
13    }
14 }
```

Bornes

- ▶ Si pas de borne : la borne est `Object`.
- ▶ Borne doit être un `Object` (pas type primitif).
- ▶ La borne peut être une autre variable de type.

```
1 public interface Djsdijklq <T extends Number, U extends T> {
```

Bornes multiple

- ▶ On peut spécifier plusieurs types à une borne, séparation par **&**.
 - ▶ 0 ou 1 classe **puis** des interfaces.
 - ▶ Pas de multiple si variable de type.

```
1 public class Dauphine<T extends Clonable & Closable> { //2 interf, ok
2 public class Dauphine<T extends Clonable & Date> { //KO, Date pas interf
3 public class Dauphine<T extends Date & Clonable> { //ok
4 public class Dauphine<T, U extends T & Clonable> { //ko
```


Cours 6 : Types et méthodes paramétrés

Types paramétrés

Wrappers

Erasure

Wildcard

Redéfinition

Type primitif ?

Un type argument d'un type paramétré doit être un objet

- **new** ArrayList<String>(); // ok
- **new** ArrayList<int>(); // ne compile pas !

il ne peut pas être un type primitif !

Type wrapper / box

L'API de Java possède des classes pré-existantes non modifiables qui stockent un champ de type primitif

Il existe une classe par type primitif

java.lang.Boolean stocke un boolean

java.lang.Byte stocke un byte

java.lang.Short stocke un short

java.lang.Character stocke un char

java.lang.Integer stocke un int

java.lang.Long stocke un long

java.lang.Float stocke un float

java.lang.Double stocke un double

Box et type paramétré

donc au lieu de

```
var list = new ArrayList<int>(); // ne compile pas
```

on va écrire

```
var list = new ArrayList<Integer>();
```

```
Integer box = Integer.valueOf(3); // convertit un int en Integer  
list.add(box);
```

```
Integer box2 = list.get(0);
```

```
int value = box2.intValue(); // convertit un Integer en int
```

valueOf() et **intValue()** permettent les conversions

Auto-boxing / Auto-unboxing

Lorsque l'on fait un appel de méthode ou une assignation (un '='), le compilateur fait les conversions automatiquement

```
var list = new ArrayList<Integer>();
```

```
Integer box = 3; // appel Integer.valueOf(3)  
list.add(box);
```

```
int box2 = list.get(0); // appel integer.intValue()
```

Apparté Wrappers

- ▶ Classes wrappers pour **voir un type primitif comme un objet**.
- ▶ Permet d'utiliser du code marchant avec des objets.
 - ▶ Par exemple, la méthode `add` de `List` attend un objet en argument mais on veut pouvoir avoir des listes d'entiers (type primitif).

Les box sont des classes comme les autres

A part l'auto-boxing/auto-unboxing, une box se comporte comme une classe classique

```
var value1 = Integer.valueOf(1_000);  
var value2 = Integer.valueOf(1_000);  
value == value2 // false
```

Attention: Ne pas faire de test `==` ou `!=` sur les wrappers

```
Integer box = null;  
int value = box; // NPE
```

Attention: au wrapper null et l'unboxing

Cours 6 : Types et méthodes paramétrés

Types paramétrés

Wrappers

Erasure

Wildcard

Redéfinition

Generics: Nom de l'implantation des types paramétrés en Java

- Le compilateur traduit les types paramétrés et le code d'appel en code classique !
 - Les types paramétrés n'existent que pour le compilateur pas pour la VM à l'exécution
- Cette astuce s'appelle l'eraseure

Erasure

- **Pas de type paramétré à l'exécution !**
- Remplacé par sa borne (donc Object si pas de borne).

```
1 public class Node<T> {  
2     private final T t ;  
3     public Node(T t) {  
4         this.t = t ;  
5     }  
6     public void m() {f(t) ;}  
7  
8     public void f(Object o) {  
9         System.out.println("obj") ;  
10    }  
11    public void f(Integer i) {  
12        System.out.println("int") ;  
13    }  
14    public static void main(String[] args) {  
15        Node<Integer> n = new Node<Integer>(1) ;  
16        n.m() ;  
17    }  
18 }
```

- Affiche obj.
 - T est remplacé par Object dans la classe Node (donc t est de type Object) !

Erasure

- Conflit possible également entre méthodes paramétrées.

```
1 public class Conflit<E> {  
2     public void m(E e) {  
3     }  
4     public <T> void m(T o) {  
5     }  
6     public <U> void m(U o) {  
7     }  
8 }
```

- La borne de E, T et U est Object.
- Les trois méthodes ont la même signature après compilation : **conflit**.

Limitation de l'eraseure

Certaines opérations sur les variables de type ne sont pas sûres voir interdites (souvent à cause de l'*erasure*):

- implements/extends T (interdit)
- new T ou new T[] (interdit)
- T.class (interdit)
- instanceof T (interdit)

Cours 6 : Types et méthodes paramétrés

Types paramétrés

Wrappers

Erasure

Wildcard

Redéfinition

Sous-typage et type paramétré

- Sous-typage classe fonctionne sur les types paramétrés avec le même type argument.

```
1 ArrayList<String> al = new ArrayList<>();  
2 List<String> l4 = al; //ok sous typage
```

Sous-typage et type paramétré

- Mais pas si le type argument n'est pas le même !

```
1  ArrayList<String> al1 = new ArrayList<>() ;  
2  ArrayList<Object> al2 = al1 ; //compile pas  
3  
4  //si compilait :  
5  al2.add(new Integer(3)) ; //compile !! :(
```

Sous-typage et type paramétré - wildcards !

- Écriture pour faire du sous-typage : wildcard.

```
1  ArrayList<String> al1 = new ArrayList<>();  
2  ArrayList<?> al2 = al1; //compile  
3  
4  al2.add(new Integer(3)); //compile pas !  
5  //ClassCastException évité plus loin !
```

- `List<?>` : on hérite d'un type que l'on ne connaît pas.

Wildcards

- ▶ “?” n'est pas un type en tant que tel.
- ▶ **Représente** un type que l'on ne **connait pas** (que l'on ne veut pas connaître) en tant que **type argument** d'un type paramétré.
- ▶ Capture un type que l'on ne connaît pas.

```
1  ArrayList<String> l1= new ArrayList<>() ;  
2  l1.add("foo") ;  
3  ArrayList<?> l2=l1 ; // compile  
4  
5  Object object=l2.get(0) ; // compile
```

Bornes

- ▶ Deux sortes de wildcards :
 - ▶ `List< ? extends Type>` : liste d'un type qui est un **sous-type** de `Type`.
 - ▶ `List< ? super Type>` : liste d'un type qui est un **super-type** de `Type`.
- ▶ `List< ?>` est équivalent à `List< ? extends Object>`.
- ▶ Une seule borne possible.

Wildcards - intérêt ?

java.util

Interface Collection<E>

boolean addAll(Collection<? extends E> c)

```
1  Collection<Number> numberList = Arrays.<Number>asList(1, 2) ;
2  Collection<Object> objectList = Arrays.<Object>asList("bla", 5) ;
3  Collection<Integer> intList = Arrays.<Integer>asList(3, 4) ;
4
5  objectList.addAll(numberList) ; //ok
6  numberList.addAll(objectList) ; //compile pas
7  objectList.addAll(intList) ; //ok, on "simule" du sous typage !
8  numberList.addAll(intList) ; //ok, on "simule" du sous typage !
```

Wildcards - intérêt ?

copy

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

```
1  List<String> stringlist = Arrays.<String>asList("denis", "cornaz") ;  
2  List<Object> objectList = new ArrayList<>() ;  
3  
4  Collections.copy(objectList, stringlist) ; //ok  
5  Collections.copy(stringlist, objectList) ; //ko
```

Instanciation

- Wildcard est un type abstrait : pas possible de l'instancier.

```
1  ArrayList<? extends String> list= new ArrayList<? extends String>() ; //  
    interdit  
2  ArrayList<?> list2= new ArrayList<?>() ; // interdit  
3  ArrayList<?> list3= new ArrayList<String>() ; // ok
```

? ≠ ?

- Un wildcard est différent d'un autre wildcard !

```
1 public static void cutPaste(List<?> l1, List<?> l2) {  
2     l1.addAll(l2) ; //compile pas  
3     l2.clear() ;  
4 }
```

- Comment faire pour avoir deux types liés ?
- Variable de type !

```
1 public static <T> void cutPaste(List<T> l1, List<T> l2) {  
2     l1.addAll(l2) ;  
3     l2.clear() ;  
4 }
```

? extends Type

- On ne connaît pas le type de ?, mais on sait que c'est un sous-type de Type.

```
1 List<Integer> l1 = new ArrayList<>();  
2 l1.add(1);  
3 List<? extends Number> l2 = l1; //ok  
4 l2.add("miage"); //compile pas  
5 l2.add(3); //compile pas, ? p-e un Number autre que Integer ! (Double,...)  
6 Number n = l2.get(0); //ok  
7 l2.clear(); //ok
```

- List<? extends Type> : liste où l'on peut uniquement **sortir** des valeurs (ou clear) : **Lecture seule !**

? super Type

- On ne connaît pas le type de ?, mais on sait que c'est un super-type de Type.

```
1 List<Object> l1= new ArrayList<>() ;  
2 List<? super Number> l2=l1 ; // compile  
3 l2.add("toto") ; // interdit  
4 l2.add(3) ; // ok  
5 Number n=l2.get(0) ; // interdit  
6 Object o=l2.get(0) ; // ok, tout objet hérite de Object  
7 l2.clear() ; // ok
```

- List< ? super Type> : liste où l'on peut uniquement **ajouter** des valeurs (ou clear), et sortir que des Objects.

Cours 6 : Types et méthodes paramétrés

Types paramétrés

Wrappers

Erasure

Wildcard

Redéfinition

Redéfinition et méthodes paramétrées

- ▶ Rappel : il y a redéfinition entre deux méthodes si :
 - ▶ Toutes les deux paramétrées ou toutes les deux pas paramétrées.
 - ▶ Même signature après erasure.
 - ▶ Le type de retour de la méthode redéfinie est un sous-type de celui de la méthode originale.
- ▶ Sinon, surcharge ou conflit (par Erasure).

Redéfinition - type de retour

```
1 public class Foo <U extends Calendar>{  
2     public U bloub() {  
3         return null ;  
4     }  
5 }
```

```
1 public class Bar <T extends GregorianCalendar> extends Foo<T> {  
2     @Override  
3     public T bloub() {  
4         return null ;  
5     }  
6 }
```

- ▶ Redéfinition ?
- ▶ Oui car GregorianCalendar est un sous-type de Calendar

Redéfinition - Variables de type

```
1 public class Truc {  
2     public <E> void m(E e) {  
3     }  
4     public <E extends Number> void m2(E e) {  
5     }  
6 }
```

```
1 public class Muche extends Truc {  
2     @Override  
3     public <F> void m(F f) { //ok  
4     }  
5     @Override  
6     public <F extends Number> void m(F f) { //compile pas (n'Override pas) (penser  
7         au equals !)  
8     }  
9     @Override  
10    public <F extends Number> void m2(F f) { //ok  
11 }
```

- Redéfinition si les variables de type ont la **même borne**.