

# Type et méthode paramétré

Rémi Forax  
forax@univ-mlv.fr

# Pourquoi ?

Les collections sont des collections d'objets, on peut donc stocker n'importe quel objet dedans mais lorsque l'on sort un objet d'une collection, il faut utiliser un cast car le compilateur ne sait pas que seul des String sont stockés

```
public static void main(String[] args) {  
    List l = new ArrayList();  
    l.add("toto"); // ok add(Object)  
    String s2 = (String)l.get(0); // cast obligatoire  
}
```

# Pourquoi ?

De plus, il n'y a pas de moyen de spécifier des contraintes vérifiant que l'on ne fait pas n'importe quoi

```
public static void copy(List dst, List src) {  
    for(Object o: src) {  
        dst.add(o);  
    }  
}  
...  
List stringList = Arrays.asList("foo", "bar");  
List integerList = Arrays.asList(2, 3);  
copy(stringList, integerList);    // ok, compile aaaaahh
```

# Type et méthode paramétré

Ajouter une ou des contraintes de type qui seront vérifiées par le compilateur

```
interface List<T> { ... }
class ArrayList<T> implements List<T> { ... }
...
List<String> stringlist = new ArrayList<String>();
stringlist.add("foo");
String s = stringlist.get(0); // plus de cast, cool !

public static <E> void copy(List<E> dst, List<E> src) {
    for(E element: src) {
        dst.add(element);
    }
}
...
List<Integer> integerList = Arrays.<Integer>asList(2, 3);
copy(stringList, integerList); // compile plus !!
```

# Declaration de variable de type

On déclare une variable de type

après le nom pour une classe paramétré

```
interface Map<K, V> { ... }  
class HashMap<K, V> implements Map<K, V> { ... }
```

avant le type de retour pour une méthode paramétrée

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

# Quantificateur

Déclarer une variable de type pour une classe ou une méthode ne signifie pas la même chose

Pour une classe, **il existe un T** tel que

```
public class Stack<T> {  
    public void push(T t) {...}  
    public T pop() { ... }  
}
```

Pour une méthode, **quelque soit T** tel que

```
public class Utils {  
    public <T> void transfer(Stack<T> s1, Stack<T> s2) { ... }  
}
```

# Bornes de variable de type

Une variable de type peut avoir une ou plusieurs bornes (séparé par &)

dans ce cas, la variable de type est typé comme sa borne

```
static <T extends Object> T requireNonNull(T object) {  
    if (object == null) { throw ... }  
    return object;  
}
```

mais une variable de type n'a pas la visibilité de sa borne

```
static <F extends Closeable & CharSequence> F bar(F foo) {  
    foo.charAt(0); // comme un CharSequence  
    foo.close();   // comme un Closeable  
    ...  
}
```

# Variable de type et static

Les variables de type des types paramétrés ne sont pas accessible dans un contexte static car elles sont liées à une instance

```
class Foo<E> {  
    E e; // ok  
    List<E> list; // ok  
    E foo(E e) { return e; } // ok  
    void foo2() { E e; } // ok  
  
    static E e; // non  
    static E bar(E e) { return e; } // non  
    static void bar2() { E e; } // non  
}
```



# Generics vs Template

En C++, les types paramétrés génèrent un code par instantiation.

En C# ou en Java, il n'y a qu'un code pour toutes les instantiations, il n'est donc pas possible d'hériter/d'implanter une variable de type

```
class Cool<T> extends T { // erreur
    ...
}
```

# Utilisation des types arguments

On utilise un type argument

après le type/la classe pour un type/une classe paramétrée

```
List<String> stringlist = new ArrayList<String>();
```

après le '.' et avant le nom pour une méthode paramétrée

```
List<Integer> integerList = Arrays.<Integer>asList(2, 3);
```

# Héritage figé

On peut aussi hériter d'un type paramétré instancié avec un type argument

```
class StringComparator implements Comparator<String> {  
    public int compare(String s1, String s2) {  
        return - s1.compareToIgnoreCase(s2);  
    }  
}
```

ou avec la syntaxe des classes anonymes

```
Comparator<String> comparator = new Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return - s1.compareToIgnoreCase(s2);  
    }  
};
```

# Borne récursive

Il est possible de déclarer une borne d'une variable de type paramétré par la variable déclarée

```
public interface Orderable<T extends Orderable<T>> {  
    public boolean lessThan(T t);  
}  
  
public class MyInteger implements Orderable<MyInteger> {  
    private final int value;  
    public MyInteger(int value) {  
        this.value=value;  
    }  
    public boolean lessThan(MyInteger i) {  
        return value<i.value;  
    }  
}
```

Cela permet de spécifier que le type est ordonnable avec lui-même

# Inférence des types arguments

On peut demander au compilateur de calculer/deviner les types arguments tout seul

pour une classe paramétrée (syntaxe diamand)

```
List<String> stringlist = new ArrayList<>();
```

ne marche pas avec la syntaxe des classes anonymes

– pour une méthode paramétrée

- en fonction des arguments

```
List<Integer> integerList = Arrays.asList(2, 3);
```

- en fonction du contexte

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

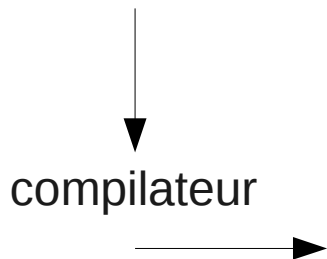
```
List<Integer> foo() { return Collections.emptyList(); }
```

marche pas dans un appel de méthode (Java 8?)

# Implantation en Java (erasure)

Java implante les generics en supprimant les types argument lors de la compilation et remplace les variables de type par leur borne.

```
class AsStringComparator<T> implements Comparator<T> {  
    public int compare(T t1, T t2) {  
        return t1.toString().compareTo(t2.toString());  
    }  
}  
AsStringComparator<Integer> c = new AsStringComparator<>();  
c.compareTo(2, 3);
```



```
class AsStringComparator implements Comparator {  
    public int compare(Object t1, Object t2) {  
        return t1.toString().compareTo(t2.toString());  
    }  
}  
AsStringComparator c = new AsStringComparator();  
c.compareTo(2, 3);
```

Les types paramétrés n'existent donc pas à l'exécution !

# Erasure et name clash

Comme les types arguments sont supprimés et les variables de types sont remplacés par leur borne, cela peut créer des conflits

```
class Foo {  
    void bar(List<String> list) { ... }  
    void bar(List<Integer> list) { ... } // même erasure (List)  
}
```

```
class Foo2<T extends CharSequence> {  
    void bar(T element) { ... } // borne=CharSequence  
    void bar(CharSequence seq) { ... }  
}
```

# Limitation de l'erasure

Les opérations dynamiques nécessitant les classes lors de l'exécution sont donc interdites sur les variables de types et les types paramétrés

- instanceof T ou instanceof Foo<String>
- new T
- new T[] ou new Foo<String>[]
- class Foo<T> extends Throwable {}  
car catch(Foo<String>) impossible

et un cast (T) ou (Foo<String>) produit un warning



# Type paramétré et cast

Le type argument est **perdu** après la compilation, la VM ne peut donc le vérifier lors de **l'exécution**

```
public static void main(String[] args) {  
    ArrayList<String> l = new ArrayList<>();  
    Object o = l;  
    ArrayList<String> l2 = (ArrayList<String>) o; // unsafe cast  
}
```

Un problème peut arriver **plus tard** et à un endroit où aucun cast n'est écrit

# Type paramétré et cast (2)

Le compilateur **ne peut garantir** que le code s'exécutera sans erreur de transtypage

```
public static void main(String[] args) {  
    ArrayList<String> l = new ArrayList<>();  
    l.add("toto");  
  
    Object o = l;  
    ArrayList<Integer> l2 = (ArrayList<Integer>) o; // unsafe cast  
    int i = l2.get(0); // ClassCastException Integer  
}
```

L'exception **ClassCastException** est généré lors d'un **accès** et non lors de l'**unsafe cast**

# *Unchecked warnings ?*

Indique des casts ou conversions impliquant des *generics*. A cause de l'abrasion, ils ne sont pas vérifiables par la VM

Il existe deux sortes d'unchecked warnings

- unchecked conversion

```
ArrayList<String> list = new ArrayList<String>();  
ArrayList raw = list;
```

- unsafe cast

```
Object o = new ArrayList<String>();  
ArrayList<String> list = (ArrayList<String>) o;
```

# Raw Type

On appelle *raw type* un type paramétré utilisé sans paramètre. Ce type permet la compatibilité avec du code existant (écrit avant la 1.5)

Règles de conversion :

```
List list = new List<String>();
```

- Type<T> vers RawType

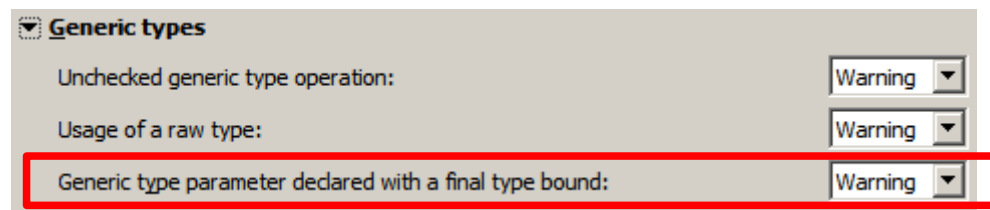
```
List<String> list = new List(); // unchecked conversion
```

- RawType vers Type<T>

# Utilisation des *Raw Type*

On utilise des raw type **uniquement** par compatibilité avec un code écrit avant la version 1.5

Eclipse possède un flag pour émettre un warning en cas d'utilisation de raw types



La version 1.7 de javac possède le même flag.

# *Raw Type* et conversion

- Attention, utiliser un raw type n'est pas sûr, le code peut lever un `ClassCastException`

```
public class Holder<T> {  
    public Holder(T t) {  
        this.t=t;  
    }  
    public T get() {  
        return t;  
    }  
    private final T t;  
}
```

- L'exception n'est pas générée à l'endroit de la conversion !!

```
public static void main(String[] args) {  
    Holder<String> holder=new Holder<String>("toto");  
    Holder rawHolder=holder;  
    Holder<String> newHolder=rawHolder;    // unchecked conversion  
    Holder<Integer> newHolder2=rawHolder;  // unchecked conversion  
  
    System.out.println(newHolder2.get()+1);  
    // ClassCastException at RawType.main(RawType.java:33)  
}
```

# @SuppressWarnings

Si le compilateur ne génère pas d'*unchecked warning* alors le code utilisant les *generics* ne peut pas lever de **ClassCastException**

L'annotation

**@SuppressWarnings("unchecked")** indique au compilateur de ne pas lever de *warning* pour raison de conversions non vérifiées

# @SuppressWarnings

Se met sur les champs, les variables locales, les méthodes

```
public static List<String> badIdentity(List<String> o) {  
    @SuppressWarnings("unchecked")  
    List<String> list = o; // unsafe cast mais pas reporté  
    return list;  
}
```

**Attention**, il ne faut utiliser @SuppressWarnings que si le code est **sur au niveau typage** mais que le compilateur ne peut le garantir



# @SuppressWarnings et code sûr

Ne pas utiliser @SuppressWarnings si le code n'est pas sûr car cela cache une CCE qui sera levée plus tard

```
@SuppressWarnings("unchecked")
public static <T> T unsafeCast(Object o) {
    return (T)list; // unsafe cast mais pas reporté
}

public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<String>();
    Arrays.addAll(list, args);
    List<Integer> list2 = unsafeCast(list);
    int i = list.get(0); // ClassCastException à l'exécution !!!!
}
```

# Exemple de code sûr

Il n'est pas possible de créer un tableau de variable de type (**new T[]**) mais

```
public class FixedSizeArray<T> {  
    private final T[] array;  
  
    @SuppressWarnings("unchecked")  
    public FixedSizeArray(int capacity) {  
        Array = (T[])new Object[capacity]; // unsafe cast  
    }  
    public T get(int index) {  
        return array[index];  
    }  
    ...  
}
```

Il est possible de créer un tableau d'**Object** et de le transformer en tableau de T, mais il y a mieux

# Variable de type et tableau

On préférera plutôt ce code car il reste vrai même si les generics sont réifiés un jour

```
public class FixedSizeArray<T> {  
    private final Object[] array;  
  
    public FixedSizeArray(int capacity) {  
        array=new Object[capacity];  
    }  
  
    @SuppressWarnings("unchecked")  
    public T get(int index) {  
        return (T)array[index];           // unsafe cast  
    }  
  
    public void set(int index,T value) {  
        array[index] = value;             // ok sous-typage  
    }  
}
```

# Exemple de code sûr

edges contient un tableau de type paramétré ce qui est unsafe mais il n'est pas visible et pas exporté, donc Ok !

```
public class EdgeList<E> {  
    private final Edge<E>[] edges;  
  
    @SuppressWarnings("unchecked")  
    public EdgeList(int capacity) {  
        edges = (Edge<E>[])new Edge<?>[capacity];  
    }  
  
    public void set(int index, Edge<E> edge) {  
        edges[index] = edge;  
    }  
  
    public Edge<E> get(int index) {  
        return edges[index];  
    }  
}
```

# Type token

A cause de l'erasure, la VM ne connaît pas le type argument d'une variable de type donc

```
public static <T> T create() throws Exception {  
    return new T(); // illégal  
}  
...  
String s = Factory.<String>create();
```

Mais en Java, une classe est un objet de la classe `java.lang.Class`, donc on peut passer un type argument en paramètre supplémentaire

# Class<T>

Permet de passer un type argument en paramètre qui non disponible à l'exécution à cause de l'abrasion.

```
public static <T> T create(Class<T> clazz) throws Exception {  
    return clazz.newInstance(); // ok  
}  
...  
String s = create(String.class); // ok  
  
int i = create(Integer.class);    // InstantiationException  
                                // le constructeur Integer()  
                                // n'existe pas
```

# Class<T>

La syntaxe **.class** permet d'obtenir la classe correspondant à un type non paramétré

La syntaxe **.class** ne marche pas sur un type non réifié et même sur un wildcard non borné

```
public static void main(String[] args) {  
    Class<String> c1 = String.class;    // ok  
    Class<Integer> c2 = Integer.class;  // ok  
  
    Class<? extends Number> cn = c2;    // ok  
  
    List<?>.class                       // illégal  
    List<String>.class                  // illégal  
    List<? extends String>.class       // illégal  
}
```

# Class<T> et instanceof

**Class.isInstance()** test dynamiquement la classe

**Class.cast()** effectue le cast ou renvoie une **ClassCastException**

```
public static <T> List<T> filter(List<?> list) {  
    List<T> tList = new ArrayList<T>();  
    for(Object o: list)  
        if (o instanceof T) // illégal  
            tList.add((T)o); // unsafe cast  
    return tList;  
}
```

```
public static <T> List<T> filter(List<?> list, Class<T> clazz) {  
    List<T> tList = new ArrayList<T>();  
    for(Object o: list)  
        if (clazz.isInstance(o)) // ok  
            tList.add(clazz.cast(o)); // safe cast !!  
    return tList;  
}
```



# java.lang.reflect.Array

Permet de créer dynamiquement des tableaux en fonction d'un objet `Class<T>`

```
public static <T> T[] toArray(List<? extends T> list) {  
    T[] array = new T[list.size()]; // illégal  
    for(int i = 0; i < list.size(); i++)  
        array[i] = list.get(i);  
    return array;  
}
```

```
@SuppressWarnings("unchecked")  
public static <T> T[] toArray(List<? extends T> list, Class<T> clazz) {  
    T[] array = (T[])Array.newInstance(clazz, list.size()); // unsafe  
    for(int i = 0; i < list.size(); i++)  
        array[i] = list.get(i);  
    return array;  
}
```

# Object.getClass()

- Permet d'obtenir un objet Class représentant la classe d'un objet particulier

```
String s1="toto";  
String s2="tutu";  
s1.getClass()==s2.getClass(); // true
```

- Il existe une règle spéciale du compilateur indiquant le type de retour de getClass() :

`Class<? extends erasure<T>> T.getClass()`

# Object.getClass()

## Utilisation de getClass() et .class

```
Class<? extends String> sClazz = "toto".getClass();
Class<? extends ArrayList> aiClazz =
    new ArrayList<Integer>().getClass();

String s = sClazz.newInstance();                // ok
ArrayList<Integer> ai =
    (ArrayList<Integer>)aiClazz.newInstance(); // warning, raw type
                                              // conversion
```

Attention !, cette règle :

- introduit un *raw type* au lieu d'un *wildcard*
- ne tient pas compte des classes **final** comme String

# Generics et sous-typage

Une liste d'oranges n'est pas une liste de fruits !  
Supposons que si:

```
ArrayList<Orange> oranges = new ArrayList<>();  
ArrayList<Fruit> fruits = oranges; // marche pas normalement  
fruits.add(new Banana()); // ok, Banana est sous type de Fruit  
                        // mais  
Orange orange = oranges.get(0); // maintenant c'est une Orange !
```

donc il n'y a pas de sous-typage entre une  
List<Orange> et une List<Fruit>

# Wildcard

Comme c'est embêtant, Java a inventé une syntax pour permettre d'exprimer le sous-typage

On peut réduire un type à n'être que producteur

```
ArrayList<Orange> oranges = new ArrayList<>();  
ArrayList<? extends Fruit> fruits = oranges; // ok  
Fruit fruit = fruits.get(0); // ok  
fruits.add(new Banana()); // compile pas  
fruits.add(null); // ok
```

ou réduire un type à n'être que consommateur

```
ArrayList<Fruit> fruits = new ArrayList<>();  
ArrayList<? super Orange> oranges = fruits; // ok  
oranges.add(new Orange()); // ok  
Orange orange = oranges.get(0); // compile pas  
Object object = oranges.get(0); // ok
```

# La notation ?

- Un Wildcard ? représente un type que l'on ne veut pas connaître et qui est sous-type (extends) ou est super-type (super) d'un type spécifié
- ? n'existe qu'entre '<' et '>', ce n'est pas un type en lui-même
- Il permet de faire du sous-typage avec les generics
- PECS: producer extends consumer super

# Wildcard en tant que type

- “?” n'est pas un type en tant que tel, il **représente** juste un type que l'on ne connaît pas en tant que type argument d'un type paramétré

```
public class GenericSubTyping {  
    public static void main(String[] args) {  
        List<String> l1=...  
        l1.add("toto");  
        List<?> l2=l1;           // compile  
        ? object=l2.get(0);      // illégal, "?" N'est pas un type  
        Object object=l2.get(0); // compile  
    }  
}
```

- On dit que ? capture un type que l'on ne connaît pas

# Wildcard (2)

A quoi cela sert ?

```
interface Collection<E> {  
    public boolean addAll(Collection<? extends E> c);  
}  
...  
Collection<String> stringlist = Arrays.<String>asList("foo", "bar");  
Collection<Object> objectList = new ArrayList<>();  
ObjectList.addAll(stringlist);
```

```
static <E> void copy(List<? super E> dst, List<? extends E> src) {  
    for(E element: src) {  
        dst.add(element);  
    }  
}  
...  
List<String> stringlist = Arrays.<String>asList("foo", "bar");  
List<Object> objectList = new ArrayList<>();  
Collections.<CharSequence>copy(objectList, stringlist);
```



# Wildcard (3)

Comme on est gentil avec celui qui nous appelle on renvoie jamais un type paramétré par un wildcard

```
static <E> List<E> filter(List<? extends E> list,
                          Predicate<? super E> filter) {
    ArrayList<E> newList = new ArrayList<>();
    for(E element: list) {
        if (filter.accept(element)) {
            newList.add(element);
        }
    }
    return newList;
}
...
List<Object> list = Foo.filter(Arrays.asList("foo", "bar"),
    new Filter<CharSequence>() {
        public boolean accept(CharSequence seq) {
            return seq.length() % 2 == 0;
        }
    });
```

# Wildcard non borné

Le wildcard non borné `List<?>` est réifié !  
donc il est possible d'écrire

- `instanceof List<?>` ou `instanceof List<?>`
- `new List<?>[]`
- et `(List<?>)` ou `(List<?>[])` produit pas de warning

# Instantiation d'un *wildcard*

Un wildcard correspond à un type abstrait, il n'est donc pas instantiable

```
public static void main(String[] args) {  
    ArrayList<? extends String> list1 =  
        new ArrayList<? extends String>(); // illégal  
    ArrayList<?> list2 =  
        new ArrayList<?>(); // illégal aussi  
    ArrayList<?> list2 =  
        new ArrayList<>(); // encore illégal  
  
    ArrayList<ArrayList<?>> list3 =  
        new ArrayList<ArrayList<?>>(); // ok  
}
```

par contre, `ArrayList<ArrayList<?>>` est instantiable

# Appel explicite de méthode

Il n'est pas possible d'utiliser le “?” en tant que type lors d'un appel explicite de méthode car cela revient à l'instancier.

```
public class Collections {  
    public static <T> List<T> emptyList() {...}  
    ...  
}  
...  
public static void main(String[] args) {  
    list = Collections.<?>emptyList();           //illégal  
    list = Collections.<? extends String>emptyList(); //illégal  
    list = Collections.<? super Number>emptyList(); //illégal  
}
```

# Wildcard avec différents “?”

Le “?” d'une List<?> n'est pas égal au “?” d'une autre List<?>

```
public class List<E> {  
    public void add(E e) {...}  
    public E remove(int index) {...}  
  
    public static void move(List<?> l1, List<?> l2) {  
        return l1.add(l2.remove(0)); // compile pas  
    }  
  
    public static void main(String[] args) {  
        List<Integer> l1=...  
        System.out.println(move(l1, l1));  
        List<String> l2=...  
        System.out.println(move(l1, l2));  
    }  
}
```

# Wildcard avec différents “?”

Pour dire que deux types sont liés, on utilise une variable de type

```
public class List<E> {  
    public void add(E e) {...}  
    public E remove(int index) {...}  
  
    public static <T> void test(List<T> l1, List<T> l2) {  
        l1.add(l2.remove(0)); // ok  
    }  
  
    public static void main(String[] args) {  
        List<Integer> l1=...  
        System.out.println(move(l1, l1)); // ok  
        List<String> l2=...  
        System.out.println(move(l1, l2)); // error  
    }  
}
```

# Sous-typage entre *wildcard*

## Les relations de sous-typage entre *wildcards*

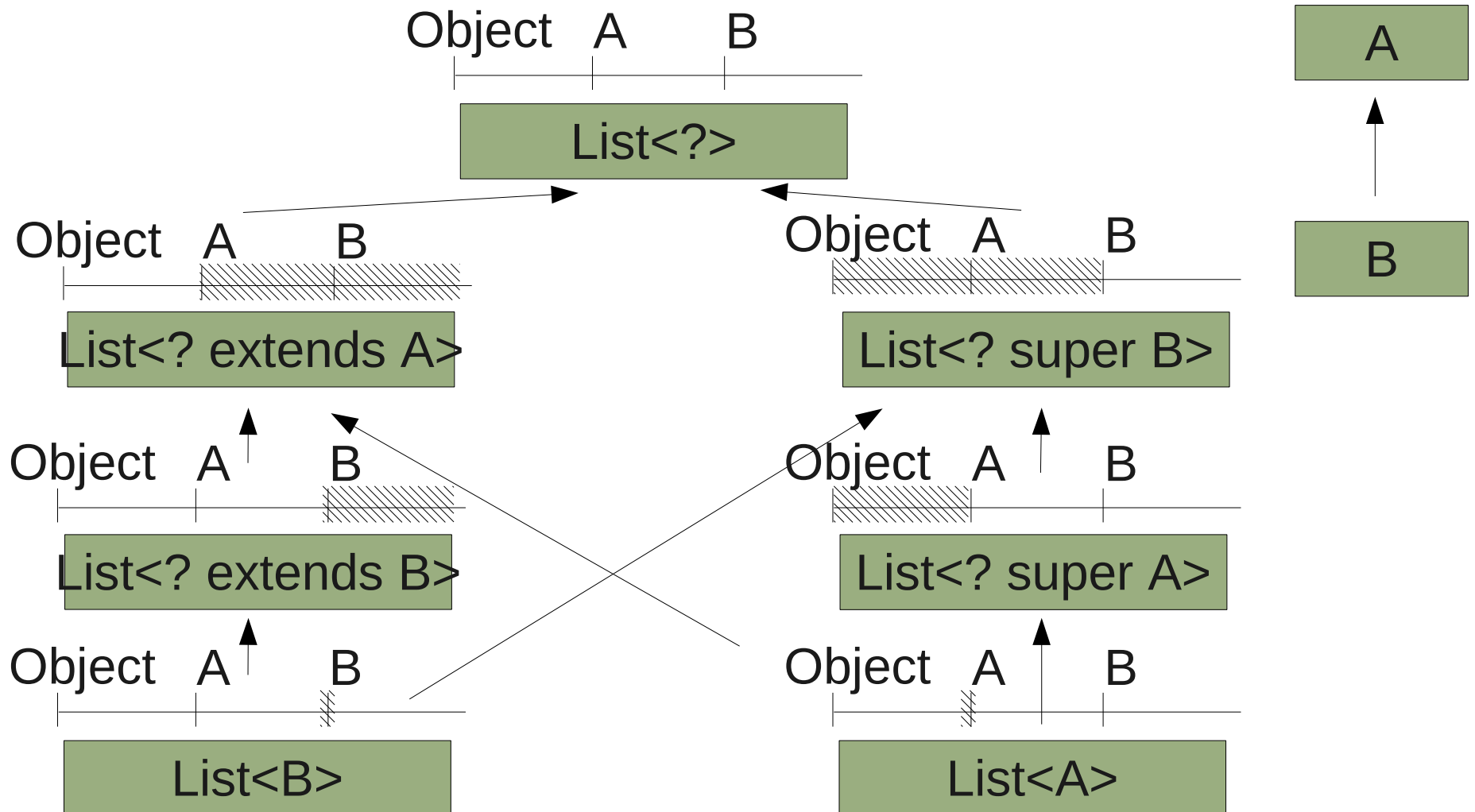
```
public static void main(String[] args) {  
    List<? extends String> l1=new ArrayList<String>();  
    List<? extends CharSequence> l2=l1;  
    List<?> l3=l2;  
  
    List<? super CharSequence> l4=new ArrayList<Object>();  
    List<? super String> l5=l4;  
    List<?> l6=l5;  
}
```

List<?> ► List<? extends CharSequence> ► List<? extends String>

List<?> ► List<? super String> ► List<? super CharSequence>

# Sous-typage avec *wildcards*

- Sous forme d'arbre, cela donne :





# Wildcard et *Raw Type*

Les règles de conversion entre types paramétrés et *raw types* s'applique aussi pour les *wildcards*

```
public class WildcardAndRawType {  
    public static void main(String[] args) {  
        List<? extends String> list=new ArrayList<String>();  
        List raw=list;      // ok, conversion implicite  
  
        List<?> list2=raw; // unchecked conversion  
    }  
}
```

On peut voir un raw type comme une sorte de wildcard qui générerait un *unsafe warning* à la place d'une erreur lors de l'utilisation

# Type paramétré par un type paramétré

**List<List<?>>** représente une liste dont chaque élément est de type **List<?>**. Pour chaque élément (pris séparément), **?** représente un type inconnu, mais qui peut être différent de celui d'un autre élément.

Ainsi **List<List<String>>** n'est pas un sous-type de **List<List<?>>**. En revanche, c'est un sous-type de **List<? extends List<?>>** puisque **List<String>** est un sous-type de **List<?>**.

# *Wildcard capture*

Mécanisme qui permet pour un appel de méthode de considérer que  $T = ?$

```
public class List<E> {  
    public static <T> List<T> reverse(List<T> list) {...}  
    public static <T> List<T> union(List<T> l1, List<T> l2) {...}  
  
    public static void main(String[] args) {  
        List<Integer> l1=...  
        List<Integer> l2=reverse(l1); // ok  
        List<Integer> l3=union(l1,l2); // ok  
        List<?> l4=l1;  
        List<?> l5=reverse(l4);      // ok, capture  
        List<?> l6=union(l4,l5);    // erreur, pas capture  
    }  
}
```

la capture ne marche que sous certaines conditions

# Conditions de *capture*

La capture des wilcards ne se fait que dans le cas où :

la variable de type n'est présente qu'une fois en paramètre en plus du type de retour

ET le type paramétré n'est pas lui-même un argument d'un type paramétré

```
public static <T> List<T> reverse(List<T> list) {...}  
// capture possible
```

```
public static <T> List<T> union(List<T> l1, List<T> l2) {...}  
// capture pas possible
```

```
public static <T> List<T> subItem(List<List<T>> list) {...}  
// capture pas possible
```

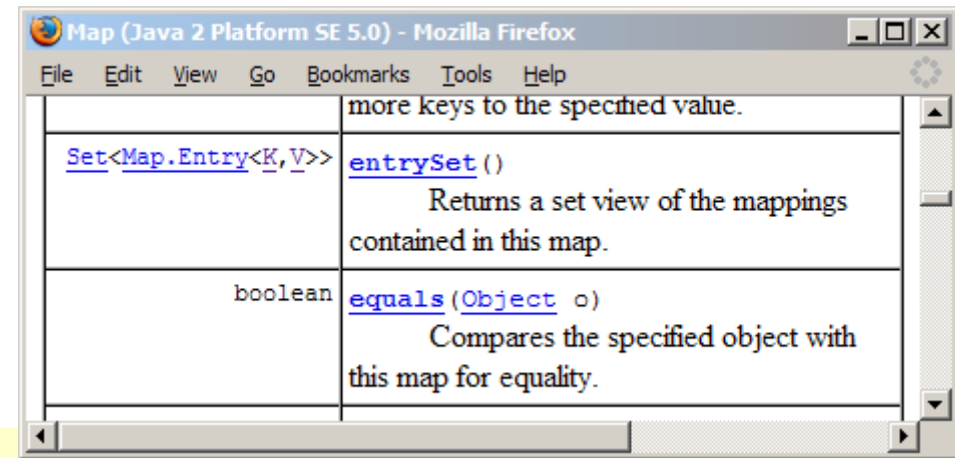
# Capture et Wildcards bornés

Le mécanisme de capture marche aussi pour les *wildcards* bornés (**extends** et **super**)

```
public class List<E> {  
    public static <T> List<T> reverse(List<T> list) {...}  
  
    public static void main(String[] args) {  
        List<?> l1 = new List<String>();           // sous-typage  
        List<?> l2 = reverse(l1);                  // ok, capture  
  
        List<? extends String> l3 = new List<String>(); // sous-typage  
        List<? extends String> l4 = reverse(l3);       // ok, capture  
  
        List<? super String> l5 = new List<String>();  // sous-typage  
        List<? super String> l6 = reverse(l3);        // ok, capture  
    }  
}
```

# Problème avec la *Capture*

Problème lorsque l'on utilise des types paramétré en tant qu'argument de type paramétré



```
public static void main(String[] args) {  
    HashMap<String,Integer> hashMap = new HashMap<>();  
    hashMap.put("toto", 3);  
    hashMap.put("titi", 4);  
    Map<String,?> map = hashMap; // sous-typage  
  
    Set<Map.Entry<String,?>> set = map.entrySet(); // problème de capture  
    //found: java.util.Set<java.util.Map.Entry<java.lang.String,capture of ?>>  
  
    Set<? extends Map.Entry<String,?>> set = map.entrySet(); // ok  
}
```

# Méthode paramétrée et redéfinition

Pour qu'il y ait redéfinition, les deux méthodes doivent

- être toutes les 2 non-paramétrées ou toutes les 2 paramétrées
- avoir la même signature (au nom de variable de type près)

La méthode redéfinie peut avoir un type de retour plus spécifique

S'il n'y a pas redéfinition, soit c'est une surcharge soit il y a un conflit dû à l'erasure

# Redéfinition de méthode paramétrée

Il y a redéfinition si les variables de type des méthodes ont la **même** borne

```
public class A {  
    public <E> void m(E e) {  
    }  
}  
public class B extends A{  
    public @Override <F> void m(F e) { // ok  
    }  
}
```

Cela ne marche pas s'il y a sous-typage entre les bornes !!



# Redéfinition de méthode paramétrée

Les variable de type des méthodes doivent avoir la **même** borne

```
public class A {  
    public <T extends A> void z(T t) {  
        System.out.println("A");  
    }  
}  
  
public class B extends A{  
    public <U extends B> void z(U u) {  
        System.out.println("B");  
    }  
    public static void main(String[] args) {  
        A b=new B();  
        b.z(b); // affiche A ou B ?  
    }  
}
```

# Redéfinition et Raw type

Pour permettre la compatibilité avec du code écrit en 1.4, une méthode avec des raw types redéfinit une méthode avec des types paramétrés (pas l'inverse)

```
public class A {  
    public List<String> m1() { ... }  
    public List m2() { ... }  
}  
  
public class B extends A {  
    public @Override List m1() { ... }           // ok  
    public @Override List<String> m2() { ... }    // illegal  
}
```

# *Erasure*

Effectuée par le compilateur, consiste à transformer un code générique en code classique

- Les variables de type sont changées en leurs bornes
- Ajout de casts sur les valeurs de retour et ajout de méthode *bridge* en cas de sous-typage
- Transforme les types paramétrés en *raw* type

L'erasure peut entrainer des conflits

# Hérité d'un type paramétré

Lorsque l'on hérite ou implante un type paramétré, les méthodes redéfinies doivent respectées une certaine signature

```
public static void main(String[] args) {  
    Holder<String> h = new StringHolder();  
    h.set("toto");  
    String s = h.get();  
}
```

```
public interface Holder<T> {  
    T get();  
    void set(T value);  
}
```

```
public class StringHolder implements Holder<String> {  
    public String get() {...}  
    public void set(String value) {...}  
}
```

Lors de l'écriture StringHolder doit implanter les méthodes (Object get() et set(Object)).

# Les méthodes *bridge*

Les méthodes bridges sont des méthodes générées par le compilateur lors de l'abrasion :

- pour la covariance du type de retour
- pour hériter d'un type paramétré

Ces méthodes permettent lors d'un appel polymorphe d'appeler la méthode redéfinie

# Les méthodes *bridge*

Permet lors de l'abrasion d'hériter d'un type paramétré

```
public class StringHolder implements Holder<String> {  
    public String get() {...}  
    public void set(String value) {...}  
}
```

Erasure



```
public class StringHolder implements Holder {  
    public String get() {...}  
    public void set(String value) {...}  
    public void set(Object value) { // bridge  
        set((String)value);  
    }  
    public Object get() { // bridge  
        return get(); // String get()  
    }  
}
```

Le compilateur à le droit de créer des méthodes ayant les mêmes noms et et paramètres !!

# Tout n'est pas paramétré !!

On utilise des types/méthodes paramétrés :

- Lorsqu'on utilise des types paramétrés déjà définies
- Lorsque l'on veut vérifier une contrainte de type entre des paramètres
- Lorsque l'on veut récupérer un objet du même type que celui stocké

Sinon, on utilise le sous-typage classique

# Tout n'est pas paramétré

On peut utiliser le sous-typage classique

```
<S extends CharSequence> void idiot(S s) {  
    ...  
}  
  
void moinsIdiot(CharSequence s) {  
    ...  
}
```