Types paramétrés

Rémi Forax
forax@univ-mlv.fr

Type paramétré

- Exemple : une liste d'éléments paramétrée par le type des éléments
- Déclaration

```
public class List<T> {
  ...
}
```

T est une variable de type.

- Utilisation (appelé aussi instantiation)

```
...
  public static void main(String[] args) {
    List<String> l=...
  }
```

Type paramétré et Java 1.4

- En Java sans type paramétré, on crée une liste d'**Object**.

```
public class List/*<T>*/ {
  void add(/*T*/Object e) {...}
  Object/*T*/ get(int index) {...}
}
```

- Comme tous les objets héritent de Object :

```
...
  public static void main(String[] args) {
    List/*<String>*/ l=new List/*<String>*/();
    l.add("toto");                // ok grace au sous typage
    String s=l.get(0);            // erreur
    String s2=(String)l.get(0); // ok
  }
```

- C'est au programmeur de faire le *cast*

Le problème fréquent

- Comme on utilise une liste d'Object, le compilateur accepte n'importe quel objet

```
...
  public static void main(String[] args) {
    List/*<String>*/ l=new List/*<String>*/();
    l.add("toto");                  // ok grace au sous typage
    l.add(new Date());              // ok grace au sous typage

    for(Object o:l) {
      System.out.println((String)o); // ClassCastException
    }
  }
```

- Ce programme compile mais plante à l'exécution

Le besoin de type paramétré

- Comme on utilise **Object** comme type,
  il est difficile d'avoir un code :
  - Générique
    (ne pas ré-écrire le code pour la liste de torchon, de serviette, de ...)
  - **ET** statiquement typé
    (vérifiable par le compilateur)
- Solution : permettre réellement les types paramétrés

Template en C++

- Définition & Instantiation :
  - On définit du code à trous (non vérifié)
  - Lors de l'édition de lien, on instantie les templates et effectue la vérification.
- Pour :
  - permet la spécialisation de code
- Contre :
  - recompilation de chaque instantiation
  - explosion du code en taille
  - code peu sûr

Generics en C#

- Définition & Instantiation :
  - Le code du generics est vérifier lors de sa définition
  - le compilateur génére un code “générique” spéciale
- Pour
  - Information de type disponible à l'exécution
- Contre
  - Pas compatible ancienne version
  - Pas de spécialisation de code
  - Code plus lent (pour l'instant mais peut-être ...)

Generics en Java

- Définition & Instantiation :
  - Le code du generics est vérifié lors de sa définition
  - le compilateur transforme le code générique en code classique
- Pour
  - VM classique, compatible ancienne version
- Contre
  - Pas d'information de type lors de l'exécution
  - Pas de spécialisation de code

Plus en détail, en Java

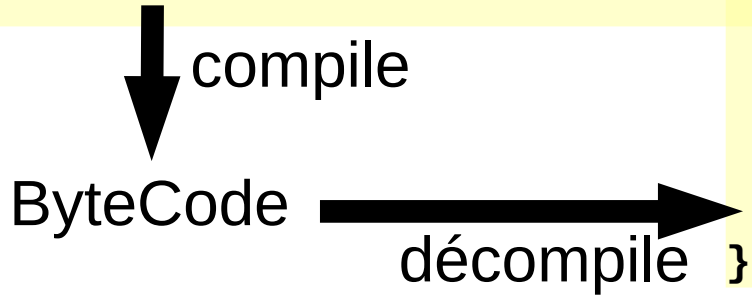
- Il est possible en Java de déclarer :
  - des types paramétrés
  - des méthodes paramétrées
  - des classes internes paramétrées

- Seul le compilateur connait les types paramétrés. A l'exécution, **pour la VM**,
  les types paramétrés n'existent pas

Le principe de l'*erasure*

- Le compilateur traduit les types paramétrés et le code d'appel en code classique

```
public class List<T> {
  void add(T e) {...}
  T get(int index) {...}
  public static void main(String[] args) {
    List<String> l=new List<String>();
    l.add("toto");
    String s=l.get(0);
  }
}
```




```
public class List {
  void add(Object e) {...}
  Object get(int index) {...}
  public static void main(String[] args) {
    List l=new List();
    l.add("toto");
    String s=(String)l.get(0);
  }
}
```

- Cette étape est appelé abrasion (e*rasure)*

Type paramétré

- Un type paramétré (**Pair**) déclare plusieurs variables de type (**T** et **U**)

```
public class Pair<T, U> {
  public Pair(T t,U u) {
    this.t=t;
    this.u=u;
  }
  private final T t;
  private final U u;
}
```

- Lors de l'instantiation, les types arguments (**Integer** et **String**) sont associés aux variables de types

```
public static void main(String[] args) {
  Pair<Integer,Object> pair=new Pair<Integer,Object>(3,"toto");
}
```

Utilisation de types paramétrés

- La documentation indique que la classe **ArrayList** est déclarée **ArrayList<E>**

- **add** est déclaré add(E)

ArrayList() Constructs an empty list with an initial capacity of ten.
ArrayList(Collection<? extends E> c) Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.
ArrayList(int initialCapacity) Constructs an empty list with the specified initial capacity.

Method Summary	
boolean	**add**(E o) Appends the specified element to the end of this list.
void	**add**(int index, E element) Inserts the specified element at the specified position in this list.
boolean	**addAll**(Collection<? extends E> c) Appends all of the elements in the specified Collection to the end of this list, in the order that they are returned by the specified Collection's Iterator.
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified Collection into this list, starting at the specified position.

Utilisation de types paramétrés (2)

- Il faut spécifier un type lors de l'instantiation du generics (ici String)

```
public static void main(String[] args) {
  ArrayList<String> list=new ArrayList<String>();
```

- Pour le compilateur, E=String donc

```
list.add(3);      // illégal, int->String impossible
list.add("toto"); // ok
```

- De même, pour E get(int index)

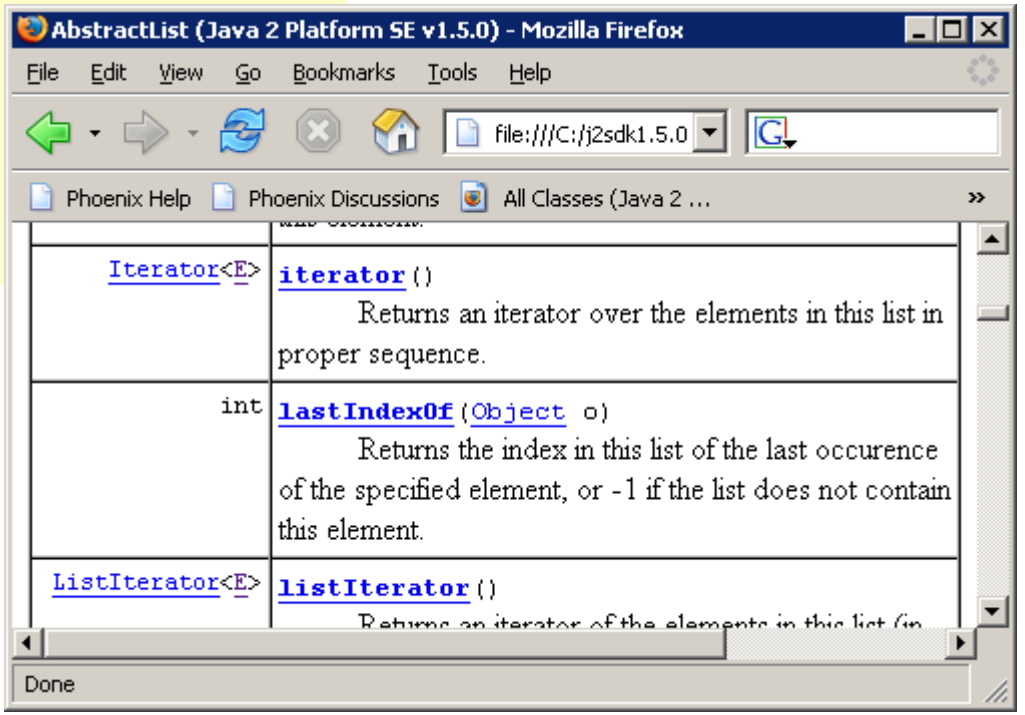
```
Number n=list.get(0); // illégal
String s=list.get(0); // ok
```

Utilisation de types paramétrés (3)

- L'association entre une variable de type et un type se propage

```
public static void main(String[] args) {
  ArrayList<String> list=
   new ArrayList<String>();

  Iterator<Number> it=
    list.iterator(); // illégal
  Iterator<String> it=
    list.iterator(); // ok
```

- Pour le compilateur, E=String



Méthode paramétrée

- Déclaration

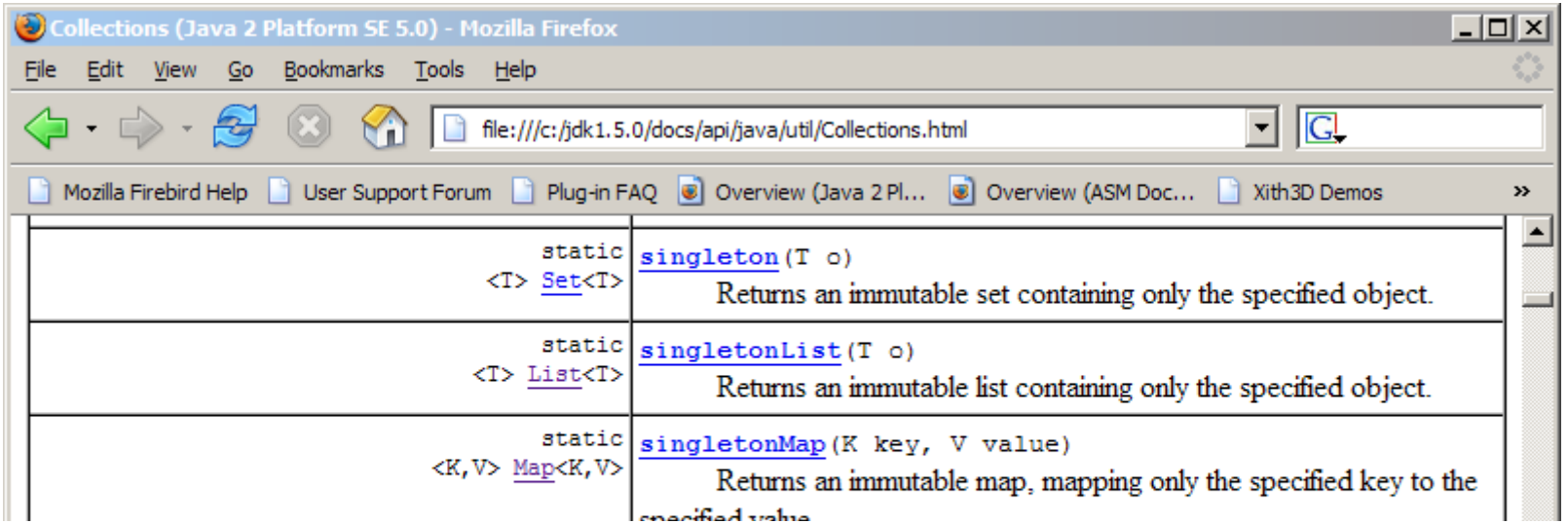
```
public class HelloGenerics {
  public static <T> boolean isPresent(T element, T... array) {
    for(T t:array)
      if (t.equals(element))
        return true;
    return false;
  }
}
```

- La variable de type (**T**) est déclarée entre les modificateurs et le type de retour

Utilisation des méthodes paramétrées

- De même, les types arguments sont associés aux variables de types

```
public static void main(String[] args) {
  List<String> singleton=
    Collections.<String>singletonList(args[0]);
  Map<String,Integer> map=
    Collections.<String,Integer>singletonMap(args[0],2);
```



Utilisation des types paramétrés

- Un type paramétré est utilisé pour typer :
  - Un paramétre ou une valeur de retour
  - Une variable local
  - Un champs

```
public class MyBasket {
  ...
  public void add(String text) {
    list.add(text);
  }
  private final ArrayList<String> list;
}
```

Type paramétré et cast

- Le type argument est **perdu** après la compilation, la VM ne peux donc le verifier lors de **l'exécution**

```
public class GenericCast {
  public static void main(String[] args) {
    ArrayList<String> l=new ArrayList<String>();
    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 class GenericCast2 {
  public static void main(String[] args) {
    ArrayList<String> l=new ArrayList<String>();
    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**

Raw Type

- On appel *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 :
  - Type<T> vers RawType

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

  - RawType vers Type<T>

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

Raw Type et conversion

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

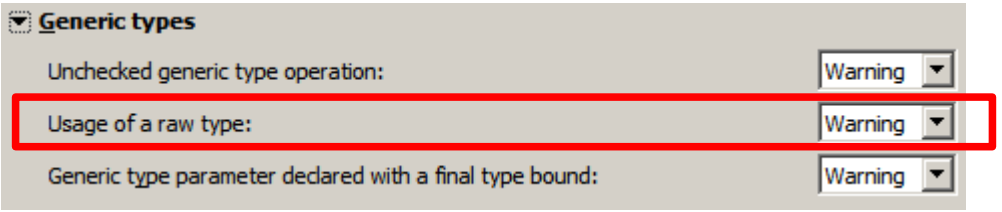
```
public class Holder<T> {
  public Holder(T t) {
    this.t=t;
  }
  public T get() {
    return t;
  }
  private final T t;
```

- 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)
}
```

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 prochaine version de javac (1.7) devrait possèder le même flag.

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 plusieurs 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;
```

Unchecked warnings (2)

- La règle est la suivante :
  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 convertions non verifiées

@SuppressWarnings

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

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

- **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 @SupressWarnings si le code n'est pas sûr car cela cache une CCE qui sera levé plus tard

```
@SuppressWarnings("unchecked")
public static List<Integer> unsafeCast(List<String> o) {
  List<?> list=o;
  return (List<Integer>)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 !!!!
}
```

Déclaration d'un type paramétré

- Un type paramétré déclare plusieurs variables de type ayant chacunes des bornes

```
public class Pair<T extends Number, U extends Object> {
  public Pair(T t,U u) {
    this.t=t;
    this.u=u;
  }
  private final T t;
  private final U u;
}
```

- Lors de l'instantiation, les types paramétrés doivent être des sous-types des bornes

```
    Pair<Integer,Object> pair=new Pair<Integer,Object>(3,"toto");
```

Borne des variables de type

- Si l'on omet la borne d'un type paramétré, Object est utilisé comme borne

```
public class Holder<T> {
  public Holder(T value) {
    this.value=value;
  }
}
```

- Les bornes des variables de types sont obligatoirements des objets, les types primitifs ne sont pas permis

Variable de type & type primitif

- Il n'est pas possible d'utiliser un type primitif en tant qu'argument d'un type paramétré

```
public static void main(String[] args) {
  List<int> list=...
  // int n'est pas un sous-type de Object
}
```

- Il est par contre possible d'utiliser les *wrappers* et l'auto-[un]boxing

```
public static void main(String[] args) {
  List<Integer> list=...
  list.add(3);           // auto-boxing
  int value=list.get(0); // auto-unboxing
}
```

- Le code n'est pas optimisé (pour l'instant :)

Borne des variables de type

- Il est possible de spécifier plusieurs types à une borne
  - Soit une classe et des interfaces (séparé par **&**)

```
public class A<T extends Clonable & Closable> {
}
public class B<T extends Cloneable & String> {
} // erreur, String n'est pas une interface
public class C<T extends String & Cloneable> {
} // ok
```

  - Soit une variable de type (attention à l'ordre de déclaration)

```
public class A<U, T extends U> {
} // ok
public class B<U, T extends U & Comparable<T>> {
} // aïe, mélange de variable de type et type classique
```

Type d'une variable de type

- Une variable de type se comporte comme sa borne dans un code générique

```
public class Holder<T> {
  ...
  public void f() {
    m(t);
  }
  public void m(Object n) { }
  public void m(Integer i) { }
  final T t;
  ...
  public static void main(String[] args) {
    Holder<Integer> holder=new Holder<Integer>(3);
    holder.f(); // appel m(Object) !!
  }
}
```

- Donc pas de spécialisation :(

Type d'une variable à borne multiple

- Il est possible de spécifier une classe et plusieurs interfaces (séparé par un &) en tant que borne

```
interface AnInterface {
  void m();
}
public class Holder<T extends Number & AnInterface> {
  public void hold(T t) {
    t.m();         // T est vue comme un I
    t.intValue(); // T est vue comme un Number
  }
}
```

- La variable se comporte alors comme l'union de des types de sa borne

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

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

Raw Types utilisés comme borne

- Ne pas faire, rend le code unsafe, List doit être paramétré

```
public class RawTypeAsBound {
  public static <T extends List> void addAll(T a, T b) { // unsafe
    a.addAll(b);
  }
  public static void main(String[] args)  {
    List<Integer> a = new ArrayList<Integer>();
    List<String> b = new ArrayList<String>();
    b.add("toto");
    addAll(a, b);
    int value=a.get(0); // ClassCastException
  }
}
```

- Par contre, les *wildcards* sont permis comme borne

Utilisation des variables de type

- Une variable de type est utilisée pour typer :
  - Un paramètre ou une valeur de retour
  - Une variable locale
  - Un champ
    mais pas **static** !!

```
public class List<T> {
  public T get(int index) {
    T value=array[index];
    return value;
  }
  private final T[] array;
}
```

- Une variable de type ne peut pas être utilisée en tant que classe de base ou interface à implanter

```
public class Bug1<T> extends T {}    // erreur
public class Bug2<T> implements T {} // erreur
```

Utilisation des variables de type (2)

- Une variable de type peut être utilisé en tant qu'argument d'un type paramétré

```
public class List<T> {
  public List(int offset,int length) {
    ...
  }
  public List<T> subList(int offset, int length) {
    return new List<T>(offset,length);
  }
  private final int offset, length;
}
```

```
public static void main(String[] args) {
  List<String> ls=new List<String>();
  List<String> sub=ls.subList(0,3);
}
```

Exemple de cast non sûr

- Comme la valeur de T (ici String) n'est pas connu à l'exécution, le cast ne peut être vérifiée

```
public class Beurk<T> {
  public T f(Object o) {
    return (T)o; // unsafe cast
  }
  public static void main(String[] args) {
    Beurk<String> b=new Beurk<String>();
    String s=b.f(3); // compile, mais plante !!
    Object o=b.f(3); // compile, mais devrait planté
  }
}
```

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

Opérations non sûre

- Le type argument est perdu à l'exécution à cause de l'erasure donc certaines opérations sur les variables de type ne sont pas sûres voir interdites :
  - implements/extends T (interdit)
  - new T ou new T[] (interdit)
  - T.class (interdit)
  - instanceof T (interdit)
  - cast : (T) ou (T[]) (warning)

Variable de type et tableau

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

```
public class List<T> {
  public List(int capacity) {
    array=new T[capacity];            // illégal
    array=(T[])new Object[capacity]; // unsafe cast
  }
  public T get(int index) {
    return array[index];
  }
  private final T[] array;
}
```

- 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 (2)

- Pour l'exemple précédent, on peut aussi écrire :

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

Variable de type et exceptions

- Il est possible d'utiliser une variable de type dans un **throws** mais pas dans un **catch**

```
public interface Function<E extends Exception> {
  public Object run(String value) throws E;
}
...
public static void main(String[] args) {
  Function<NumberFormatException> f=
    new Function<NumberFormatException>() {
    public Integer run(String value) throws NumberFormatException {
      return Integer.parseInt(value);
    }
  };
  try {
    System.out.println(f.run(args[0]));
  } catch(NumberFormatException e) { // faire qqchose
  }
}
```

Accès aux membres statiques

- Les membres statiques ne sont pas sujets à la “paramétrisation”

```
public class List<T> {
  public static T f(T t) {..} //illégal, T pas accessible
  public static int f(int i) {...}
  public static void main(String[] args) {
    List<String>.f(3); // illégal
    List.f(3);         // ok
  }
}
```

- L'accès aux membres statique se fait donc en utilisant le *raw type*.

Variable de type et statique

- Une variable de type n'est pas accessible :
  - aux membres statiques (classes, champs)
  - aux enumérations (enum)
  - aux interfaces

```
public class List<T> {
  public static T f(T t) {
    return t;
  } //non-static class T cannot be referenced
    //from a static context
}
```

- Une variable de type n'est visible que si le contexte n'est pas statique

Variable de type et classe interne

- Les variables de type déclarées dans une classe englobantes sont :
  - accessibles aux *inner classes*
  - inaccessibles aux classes internes statiques

```
public class Outer<T> {
  public class Inner {
    T f() {...}
  }
  public static class Internal {
    T f() {...} // non-static class T cannot be
                // referenced from a static context
  }
}
```

Méthode paramétrée

- Lors de l'appel d'une méthode paramétrée, les variables de type peuvent être :
  - soit écrites explicitement
  - soit inférées (devinées) par Java

```
public class HelloGenerics {
  public static <T> boolean isPresent(T element, T... array) {
    ...
  }
  public static void main(String[] args) {
    isPresent("toto","toto","titi"); // inféré
    HelloGenerics.<String>isPresent("toto","toto","titi");
  }
}
```

Appel explicite de méthode

- On peut indiquer au compilateur les types arguments d'une méthode paramétrée en déclarant ceux-ci **avant** le nom de la méthode

```
public class Explicit {
  public static <T> T f(T element) {...}
  public <E> E g(E element) {...}

  public static void main(String[] args) {
    Explicit e=new Explicit();
    e.<String>g(null);         // ok
    Explicit.<String>f(null); // ok
    <String>f(null);           // illégal, problème de grammaire
  }
}
```

- L'appel explicite nécessite une notation pointée (“.”)

Inférence de type

- Le compilateur utilise le type des paramètres et la signature de la méthode pour trouver le type de retour

```
public class Implicit {
  public static <T> T choose(boolean test, T e1, T e2) {
    return (test)?e1:e2;
  }
  public static void f(String s) {...}        // 1
  public static void f(Object o) {...}        // 2
  public static void f(Comparable<?> c) {...} // 3

  public static void main(String[] args) {
    f(choose(true,"toto","titi"));             // appel 1
    f(choose(true,"toto",3));                  // appel 3
    f(Implicit.<Object>choose(true,"toto",3)); // appel 2
  }
}
```

Opérateurs paramétrés

- Les opérateurs ==, !=, ?: sont paramétrés
- Signatures :
  - <T> boolean T**==**T
  - <T> boolean T**!=**T
  - <T> T **?**(condition)T**:**T

```
public static void main(String[] args) {
  String name="toto";
  Date date=new Date();
  Comparable<?> c=(args.length==0)?name:date;
  // Comparable<?> indique un comparable de n'importe quoi

  if (date==name); //erreur, Incompatible operand types Date and String
}
```

Méthode avec variable de type libre

- Il est possible de déclarer des variables de type non liées à un type de paramètre

```
public class Inference {
  public static <T> List<T> emptyList() {...}
    // T est utilisé uniquement en tant que type de retour

  public static void main(String[] args) {
    Inference.<String>emptyList(); // List<String>
    Inference.<Double>emptyList(); // List<Double>
    Inference.emptyList();         // List<?>
  }
}
```

- La notation *wildcard* (List<?>) est utilisée si l'inférence n'est pas possible

Inférence & constructeur

- L'inférence **n'est pas effectuée** sur les constructeurs pour déterminer les types arguments des variables de type d'une classe

```
public class Beurk<U,V> {
  public Beurk(U u,V v) {
  }
  public <T extends Integer> Beurk(T t,U u,V v) {
  }
  public static void main(String[] args) {
    Beurk<String,Double> p=new Beurk("toto",3.0); // unchecked call
    // to Beurk(U,V) as a member of the raw type Beurk
    Beurk<String,Double> p=new Beurk<String,Double>("toto",3.0);

    Beurk<String,Double> p=new Beurk<String,Double>(3,3.0);
    Beurk<String,Double> p=new <Integer>Beurk<String,Double>(3,"t",3.0);
  }
}
```

Quantificateur

- Déclaré 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) { ... }
}
```

Quantificateur

- Il n'est **pas possible** pour une classe d'avoir le quantificateur **pour tout T** tel que
- On peut utiliser des wilcards non bornés et un unsafe cast

```
public class Registry {
  public <T> void register(Class<T> type,T element) {
    map.put(type,element);
  }
  @SupressWarnings(“unchecked”)
  public <T> T get(Class<T> type) {
    return (T)map.get(type)
  }
  private final HashMap<Class<?>,?> map=
    new HashMap<Class<?>,Object>();
}
```

Restrictions d'utilisations

- Il est possible d'utiliser un type paramétré à la place d'un type classique **sauf**
  - Avec instanceof
  - En tant que type d'un tableau pour la création

    List<String>[] s = new List<String>[10]; // erreur
  - En tant qu'exception ou énumération
  - En utilisant la notation **.class**
- On ne peut pas créer d'objet dont le type est donné par une variable
  (pas de new T() ou new T[]);

instanceof et type paramétré

- Il n'est pas possible d'utiliser **instanceof** avec un type paramétré autre qu'en utilisant un *wilcard* non borné

```
public class InstanceofGenerics {
  public static void main(String[] args) {
    List<String> l=new ArrayList<String>();
    if (l instanceof List);                    // ok
    if (l instanceof List<?>);                 // ok
    if (l instanceof List<String>);            // erreur
    if (l instanceof List<? extends String>); // erreur
    if (l instanceof List<? super String>);   // erreur
  }
}
```

- Le type argument n'est pas disponible lors de l'exécution

Type paramétré et tableau

- Les tableaux permettent le sous-typage sans restriction, il n'est donc pas possible de faire des tableaux de type paramétré sûr.

```
public class ArrayRappel {
  public static void main(String[] args) {
    Object[] array=args; // sous-typage
    array[0]=3;          // compile mais ArrayStoreException
  }
}
```

```
public class ArrayGenerics {
  public static void main(String[] args) {
    List<String>[] lists=new List<String>[10]; // illégal
    Object[] array=lists; // sous-typage
    List<Integer> values=new List<Integer>();
    values.add(3);
    array[0]=values; // marche mais ne devrait pas
    String s=lists[0].get(0); // ClassCastException
  }
}
```

Type paramétré et tableau (2)

- Les tableaux de *wildcard* non borné sont permis

```
public class ArrayGenerics {
  public static void main(String[] args) {
    List<?>[] lists=new List<?>[10]; // ok
    Object[] array=lists;            // sous-typage
    List<Integer> values=new List<Integer>();
    values.add(3);
    array[0]=values;           // ok
    String s=lists[0].get(0); // illégal
    Object o=lists[0].get(0); // ok
  }
}
```

- Les tableaux de *raw types* sont possibles aussi mais pas conseillé

Type paramétré et enum

- Un enum ne peut être paramétré car cela voudrait dire paramétrer les valeurs de l'enum qui par définition sont des constantes
(valeurs static et final)

```
public enum EnumGenerics<T> { // erreur
  hello, world;
}
```

Type paramétré et Exception

- Un type paramétré ne peut hériter de **Throwable**

```
public class GenericException<T> extends Throwable { // erreur
  public static void main(String[] args) {
    try {
      throw (args.length==0)?new GenericException<String>():
        new GenericException<Integer>();

    } catch(GenericException<Integer> e) { // erreur
    } catch(GenericException<String> e) {  // erreur
    }
  }
}
```

- Il n'est pas possible d'obtenir le type argument à l'exécution (abrasion)

Héritage et type paramétré

- Une classe peut hériter de/implanter :
  - une classe/interface **paramétrée**

```
public class MyList<T> extends ArrayList<T> {
}
public abstract class MyList2<T> implements List<T> {
}
```

  - une classe/interface **paramétrée instanciée**

```
public class StringList extends ArrayList<String> {
}
public abstract class StringList2 implements List<String> {
}
```

Interfaces paramétrées

- Il n'est pas possible d'implanter la même interface avec différents types arguments

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

public class MyHolder implements Holder<String>, Holder<Object> {
  //Holder cannot be inherited with different arguments:
  //<java.lang.String> and <java.lang.Object>
  public String get() {...}
  public void set(String value) {...}
  public void set(Object value) {...}
}
```

Sous-typage et type paramétré

- Le sous-typage classique marche sur les types paramétrés ayant le même type argument

```
public class ArrayList implements List {
  ...
}

public class ClassicSubTyping {
  public static void main(String[] args) {
    ArrayList<String> list=new ArrayList<String>();
    List<String> myList=list; // sous-typage
  }
}
```

- Mais **ne marche pas** pas si le **type argument** n'est pas le **même**

Sous-typage et type paramétré (2)

- Il n'y a pas de sous-typage entre des types paramétrés ayant des types arguments sous-types

```
public class GenericSubTyping {
  public static void main(String[] args) {
    List<String> l1=...
    List<Object> l2=l1; // supposons que cela marche
    l2.add(3);          // boxing, aie mais plante pas
    String s=l1.get(0); // ClassCastException
  }
}
```

- Pas de sous-typage entre **List<Object>** et **List<String>**. Donc comment on fait ?

Wildcard

- Il existe une **écriture** spécifique qui permet de faire du sous-typage sur les types paramétrés

```
public class GenericSubTyping {
  public static void main(String[] args) {
    List<String> l1=...
    List<?> l2=l1;       // ok, cela compile
    l2.add(3);           // illégal, cela ne compile pas
  }
}
```

- **List<?>** veut dire une liste d'un type que l'on ne connait pas “**?**” mais qui hérite de **Object**

Wildcard en tant que type

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

Instantiation d'un *wildcard*

- Un wildcard correspond à un type abstrait (non-réifiable), il n'est donc pas instantiable.

```
public class BadWildcardInstantiation {
  public static void main(String[] args) {
    ArrayList<? extends String> list=
      new ArrayList<? extends String>(); // illégal
    ArrayList<?> list2=
      new ArrayList<?>();                // illégal aussi

    ArrayList<ArrayList<?>> list3=
     new ArrayList<ArrayList<?>>();     // ok
  }
}
```

- Par contre, ArrayList<ArrayList<?>> est reifiable

Appel explicite de méthode

- Il n'est pas possible d'utiliser le "**?**" en tant que type lors d'un appel explicite de méthode

```
public class Collections {
  public static <T> List<T> emptyList() {...}
  ...
}
public class NoWildcardExplicit {
  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 "**?**" (2)

- 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
  }
}
```

Wildcard et borne

- En fait, il existe deux sortes de *wildcard* :
  - List<? **extends** Type> : une liste d'un type que l'on sait être un sous-type de Type.
  - List<? **super** Type> : une liste d'un type que l'on sait être un super-type de Type.
- **List<?>** est une notation raccourçie pour **List<? extends Object>**
- Il n'est pas possible de mettre **plusieurs** borne ou une borne **extends** et **super**

P<? extends Type>

- "**?**" est un type que l'on ne connait pas qui est un **sous-type** de Type donc

```
public class List<E> {
  public void clear() {...}
  public void add(E e) {...}
  public E get(int index) {..}
  public static void main(String[] args) {
    List<Integer> l1=...
    l1.add(3);                        // ok
    List<? extends Number> l2=l1; // compile
    l2.add("toto");                   // illégal
    l2.add(3);                        // illégal, si ?=Double
    Number n=l2.get(0);               // ok
    l2.clear();                       // ok
  }
}
```

- List<? extends Type> est une liste où l'on peut uniquement **sortir** des valeurs

P<? extends Type> et **null**

- Comme il est possible d'assigner **null** à une variable de n'importe quel type
- **null** est la seule valeur qui peut être pris en tant que paramètre d'un "**? extends Type**".

```
public class List<E> {
  public void clear() {...}
  public void add(E e) {...}
  public E get(int index) {..}

  public static void main(String[] args) {
    List<Integer> l1=...
    List<? extends Number> l2=l1; // compile
    l2.add(null);                 // ok
  }
}
```

P<? super Type>

- ? est un type qui est un **super-type** de Type que l'on ne connait pas donc

```
public class List<E> {
  public void clear() {...}
  public void add(E e) {...}
  public E get(int index) {..}
  public static void main(String[] args) {
    List<Object> l1=...
    List<? super Number> l2=l1;   // compile
    l2.add("toto");               // illégal
    l2.add(3);                    // ok
    Number n=l2.get(0);           // illégal
    Object o=l2.get(0);           // ok, tous objets héritent de Object
    l2.clear();                   // ok
  }
}
```

- List<? super Type> est une liste ou l'on peut **mettre** des valeurs

P<? super Type> et **Object**

- Comme il est possible de stocker tout objet dans une variable de type Object
- **Object** est le seul type dans lequel on peut stocker le résultat d'une méthode d'un "**? super Type**".

```
public class List<E> {
  public void clear() {...}
  public void add(E e) {...}
  public E get(int index) {..}

  public static void main(String[] args) {
    List<Number> l1=...
    List<? super Integer> l2=l1; // compile
    Object o = l2.get(0);        // ok
  }
}
```

Sous-typage entre *wildcard*

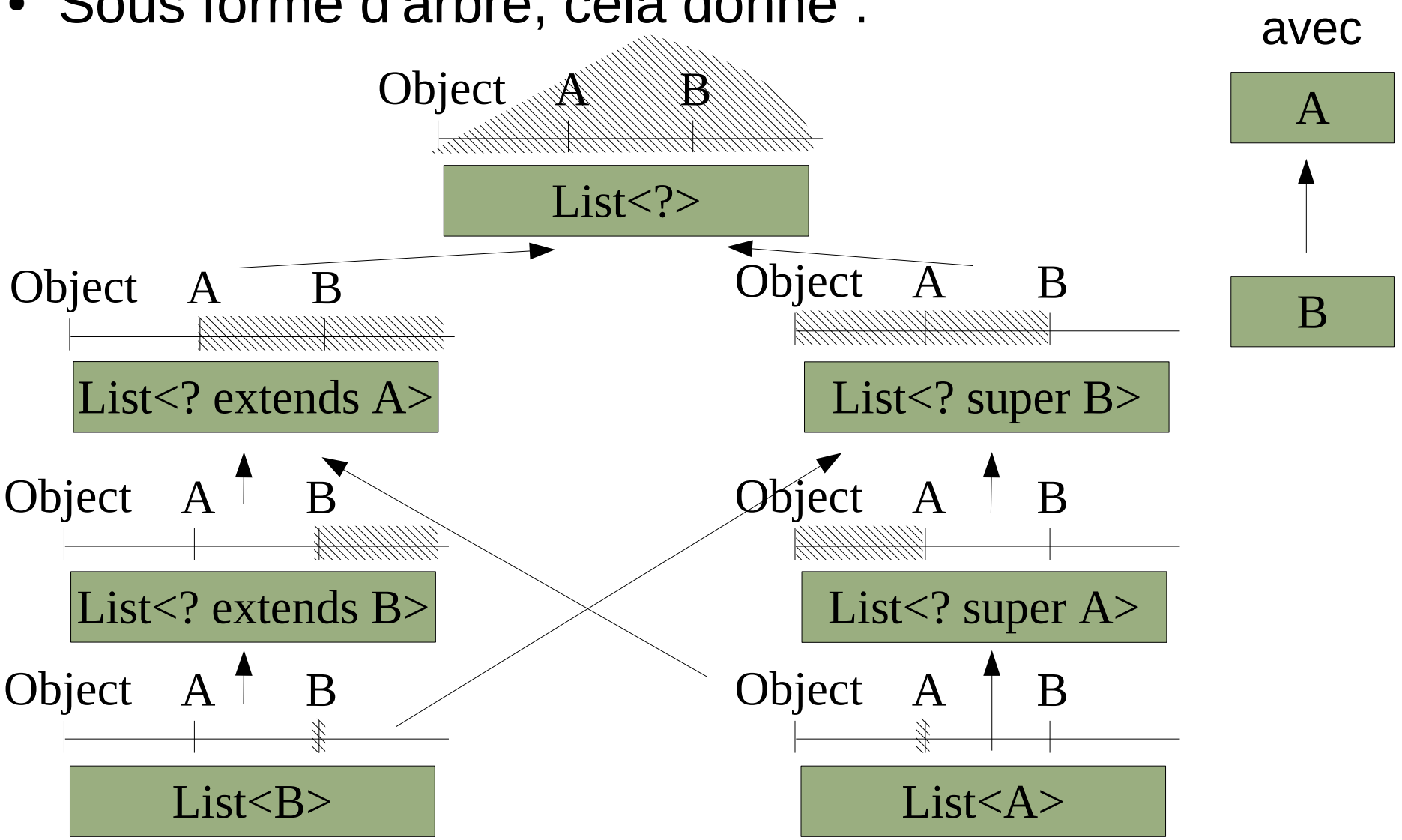
- Les relations de sous-typage entre *wildcards*

```
public class WildcardSubtyping {
  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 rêgles de conversion entre types paramétrés et *raw types* s'applique aussi pour les *wilcards*

```
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 wilcard 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 paramétre en plus du type de retour
  - ET le type paramétré n'est pas lui-même un argument d'un type paramétré

```
public class Capture {
  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
}
```

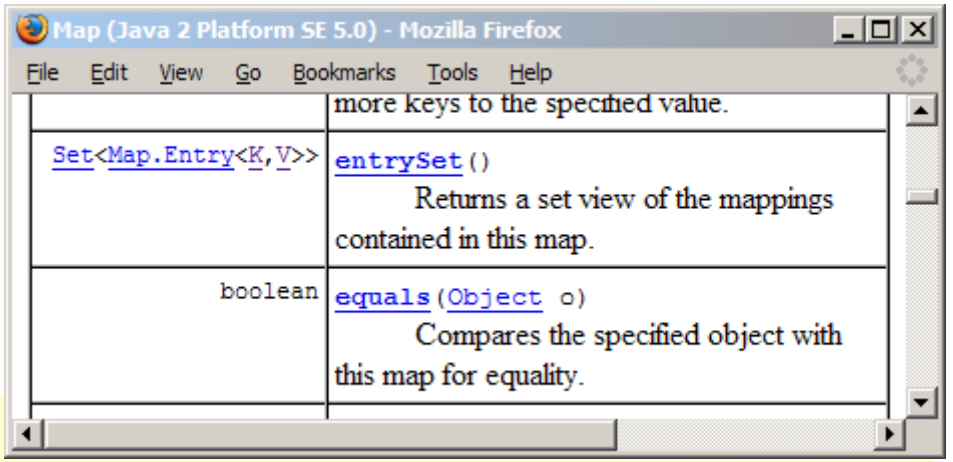
Capture et *Wildcards* bornés

- 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(l4);                         // 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 class CaptureProblem {
  public static void main(String[] args) {
    HashMap<String,Integer> hashMap=new HashMap<String,Integer>();
    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'abrasion

Redéfinition : classe paramétrée

- Redéfinition entre méthodes de classes paramétrées

```
public class C<T extends A> {
  public T z() {
    return null;
  }
}

public class D<U extends B> extends C<U> {
  public @Override U z() { // ok
    return null;
  }
}
```

- U doit avoir une borne qui est un sous-type de la borne de T

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 ?
  }
}
```

Méthode paramétrée et non

- Il n'y a pas de redéfinition entre méthodes paramétrées et non paramétrées

```
public class A {
  public Object noParameter() {
    return null;
  }
  public <U> U withParameter() {
    return null;
  }
}
```

```
public class B extends A{
  public <U> U noParameter() {
    return null;
    // name clash: noParameter() and <U>noParameter() in A
    // have the same erasure, yet neither overrides the other
  }
  public Object withParameter() {
    return null;
    // name clash: withParameter() and <U>withParameter() in A
    // have the same erasure, yet neither overrides the other
  }
}
```

Redéfinition et Raw type

- Pour permettre la compatibilité avec du code écrit en 1.4, il y a redéfinition entre les types paramétrés et les raw types

```
public class A {
  public List m1() { ...}
  public List m2() { ... }
  public List<String> m3() { ... }
}

public class B extends A{
  public @Override List<String> m1() { ... } // plante
  public @Override List<?> m2() { ... } // plante
  public @Override List m3() { ... }
}
```

Entre raw type et wilcard

- Il y a redéfinition entre les *raw types* et *wildcards*

```
public class A {
  public List<?> m1() { ...}
  public List m2() { ... }
  public List m3() { ... }
  public void m4(List<? extends String> list) {
  }
}

public class B extends A{
  public @Override List<? extends String> m1() { ... }
  public @Override List<?> m2() { ... }
  public @Override List<? super Integer> m3() { ... }
  public void m4(List<?> list) {
  } // name clash, mais pourrait marcher
}
```

Abrasion (*erasure*)

- Effectuée par le compilateur, consiste à tranformer 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'abrasion 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 interface Holder<T> {
  T get();
  void set(T value);
}
```

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

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

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

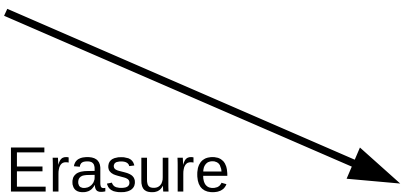
Les méthodes ponts (*bridge*)

- 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 ponts (*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) {...}
}
```




```
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 paramétres !!

Les méthodes pont (2)

- Une méthode bridge redirige l'appel vers une méthode ayant la bonne signature

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

Erasure

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

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

Abrasion et méthode paramétrée

- Comme les variables de types sont remplacées par leurs bornes, il peut y avoir conflit entre deux méthodes

```
public class ErasureConflict<E> {
  public void m(E e) {
    // name clash: m(E) and m(java.lang.Object)
    // have the same erasure
  }
  public void m(Object o) {
  }
}
```

- Ici, la borne de **E** est Object

Méthodes paramétrées et abrasion

- De même que pour les types paramétrés, il peux y avoir conflit entre deux méthodes paramétrées

```
public class ErasureConflict<E> {
  public void m(E e) {
  }
  public <T> void m(T o) {
    // name clash: m(E) and <T>m(T) have the same erasure
  }
  public <U> void m(U o) {
    // <T>m(T) is already defined in ErasureConflict
  }
}
```

- Ici, la borne de E, T et U est Object

Class<T>

- La classe Class<T> représente la classe d'un objet de type T lors de l'exécution.
- La syntaxe **.class** permet d'obtenir la classe correspondant à un type non paramétré

```
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 abrasion

- Permet de passer un type en paramètre non disponible à l'éxécution à cause de l'abrasion.

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

- Mais, on peut utiliser la classe Class<T>

```
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: Integer() n'existe pas
```

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() (2)

- 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>)aiClass.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

Class<T> et instanceof

```
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;
}
```

- **isInstance()** test dynamiquement la classe
- **cast()** effectue le cast ou renvoie une ClassCastException

```
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;
}
```

```
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;
}
```

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 paramétres
  - Lorsque l'on veux récupérer un objet du même type que celui stocké
- Sinon, on utilise le sous-typage classique

Tout n'est pas paramétré

- on peux utiliser le sous-typage classique

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

void moinsIdiot(CharSequence s) {
  ...
}
```