

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 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.

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);
```

Types paramétrés - Utilisation

- ▶ Peut demander au compilateur de deviner les types arguments (Java 7).
 - ▶ Pour une classe paramétrée, syntaxe diamond.

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

- ▶ Pour une méthode paramétrée :
 - ▶ Selon les arguments (ici de `asList` : `Integer`) :

```
1 List<Integer> l2 = Arrays.asList(1,2);
```

- ▶ Selon le contexte (ici dans une list d'integers) :

```
1 List<Integer> l2 = Collections.emptyList();
```

Types paramétrés - Utilisation

- ▶ Parfois impossible à deviner pour le compilateur.

```
1 public static void truc(List<Integer> l) {  
2     }  
3  
4 public static void truc(Collection<String> l) {  
5     }  
6  
7 public static void main(String[] args) {  
8     truc(Collections.emptyList()); //compile pas  
9     truc(Collections.<Integer>emptyList()); //ok  
10    truc(Collections.<String>emptyList()); //ok  
11 }
```


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 Djsdjkqlq <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 }
```

Borne réursive

- ▶ Possible de déclarer une borne de variable de type paramétré par la variable déclarée.

```
1 public interface Comparable<T extends Comparable<T>> {  
2     public boolean smallerThan(T t);  
3 }
```

```
1 public class MyLong implements Comparable<MyLong> {  
2     private final long val;  
3  
4     public MyLong(long val) {  
5         this.val = val;  
6     }  
7  
8     @Override  
9     public boolean smallerThan(MyLong t) {  
10        return val < t.val;  
11    }  
12 }
```

- ▶ Permet de spécifier que le type est comparable avec lui même.

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.

Apparté Wrappers - Pourquoi les primitifs alors ?

- ▶ Types primitifs sur la pile, wrapper objet donc sur le tas (référence vers l'objet sur la pile).
- ▶ Donc **les wrappers sont plus couteux** !
 - ▶ Plus gros en mémoire (header de l'objet, alignement en mémoire).
 - ▶ Travail pour le GC.
- ▶ Ne pas utiliser de wrapper en tant que champ d'un objet.

Apparté Wrappers - Attention 1

- ▶ Ne pas faire de test `==` ou `!=` sur les wrappers.

```
1 Integer i1 = 10 ;  
2 Integer i2 = 10 ;  
3 Integer i3 = new Integer(10) ;  
4 Integer i4 = 500 ;  
5 Integer i5 = 500 ;  
6  
7 System.out.println(i1 == i2) ;  
8 System.out.println(i1 == i3) ;  
9 System.out.println(i4 == i5) ;
```

- ▶ Sortie ?

Apparté Wrappers - Attention 1

- ▶ Ne pas faire de test `==` ou `!=` sur les wrappers.

```
1 Integer i1 = 10 ;
2 Integer i2 = 10 ;
3 Integer i3 = new Integer(10) ;
4 Integer i4 = 500 ;
5 Integer i5 = 500 ;
6
7 System.out.println(i1 == i2) ;
8 System.out.println(i1 == i3) ;
9 System.out.println(i4 == i5) ;
```

- ▶ Sortie ?

1. True
2. False
3. False

Apparté Wrappers - Attention 2

- ▶ Attention au wrapper null et l'unboxing.

```
1 Integer i = null ;  
2 if (new Random().nextBoolean())  
3     i = 1 ;  
4 int val = i ; // risque de NPE
```

- ▶ Pas facile à déboguer !

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[]
7         args) {
8         MyList<String> l=new MyList<
9             String>();
10        l.add("bloub");
11        String s=l.get(0);
12    }
13 }
```

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[]
7         args) {
8         MyList<String> l=new MyList<
9             String>();
10        l.add("bloub");
11        String s=l.get(0);
12    }
13 }
```

```
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[]
11        args) {
12        MyList l = new MyList();
13        l.add("bloub");
14        String s = (String)l.get(0);
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 obj.
 - ▶ T est remplacé par Object dans la classe Node (donc t est de type Object) !

Affiche ?

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**.

Précision tableaux

- ▶ Interdiction de créer des tableaux de type paramétrés.
- ▶ L'**erasure** transformerai un `E[]` en `Object[]`.
- ▶ Le compilateur ne pourrai **pas** garantir qu'il y a **que des E** dans le tableau !

Précision tableaux

- ▶ Interdiction de créer des tableaux de type paramétrés.
- ▶ L'**erasure** transformerai un `E[]` en `Object[]`.
- ▶ Le compilateur ne pourrai **pas** garantir qu'il y a **que des E** dans le tableau !
- ▶ Voulu :

```
1   Object[] a = new String[5] ;  
2   a[0] = "bla" ;  
3   a[1] = 5 ; //lève ArrayStoreException
```

Précision tableaux

- ▶ Interdiction de créer des tableaux de type paramétrés.
- ▶ L'**erasure** transformerai un `E[]` en `Object[]`.
- ▶ Le compilateur ne pourrai **pas** garantir qu'il y a **que des E** dans le tableau !
- ▶ Voulu :

```
1 Object[] a = new String[5] ;
2 a[0] = "bla" ;
3 a[1] = 5 ; //lève ArrayStoreException
```

- ▶ Si c'était possible :

```
1 public void m(E e) {
2     Object[] a = new E[5] ;
3     a[0] = e ;
4     a[1] = "bla" ; //aucun moyen de lever ArrayStoreException si E était par
                    //exemple Integer car transformé en Object
5 }
```

Cast et type paramétré

- ▶ Type argument perdu à la compilation → la VM ne peut pas vérifier si le type argument est correct à l'exécution !
- ▶ Un problème peut survenir **plus tard**, i.e. là où il n'y a pas de cast !

```
1 ArrayList<String> l=new ArrayList<String>() ;
2 l.add("miage") ;
3 Object o=l ;
4 ArrayList<Integer> l2=(ArrayList<Integer>)o ; // unsafe cast mais compile et s'
    execute.
5
6 int i = l2.get(0) ; //ClassCastException alors que pas de cast !
```

- ▶ Compilateur ne peut garantir l'absence d'erreur de transtypage → warning Unsafe cast
- ▶ Garantie l'absence de ClassCastException aux endroits sans cast si pas de warning.

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 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 ? 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 !
```

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
```

Wildcards - intérêt ?

```
1 public static void methodeQuiVeutQueDesNumbers(List<Number> l) {
2 }
3
4 public static void methodeQuiVeutQueDesNumbers2(List<? extends Number> l) {
5 }
6
7 public static void main(String[] args) {
8     methodeQuiVeutQueDesNumbers(new ArrayList<Number>()); //ok
9     methodeQuiVeutQueDesNumbers(new ArrayList<Integer>()); //compile pas !
10
11     methodeQuiVeutQueDesNumbers2(new ArrayList<Number>()); //ok
12     methodeQuiVeutQueDesNumbers2(new ArrayList<Integer>()); //ok
13     methodeQuiVeutQueDesNumbers2(new ArrayList<Double>()); //ok
14 }
```

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 ?

? ≠?

- ▶ 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 }
```

? ≠?

► Ou mieux :

```
1 //public static <T> void cutPaste(List<? super T> l1, List<T> l2) {
2 //public static <T> void cutPaste(List<T> l1, List<? extends T> l2) {
3 public static <T> void cutPaste(List<? super T> l1, List<? extends T> l2) {
4     l1.addAll(l2);
5     l2.clear();
6 }
7 public static void main(String[] args) {
8     List<Object> l1 =...
9     List<Integer> l2=...
10    List<String> l3=...
11    cutPaste(l1, l2); //ok
12    cutPaste(l2,l3); //compile pas
13 }
```


Capture

- Capture : mécanisme pour considérer que $T = ?$ lors de l'appel.

```
1 public class MyList<E> {
2     public static <T> List<T> reverse(List<T> list) {...}
3     public static <T> List<T> union(List<T> l1, List<T> l2) {...}
4
5     public static void main(String[] args) {
6         List<Integer> l1= new ArrayList<>();
7         List<Integer> l2=reverse(l1);
8         List<Integer> l3=union(l1,l2);
9         List<?> l4=l1;
10        List<?> l5=reverse(l4); //ok, capture
11        List<?> l6=union(l4,l5); //pas capture, le ? de l4 != du ? de l5
12    }
13 }
```

? 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 !**

? extends Type

- ▶ Ce n'est **pas** “N'importe quoi qui est un sous-type de Type”, mais “Quelque chose que j'ignore qui est un sous-type de Type”

```
1 //Fille et Fils implements Pere
2
3 List<Pere> list = new ArrayList<>();
4 list.add(new Fille()); //ok
5 list.add(new Fils()); //ok
6 Pere t = list.get(0); //ok
```

```
1 List<Fils> list = new ArrayList<>();
2 List<? extends Pere> list2 = list; // ok
3 list2.add(new Fille()); // pas ok !
4 //Si c'était ok (car Fils est un Pere)
5 Fils b = list.get(0); // Problème ! Une List<Fils> va me retourner une Fille !!
```

? 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.

Super ou Extends ?

- ▶ Moyen mnémotechnique : **PECS**
 - ▶ (Producer Extends, Consumer Super)
 - ▶ Pour un producteur (get), utiliser extends
 - ▶ Pour un consommateur (add), utiliser super

Super ou Extends ?

- ▶ Moyen mnémotechnique : **PECS**
 - ▶ (Producer Extends, Consumer Super)
 - ▶ Pour un producteur (get), utiliser extends
 - ▶ Pour un consommateur (add), utiliser super
 - ▶ Si les deux : ne pas utiliser les wildcards (mais le type explicite)

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

```
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**.
- ▶ 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.

Pour conclure

- ▶ Tout n'est pas paramétré.

```
1 <S extends CharSequence> void idiot(S s) {  
2 }
```

Idiot ?

Pour conclure

- ▶ Tout n'est pas paramétré.

```
1 <S extends CharSequence> void idiot(S s) {  
2 }
```

Idiot ?

```
1 void moinsIdiot(CharSequence s) {  
2 }
```

- ▶ Utiliser le sous-typage classique !