Types paramétrés et méthodes paramétrées

Rémi Forax

Rappel sur les casts

Il y a deux sortes de cast en Java

- Les casts pour le compilateur
  - double d = 2.0;
    int i = (int) d; // opération d2i
  - var p = (IntPredicate) i -> i % 2 == 0;

- Les casts pour la VM (à l'exécution)
  - Object o = 3;
    String s = (String) o; // danger! CCE à l'exécution

Pourquoi ?

Java introduit les types paramétrés (generics) dans la version 5, avant on écrivait

```
ArrayList list = new ArrayList();
list.add("hello");
...
String s = (String) list.get(0);
```

Le cast (String) peut planter à l'exécution

=> code pas sûre !

Pourquoi ? (2)

Le code suivant compile aussi:

```
ArrayList list = new ArrayList();
list.add("hello");
list.add(new URI(...));       // aaaaaaaaaah
...
String s = (String) list.get(0);
```

On détecte le problème à l'endroit du cast, pas à l'endroit où l'on a fait l'erreur :(

Idée des types paramétrés

Les **instances** des classes sont paramétrés, le compilateur

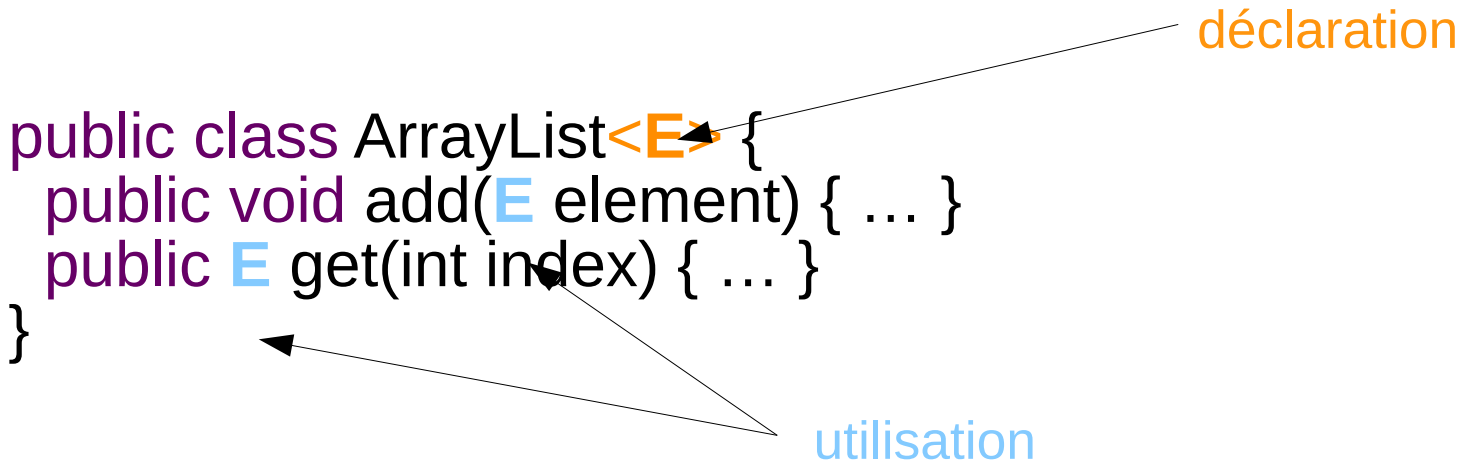
- Vérifie que ce que l'on insére est du bon type
- Renvoie les élements avec le bon type

```
ArrayList<String> list = new ArrayList<String>();
list.add("hello");
list.add(new URI(...));       // compile pas, :)

String s = list.get(0);       // pas besoin de cast :)
```

Déclaration d'un type paramétré

On introduit la possibilité de déclarer
une variable de type



A la compilation, si on utilise une instance de type ArrayList<String>, le compilateur substitue **E** par **String**

donc on obtient les méthodes add(**String**) et **String** get(int)

Déclaration de types paramétrés

Un type paramétré peut avoir besoin de plusieurs variables de types, elles sont alors séparés par des virgules

```
public class HashMap<K, V> {
  public void put(K key, V value) { … }
  public V get(Object key) { … }
}
```

Methode paramétrée

En fait, on a le même problème avec les méthodes

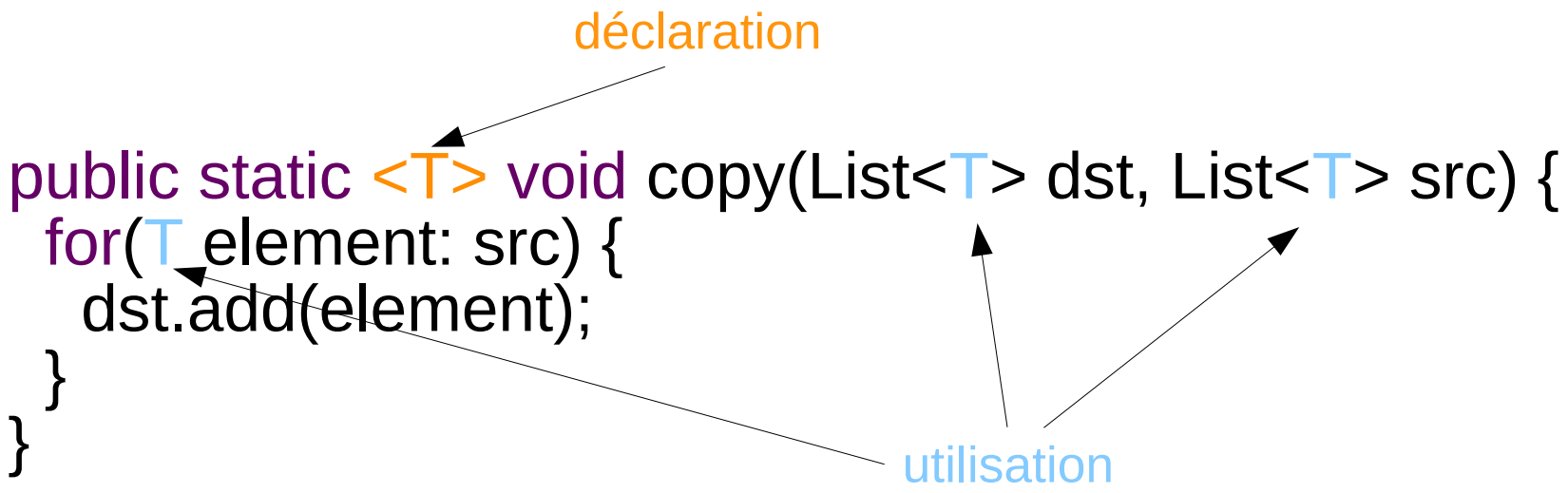
```
public static void copy(List dst, List src) {
  ...
}
```

On veut pouvoir dire que les types des élements dans dst est le même que celui dans src*

* on verra que l'on peut faire mieux

Déclaration d'une methode paramétrée

On déclare aussi **une variable de type**
(après les modificateurs de visibilité et avant le type de retour)



On peut utiliser une variable de type aux endroits où, habituellement, on met un type

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

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

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

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

Variable de type et static

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

```
class Foo<E> {
  E e;                                  // oui
  List<E> list;                         // oui
  E foo(E e) { return e; }              // oui
  void foo2() { E e;  }                 // oui
  class B implements List<E> {}  // oui

  static E e;                                  // non
  static E bar(E e) { return e; }              // non
  static void bar2() { E e; }                  // non
  static class B implements List<E> {}  // non
}
```

Borne d'une variable de type

Une variable de type se comporte comme sa borne

Si on ne définie pas de borne, Object est utilisé

```
public static <R extends Runnable> R foo(R r) {
  r.run();   // on peut appeler run() !
  ...
}
```

Attention, ici **extends** veut dire **sous-type** de,
cela n'a rien avoir avec l'héritage !

Une variable de type n'a pas la visibilité de sa borne !

Plusieurs bornes

Une variable de type peut avoir plusieurs bornes (séparées par &)

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

Une des bornes peut être une classe, elle doit être déclarée en premier

```
<R extends Reader & Closeable>
```

Borne récursive

Une borne peut être paramétrée par une variable de type

```
interface Orderable<T extends Orderable<T>> {
  public boolean lessThan(T element);
}
```

Par exemple, cela permet d'être paramétré par soit même

```
class MyInteger implements Orderable<MyInteger> {
  private final int value;
  ...
  public boolean lessThan(MyInteger element) {
    return value < element.value;
  }
}
```

Type primitif

Les variables de type ont des objets pour borne, il n'est donc pas possible de paramétrer un type ou une méthode par des types primitifs

```
ArrayList<int> list = …  // compile pas
```

On utilise les wrappers Integer, Character, Long, etc et l'auto-boxing/unboxing

```
ArrayList<Integer> list = …
list.add(1);                    // convertion int → Integer
int value = list.get(0);    // convertion Integer → int
```

Inférence

Type argument

Lorsque l'on crée une instance d'un type paramétré, on peut indiquer le type argument

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

Où lorsque l'on appele une méthode paramétrée

```
Stack<String> stack1 = …
Stack<String> stack2 = ...
Utils.<String>transfer(stack1, stack2);
```

Attention, on doit spécifié la classe ou une instance devant le '.' sinon le < de <String> est considéré comme un inférieur par le parseur :(

Inférence new + methode

Syntaxe Diamand

```
HashMap<String, URI> map = new HashMap<>();
```

- Marche aussi avec les classes anonymes (Java 9)

Appel de méthode paramétré:

- En fonction des arguments
  - Avec `static <T> List<T> asList(T… elements) { … }`

    ```
    List<Integer> list = asList(1, 4);
    ```
- En fonction du type de gauche
  - Avec `static <T> List<T> emptyList() { … }`

    ```
    List<Integer> empty = emptyList();
    ```

Inférence des types arguments

En fonction du contexte

- En fonction du type de retour de
  - Avec `static <T> List<T> emptyList() { … }`

    `List<Integer> empty() { return emptyList(); }`

- En fonction de la méthode appelée
  - Avec `static void foobar(List<String> list) { … }`

    `foobar(Collections.emptyList());  // List<String>`

lorsqu'il y a de la surcharge, c'est le bazard !

(https://docs.oracle.com/javase/specs/jls/se8/html/jls-18.html)

Algo d’inférence

On peut demander au compilateur d’utiliser le contexte pour trouver (inférer) les types arguments

- Si l’inférence ne marche pas, le compilateur utilise Object :(
- Si plusieurs types sont possibles, le plus spécifiques (sous-type) est utilisé

L’inférence des “var”, des lambdas et des types arguments est un seul et même algorithme

Inférence / type plus spécifique

L'inférence calcule le type le plus spécifique (pas forcément celui que l'on veut)

```
void m(List<Object> list) { … }

var list = List.of("foo", 42);
m(list);   // compile pas, le type est List<? extends Comparable<? extends Comparable<? …)
```

Indiquer le type explicitement marche

```
var list = List.<Object>of("foo", 42);
```

Erasure

Template vs Generics

C++ possède un mécanisme qui prédate les types paramétrés de Java

- Les templates
  - Le compilateur génére autant de classes sur le disque que d'instantiations de types paramétrés
    - HashMap<String, String>, HashMap<String, URI>, etc
- => génère trop de code

En Java, on parle de generics, les types paramétrés sont présents à la compilation **mais absent à l'exécution**

Erasure (abrasion)

Le compilateur vérifie les types paramétrés mais génére un code qui remplace

- remplace les types paramétrés par une version sans les ‘<’ ‘>’ (raw type)
- remplace les variables de type par leur première borne
- introduit les casts là où il faut

Un seul fichier .class pour plusieurs instantiations

Pas possible d’avoir le type argument d’une variable de type à **l’exécution**

Erasure

Pour le type paramétré

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

```
class Stack {
  void push(Object t) { … }
  void Object pop() { … }
}
```

Pour les utilisations de types paramétrés

```
Stack<String> stack = new Stack<String>();
stack.push("hello");
String s = stack.pop();
```

on obtient l’erasure

```
Stack stack = new Stack();
stack.push("hello");
String s = (String) stack.pop();
```

Limitations due à l'érasure

Comme les types arguments des variables de type ne sont pas présent à l'exécution

Les opérations suivantes **ne compilent pas**

- instanceof T ou instanceof Foo<String>
- switch(o) { case T t -> ...
- new T
- new T[] ou new Foo<String>[]
- class Foo<T> extends Throwable { … }
  car catch(Foo<String>) impossible

Un cast (T) ou (Foo<String>) produit un warning (cf plus tard)

Limitations due à l'érasure

On peut avoir des conflits de compilation (name clash) car on peut avoir deux méthodes avec la même signature après erasure

```
class Foo {
  void bar(List<String> list) { … }
  void bar(List<Integer> list) { … }
}
```

De même, une classe ne peut pas implanter deux fois la même interface avec des types arguments différents

```
class Foo implements Bar<String>, Bar<Integer> {
}
```

Erasure et Redefinition

Le compilateur fait de la magie si on implante une interface en figeant les types

```
interface Function<T, R> {
  public R apply(T t);
}

class StringFunction
                implements Function<String, String> {
  public String apply(String t) { … }
}


Function<String,String> fun = new StringFunction();
String s = fun.apply("foo");
```

Erasure et Redéfinition (2)

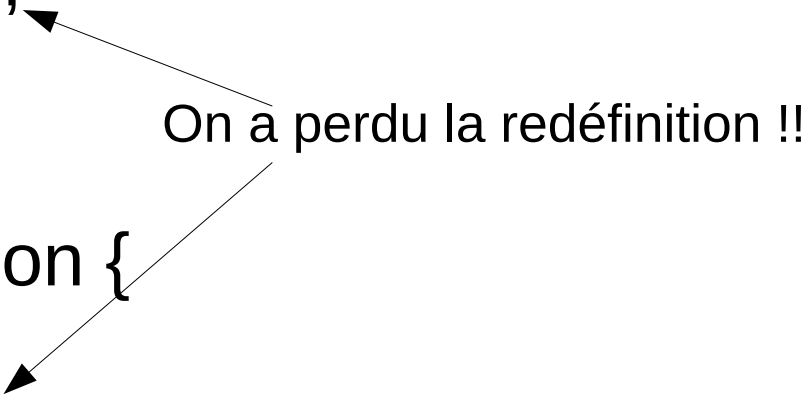
L’erasure est la suivante

```
interface Function {
  public Object apply(Object t);
}

class StringFunction
                implements Function {

  public String apply(String t) { … }
}


Function fun = new StringFunction();
String s = (String) fun.apply("foo");
```



Method bridge

Le compilateur ajoute une méthode bridge !
qui apparait dans les stacktraces :(

```
interface Function {
  public Object apply(Object t);
}

class StringFunction implements Function {
  public /* bridge */ Object apply(Object t) {
    return apply((String)t);
  }
  public String apply(String t) { … }
}


Function fun = new StringFunction();
String s = (String) fun.apply("foo");
```

Les tableaux
de types paramétrés

Les tableaux sont covariants

Les tableaux en Java ne sont pas sûr !

```
String[] s = new String[] { "hello" };
Object[] o = s;   // n'importe quoi !
```

Java permet le sous-typage sur les tableaux de sous-types !

```
o[0] = new URI();   // ArrayStoreException !
```

Le compilateur permet un code non sûre car la VM fait un instanceof à l'exécution !

Les tableaux sont covariants

Les tableaux de variable de types et de types paramétrés sont interdits

- À cause de l'érasure, les types arguments sont perdus à l'exécution
- La VM ne peut pas levée l'ArrayStoreException !

Problèmes, écrire le code de ArrayList/HashMap demande de créer des tableaux de variable de type !

Cast non safe

Pour obtenir des tableaux de variable de type ou des tableaux de types paramétrés, on utilise un cast non safe

```
public class ArrayList<E> {
  private E[] elements;

  public ArrayList() {
    elements = (E[]) new Object[16];
  }
  …
}
```

Le compilateur émet un warning

Marche à l'exécution car le cast n'apparait pas dans le bytecode car E[] a pour erasure Object[]

@SuppressWarnings

On peut dire au compilateur que le cast ne va jamais planter (car le code ne stocke que des E dans le tableau) et supprimer le warning.

```
public class ArrayList<E> {
  private E[] elements;

  @SuppressWarnings("unchecked")
  public ArrayList() {
    elements = (E[]) new Object[16];
  }
  …
}
```

On utilise Object car c'est l'érasure de E

@SuppressWarnings

On peut mettre @SuppressWarnings sur la déclaration d'une variable locale pour réduire sa portée

```
public class ArrayList<E> {
  private E[] elements;

  public ArrayList() {
    @SuppressWarnings("unchecked")
    E[] elements = (E[]) new Object[16];
    this.elements = elements;
  }
  ...
}
```

Et pour les tableaux de types paramètrés

Il existe une notation ? qui correspond au supertype de toutes les instanciations possibles

```
public class HashMap<K, V> {
  static class Entry<K, V> {
    ...
  }
  private Entry<K,V>[] entries;
  @SuppressWarnings("unchecked")
  public HashMap() {
        entries = (Entry<K,V>[]) new Entry<?,?>[16]
  }
  ...
}
```

On utilise Entry<?,?> car c'est l'erasure de Entry<K,V>

@SuppressWarnings est dangereux !

Si utilisé alors que le code est pas sûr

```
@SuppressWarnings("unchecked")
public static <T> T unsafeCast(Object o) {
  return (T) o;
}
```

On va avoir des ClassCastExceptions plus tard !

Dès fois pas loin:

```
String s  = unsafeCast(3);    // CCE !
```

Dès fois, dans un autre module

```
List<String> l = unsafeCast(List.of(3));
...
String s = l.get(0);   // CCE !
```

Varargs

Les “…” en Java demande au compilateur de créer un tableau des arguments avant d’appeler la méthode

- On peut créer un tableau de type paramétré ce qui générera un warning

Par exemple,

```
static <T> List<T> asList(T… elements) { … }

asList("a", "b");                          // String[]  ok !
asList(Set.of("a"), Set.of("b"));  // Set<String>[] unsafe !
```

Varargs

En fait, le problème de la covariance des tableaux n'apparait que si l'on stocke des valeurs qui n'ont pas le bon type

- Si l'on garantie que cela n'arrive pas (parce que l'on stocke que des E) alors le code est sûre

L'annotation @SafeVarargs permet d'indiquer au compilateur de ne pas lever de warning

@SafeVarargs

Comme cela dépend de l'implantation, on ne peut mettre l'annotation que sur des méthodes qui ne peuvent pas être redéfinie

méthode static ou private ou final ou constructeur

Comme asList() est static:

```
@SafeVarargs
static <T> List<T> asList(T… elements) { … }
```

```
asList("a", "b");                          // String[]  ok !
asList(Set.of("a"), Set.of("b"));  // Set<String>[] ok !
```

Legacy code

Legacy Code et Générification

Avec l’erasure, il est possible de générifier un code i.e. introduire des types paramétrés dans un code existant

- C’est ce qui a été fait pour java.util

Dans les anciens codes qui utilisent la version non paramétré d’un type paramétré (raw type)

- ArrayList list = …

le compilateur émet un warning

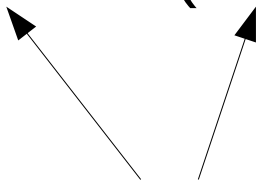
Code Legacy et Redéfinition

Dans le but de pouvoir générifier un module sans générifier les modules dont il dépend

Le code suivant compile

```
interface I {
  List<String> foo(List<URI> list);
}

class C implements I {
  List foo(List list) { … }
}
```



raw types

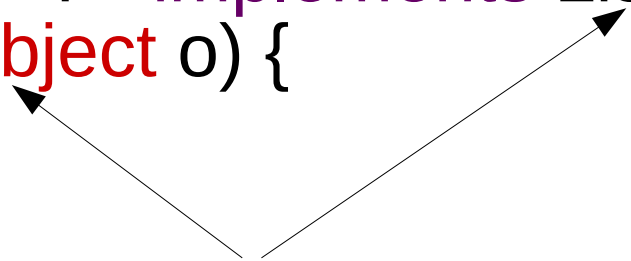
Raw Type et code moderne

Il n’existe plus beaucoup de bibliothèques non générifiée, donc c’est rare de voir un raw type

- c’est souvent que l’on a oublier les ‘<’ ‘>’

Attention, utiliser un raw type change la signature des méthodes !!

```
public class MyList<T> implements List {
  public void add(Object o) {
    ...
  }
  ...
}
```



Covariance
des types paramétrés

Les types paramétrés ne sont pas covariant !!

Contrairement au tableau, les types paramètrés ne sont pas covariant

```
List<String> list = Arrays.asList("foo");
List<Object> list2 = list;   // compile pas !!
```

Heureusement, sinon on pourrait écrire

```
list2.set(0, 3);
String s = list.get(0);    // aaaaaaaaaaaaah
```

Mais si on a pas de covariance ...

La covariance sert à éviter la duplication de code
Par exemple, avec

```
void printAll(List<Object> list) {
  list.forEach(System.out::println);
}
```

on voudrait pouvoir écrire

```
List<String> list = List.of("foo");
printAll(list);   // compile pas !!
```

Quel est le problème ?

```
interface List<E> {
  public void add(E e);
  public E get(int index);
}
```

Utiliser les méthodes qui renvoie un E n'est pas un problème

- Mais utiliser les méthodes qui prennent un E en argument est pas sûr

On veut un moyen d'utiliser List mais sans les méthodes qui prennent un E en argument

On alors une sorte de covariance

Utilisons la notation Object+ pour repésenter un type que que l'on ne connait pas qui est sous-type de Object

Alors

```
void printAll(List<Object+> list) {
  list.forEach(System.out::println);
}
```

et

```
void printAll(List<Object+> list) {
  Object o = 3;
  list.set(0, o);     // ne compile pas
}
```

l'appel à set ne compile pas car le compilateur ne sait pas de quel sous-type de Object il s'agit !

? extends

En Java, une liste d'un type que l'**on ne veut pas connaitre et est sous-type de** la borne,
s'écrit List<**? extends** Borne>

Donc

```
void printAll(List<? extends Object> list) {
  list.forEach(System.out::println);
}
```

et

```
void printAll(List<? extends Object> list) {
  Object o = 3;
  list.set(0, o);     // ne compile pas
}
```

Utilisation

Si on déclare une méthode

- void foo(List<**? extends** CharSequence> list) { … }

Alors

on peut l'appeler avec n'importe qu'elle liste d'un truc qui est **sous-type** de CharSequence

Et à l'intérieur de la méthode foo

- On peut utiliser toutes les méthodes qui ne prennent pas de E* en paramètre

* techniquement, on peut écrire list.add(null), car le type de null est sous-type de tous les types existants

Et la contravariance ?

En fait, on veut aussi faire de la contravariance

Avec

```
void addIfNonNull(String s, List<String> list) {
  if (s != null) { list.add(s); }
}
```

on voudrait pouvoir écrire

```
List<Object> list = new ArrayList<Object>();
addIfNonNull("hello", list);   // compile pas !!
```

? super

En Java, une liste d'un type que l'**on ne veut pas connaitre et est super-type de** la borne,
s'écrit List<**? super** Borne>

Donc

```
void addIfNonNull(String s, List<? super String> list) {
  if (s != null) { list.add(s); }
}
```

et

```
void addIfNonNull(String s, List<? super String> list) {
  String v = list.get(0);       // compile pas
  Object o = list.get(0);
}
```

Utilisation

Si on déclare une méthode

- void foo(List<**? super** CharSequence> list) { … }

Alors

on peut l'appeler avec n'importe qu'elle liste d'un truc qui est super-type de CharSequence

Et à l'intérieur de la méthode foo

- On peut utiliser toutes les méthodes qui ne renvoie pas de E* en paramètre

* techniquement, on peut toujours mettre le résultat dans Object

Rêgles PECS

Si on a une méthode publique qui prend en paramètre des types paramétrés

- Si le type agit comme un **P**roducteur de donnée, il doit être déclaré avec ? **e**xtends
- Si le type agit comme un **C**onsommateur, il doit être déclaré avec ? **s**uper

Si le type agit comme les deux ou si le type est un type de retour, il ne doit pas y avoir de wildcard.

Exemples

Collection.addAll

```
interface Collection<E> {
  public boolean addAll(Collection<? extends E> cs);
}
```

Stream.filter

```
interface Stream<E> {
  public Stream<E> filter(Predicate<? super E> predicate);
}
```

Collections.copy

```
public static <E>
void copy(List<? super E> dst, List<? extends E> src)
```

Wildcard non borné

On appel wildcard non borné (*unbounded wildcard*) le ‘?’ tout seul

- Il est équivalent à **? extends** Object

Comme tous les objets héritent de Object en Java, Foo<?> est sous type de tous types possible de Foo quelque soit le type argument

```
List<?> l = new ArrayList<String>();
List<?> l = new ArrayList<URI>();
```

Wildcard non borné et reifiabilité

Contrairement aux autres wildcards (? extends et ? super), un type paramétré par une wildcard non borné est réifiable

List<?> est une représentation du type de la classe List à l'exécution (à cause de l'érasure)

- instanceof List<?> // ok
- new List<?>[7] // ok
- (List<?>) // ok
- (List<?>[]) // ok

Exemple

```
public class Pair<U, V> {
  private final U first;
  private final V second;
  …
  public boolean equals(Object o) {
    return o instanceof Pair<U,V> pair &&       // compile pas
        pair.first.equals(first) &&
        pair.second.equals(second);
  }

  public int hashCode() {
    return Objects.hash(first, second);
  }
  ...
}
```

Utiliser un wildcard non borné

```
public class Pair<T, U> {
  private final T first;
  private final U second;
  …
  public boolean equals(Object o) {
    return o instanceof Pair<?,?> pair &&  // Ok !
        pair.first.equals(first) &&
        pair.second.equals(second);
  }

  public int hashCode() {
    return Objects.hash(first, second);
  }
  ...
}
```

Wildcard et capture

Wildcard vs Variable de type

Il y a une correspondance entre les variables de types et les wildcards

Les deux méthodes addAll sont équivalentes

```
interface Foo<E> {
  void addAll(Foo<? extends E> c);
  <F extends E> void addAll(Foo<F> c);
}
```

La convention en Java est d'utiliser les wildcards lorsque l'on veut représenter des informations sur la variance

Capture

Il est possible d'associer un '?' à une variable de type, on appel cette opération une capture

Par exemple,

```
<E> Iterator<E> foo(List<E> l) { return l.iterator(); }

List<? extends CharSequence> list = …
Iterator<? extends CharSequence> list2 = foo(list);
```

Dans ce cas, ? extends CharSequence est capturé par E

Capture (2)

La capture n’est pas toujours possible

Par exemple,

```
<E> void foo(List<E> list, List<E> list2) { … }

List<? extends CharSequence> list = …
List<? extends CharSequence> list2 = …
foo(list, list2);   // ne compile pas
```

Le compilateur ne sait pas s’il s’agit du même ‘?’ ou pas, donc il rejète la capture

Capture (3)

Même si la capture est possible, le résultat peut être inattendu

Par exemple,

```
<E> List<List<E>> foo(List<E> list) { … }
```

```
List<? extends CharSequence> list = …
foo(list);   // ok
List<List<? extends CharSequence>> list2 =
        foo(list);   // ne compile pas
```

Le type résultat est List<List<#capture>>, qui n'est pas un sous-type de List<List<? extends CharSequence>>

Capture - Pourquoi ?

Le code suivant est faux

```
List<List<String>> list = …
List<List<? extends CharSequence >>> list2 = list;    // ahhh
```

Même si List<String> est un sous-type de List<**? extends** CharSequence>, les types paramétrés ne sont pas covariant !

Par contre, on peut trouver un type qui se rapproche

```
List<?> list2 = list;                                          // ok
List<? extends List<?>> list2 = list;                          // ok
List<? extends List<? extends CharSequence>> list2 = list;     // ok
```

Attention !

Tout n’est pas paramétré !

On utilise des types/méthodes paramétrés

- Lorsqu’on utilise des types paramétrés déjà définie
- 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

exemple idiot:

```
<S extends Closeable> void idiot(S s)
```