Cours 6 : Types et méthodes paramétrés

Types paramétrés

Wrappers

Erasure

Wildcard

Redéfinition

Cours 6 : Types et méthodes paramétrés

Types paramétrés

Wrappers

Erasure

Wildcard

Redéfinition

Problème de départ

- Liste d'objets.

```
1  List l = new ArrayList() ;
2  l.add("Mido") ;
3  l.add(new Integer(3)) ;
4
5  for(Object o :l) {
6    System.out.println((String)o) ;
7  }
```

- Compile ?

Problème de départ

- Liste d'objets.

```
1  List l = new ArrayList() ;
2  l.add("Mido") ;
3  l.add(new Integer(3)) ;
4
5  for(Object o :l) {
6    System.out.println((String)o) ;
7  }
```

- Compile ?
- Oui ! Liste d'objets.

Problème de départ

- Liste d'objets.

```
1  List l = new ArrayList() ;
2  l.add("Mido") ;
3  l.add(new Integer(3)) ;
4
5  for(Object o :l) {
6    System.out.println((String)o) ;
7  }
```

- Compile ?
- Oui ! Liste d'objets.
- Fonctionne ?

Problème de départ

- Liste d'objets.

```
1  List l = new ArrayList() ;
2  l.add("Mido") ;
3  l.add(new Integer(3)) ;
4
5  for(Object o :l) {
6    System.out.println((String)o) ;
7  }
```

- Compile ?
- Oui ! Liste d'objets.
- Fonctionne ?
- Non ! `ClassCastException`.

Problème de départ

- Problem ?

```
1  public static void copy(List dst, List src) {
2    for(Object element : src) {
3      dst.add(element) ;
4    }
5  }
6  public static void main(String[] args) {
7    List integerList = Arrays.asList(2, 3) ;
8    List stringList = Arrays.asList("Batch") ;
9    copy(stringList, integerList) ; //argh ?
10 }
```

Problème de départ

- Si `Object` est le type, difficile d'obtenir :
  - La généricité : ne pas ré-écrire le code pour les torchons, les serviettes etc.
  - Le typage statique : vérification par **le compilateur**.
- Besoin de paramétrer !

Solution de départ

Javadoc :

```
public interface List<E>
extends Collection<E>
```

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

```
1  List<String> l = new ArrayList<String>() ;
2  l.add("Mido") ;
3  l.add(new Integer(3)) ; //compile pas  !
4  for(String o :l) {
5    System.out.println(o) ; //plus de cast !
6  }
```

```
1  public static <E> void copy(List<E> dst, List<E> src) {
2    for(E element : src) {
3      dst.add(element) ;
4    }
5  }
6  public static void main(String[] args) {
7    List<Integer> integerList = Arrays.<Integer>asList(2, 3) ;
8    List<String> stringList = Arrays.<String>asList("Batch") ;
9    copy(stringList, integerList) ; // compile plus  ! !
10 }
```

Types paramétrés

- On peut déclarer depuis Java 5 :
  - Des types paramétrés.
  - Des méthodes paramétrées.
  - Des classes internes paramétrées.
- Connus par le **compilateur**.
- A l'exécution, pour la VM, pas de type paramétré.

Types paramétrés - vocabulaire

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

Types paramétrés - Déclaration

- 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) ;
```

Types paramétrés - Déclaration

- Pour une **méthode** paramétrée, la déclaration se fait **avant le type de retour**.

```
1 public static <E> void copy(List<E> src, List<E> dst) {
2   ...
3 }
```

Types paramétrés - Utilisation

- Type argument précisé :
  - Après le type/classe pour un type/classe paramétré.

```
1 List<Integer> l = new ArrayList<Integer>() ;
```

  - Après le . et avant le nom pour une méthode paramétrée.

```
1 Objects.<Integer>requireNonNull(o) ;
2 List<Integer> integerList = Arrays.<Integer>asList(1, 2) ;
```

Static

- Variable de type pas accessible dans un contexte `static` car **lié à une instance**.

```
1  public class Static<E> {
2    private E e ; //ok
3    private List<E> l ; //ok
4    private static E es ; //ko
5
6    private E blob(E e) { //ok
7      E e2 = e ; //ok
8      return e2 ;
9    }
10
11   private static E bloub(E e) { //ko
12     E e2 = e ; //ko
13     return e2 ;
14   }
15
16   private static <E> E bloum(E e) { //ok.. mais pas le même !
17     E e2 = e ;
18     return e2 ;
19   }
20 }
```

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 Djsdjklq <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
```

Bornes multiple

- Si borne multiple :
  - Vu par sous-typage comme chacune des bornes (méthodes, stockage dans un champ...).
  - Possède l'union des méthodes.

```
1  public class Dauphine<T extends Date & Closeable> {
2    private final Date d ;
3    private final Closeable c ;
4    private final T copy ;
5
6    public Dauphine(T t1, T t2) {
7      this.d = t1 ;
8      this.c = t2 ;
9      this.copy = t2 ;
10     t2.after(new Time(0)) ; //t2 vu comme un Date
11     t2.close() ; //t2 vu comme un Closeable
12     truc(c) ; //compile pas
13     truc(t2) ; //ok
14     truc(copy) ; //ok
15   }
16
17   private void truc(Date d) {
18   }
19 }
```

Cours 6 : Types et méthodes paramétrés

Types paramétrés

Wrappers

Erasure

Wildcard

Redéfinition

Types primitifs

- Pas possible d'utiliser un type primitif en tant qu'argument.

```
1 List<int> list = ... //int n'est pas un sous-type d'objet
```

- Comment faire ?

Types primitifs

- Pas possible d'utiliser un type primitif en tant qu'argument.

```
1 List<int> list = ... //int n'est pas un sous-type d'objet
```

- Comment faire ?
- Possible d'utiliser les wrapper et l'auto (un)boxing.

```
1 List<Integer> list = ...
2 list.add(3) ; //auto boxing
3 int v = list.get(0) ; //auto unboxing
```

- Pas de boxing/unboxing entre tableau de primitif et de wrapper.

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).

Apparté Wrappers

- Simplement un objet stockant un type primitif.

```
1  public final class Integer extends Number implements Comparable<Integer> {
2
3      /**
4       * The value of the {@code Integer}.
5       *
6       * @serial
7       */
8      private final int value ;
9
10     /**
11      * Constructs a newly allocated {@code Integer} object that
12      * represents the specified {@code int} value.
13      *
14      * @param  value  the value to be represented by the
15      *                {@code Integer} object.
16      */
17     public Integer(int value) {
18         this.value = value ;
19     }
```

- Boolean, Short, Integer, Float, Byte, Character, Long, Double.

Cours 6 : Types et méthodes paramétrés

Types paramétrés

Wrappers

Erasure

Wildcard

Redéfinition

Types paramétrés : comment ça marche ?

- Le compilateur traduit les types paramétrés et le code d'appel en **code classique** !
- Principe de l'**erasure**.

```
1  public class MyList<T> {
2    void add(T e) {}
3
4    T get(int index) {return null ;}
5
6    public static void main(String[]
           args) {
7      MyList<String> l=new MyList<
             String>() ;
8      l.add("bloub") ;
9      String s=l.get(0) ;
10   }
11 }
```

Types paramétrés : comment ça marche ?

- Le compilateur traduit les types paramétrés et le code d'appel en **code classique** !
- Principe de l'**erasure**.

```
1  public class MyList<T> {
2    void add(T e) {}
3
4    T get(int index) {return null ;}
5
6    public static void main(String[]
           args) {
7      MyList<String> l=new MyList<
             String>() ;
8      l.add("bloub") ;
9      String s=l.get(0) ;
10   }
11 }
```

```
1  public class MyList {
2      public MyList() {
3          super() ;
4      }
5      void add(Object e) {}
6      Object get(int index) {
7          return null ;
8      }
9
10     public static void main(String[]
             args) {
11         MyList l = new MyList() ;
12         l.add("bloub") ;
13         String s = (String)l.get(0) ;
14     }
15 }
```

- Après compilation (vu avec `javac -XD-printflat -d dir MyList.java`)

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    private void f(Object o) {
8      System.out.println("obj") ;
9    }
10   private void f(Integer i) {
11     System.out.println("int") ;
12   }
13   private void f(T t) {
14     System.out.println("t") ;
15   }
16   public static void main(String[] args) {
17     Node<Integer> n = new Node<Integer>(1) ;
18     n.m() ;
19   }
20 }
```

Affiche ?

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    private void f(Object o) {
8      System.out.println("obj") ;
9    }
10   private void f(Integer i) {
11     System.out.println("int") ;
12   }
13   private void f(T t) {
14     System.out.println("t") ;
15   }
16   public static void main(String[] args) {
17     Node<Integer> n = new Node<Integer>(1) ;
18     n.m() ;
19   }
20 }
```

- Compile pas !
  - `f(Object)` et `f(T)` ont la même signature après erasure !

Affiche ?

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 ?

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 ?

- 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**.

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  ! !  :(
6  String s = al1.get(0) ; //Aie, ClassCastException...
```

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 evité 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 connaitre) en tant que **type argument** d'un type paramétré.
- Capture un type que l'on ne connait pas.

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

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.
  - (et exceptions, visibilité etc).
- 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 ?

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` (donc par Erasure, ok).

Redéfinition - Variables de type

```java
public class Truc {
  public <E> void m(E e) {
  }
  public <E extends Number> void m2(E e) {
  }
}
```

```java
public class Muche extends Truc {
  @Override
  public <F> void m(F f) { //ok
  }
  @Override
  public <F extends Number> void m(F f) { //compile pas (n'Override pas)

  }
  @Override
  public <F extends Number> void m2(F f) { //ok
  }
}
```

- Redéfinition si les variables de type ont la **même borne**.
- Ne fonctionne pas si sous-typage entre les bornes.

Pour conclure

- Utiliser des types et méthodes paramétrés lorsque :
  - On utilise des types paramétrés déjà définis.
  - On veut vérifier une contrainte de type entre des paramètres.
  - On veut récupérer un objet de même type que celui stocké.