

**Exemple (de l'API) :**

```
public interface Comparateur<T> {
    public int compare(T o1, T o2);
    public boolean equals(Object o);
```

- Que veut dire ce « <T> » ?
- **Comparator** est une interface **générique** : un type paramétrable par un autre type.

**Types génériques du JDK :**

- Les collections de Java  $\geq 5$  (interfaces et classes génériques).
- Ce fait seul suffit à justifier l'intérêt des génériques et leur introduction dans Java 5.
- Les interfaces fonctionnelles de Java  $\geq 8$  (pour les lambda expressions).
- **Optional**, **Stream**, **Future**, **CompletableFuture**, **ForkJoinTask**, ...

1. Ou « constructeur de type ». Mais cette terminologie est rarement utilisée en Java.

**Première solution : boîte universelle (polymorphisme par sous-typepage)**

```
class Boite { // très (trop ?) polymorphe
    public Object x; // contient des objets, supertype de tous les objets
    void echange(Boite autre) {
        Object ech = x; x = autre.x; autre.x = ech;
    }
}
```

**Réutilisabilité** : semble totale (on peut tout mettre dans la boîte).

**Inconvénient** : on ne sait pas (avant l'exécution<sup>1</sup>) quel type contient une telle boîte → difficile d'utiliser la valeur stockée (il faut tester et caster).

1. En fait, programmer des classes comme cette version de **Boîte** revient à abandonner le bénéfice du typage statique (pourtant une des forces de Java).

**Cas d'utilisation problématique :**

```
Boite b1 = new Boite();
Boite b2 = new Boite();
System.out.println(7 * (Integer) b1.x); // <- là c'est ok
b1.echange(b2);
System.out.println(7 * (Integer) b1.x); // <- ClassCastException !
```

En fait on aurait dû tester le type à l'exécution :

```
if ((b.x instanceof Integer) System.out.println(7 * (Integer) b.x);
```

... mais on préfèrerait vraiment que le code soit garanti par le compilateur<sup>1</sup>.

1. Remarque : dans cet exemple, probablement l'IDE (à défaut de javac) signalera que la conversion est hasardeuse.
- Normalement, on aura donc pensé à mettre **instanceof**. Il n'en reste pas moins que c'est un test à l'exécution qu'on aimeraient éviter (en plus d'être une lourdeur à l'écriture du programme).

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	
Compléments en 200	Aldric Degorre																			

## La bonne solution : hôte générique (→ polymorphisme générique)

```
class Boîte<C> {
    public C x;
    void échange(Boîte<C> autre) { C ech = x; x = autre.x; autre.x = ech; }

    ... // plus loin :
        Boîte<Integer> b1 = new Boîte<>(); Boîte<String> b2 = new Boîte<>();
        b1.x = 6; b2.x = "toto";
        System.out.println(7 * b1.x); // <- là c'est toujours ok (et sans cast, SyP !)
        // b1.echange(b2); // <- ici erreur à la compilation ! (ouf !)
        System.out.println(7 * b1.x);
```

La généricité consiste à introduire des types dépendants d'un paramètre de type.

La concrétisation du paramètre est vérifiée dès de la compilation<sup>1</sup> et uniquement à la compilation. Celle-ci est oubliée aussitôt<sup>2</sup> (**effacement de type** / type érasure).

1. Or le plus tôt on détecte une erreur, le mieux c'est!
2. Conséquence : les objets de classe générique ne savent pas avec quel paramètre ils ont été instanciés.

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	
Compléments en 200	Aldric Degorre																			

- **Type générique** : type (classe ou interface) dont la définition fait intervenir un **paramètre de type** (dans les exemples, c'était **T** et **C**).
- **À la définition** du type générique, le paramètre introduit dans son en-tête peut ensuite être utilisé dans son corps comme si c'était un vrai nom de type.

```
class Triplet<T,U,V> { // introduction de T, U et V
    T elt1; U elt2; V elt3; // utilisation de T, U et V
    public Triplet(T e1, U e2, V e3) { elt1 = e1; elt2 = e2; elt3 = e3; }
}
```

**Attention :** **T** | **U** | **V** ne sont utilisables qu'en contexte non statique : en effet, ils représentent des types choisis pour chaque instance de **Triplet** ⇒ il faut donc être dans le contexte d'une instance pour qu'ils aient du sens.

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	
Compléments en 200	Aldric Degorre																			

- Utilisation de classe générique par instantiation directe :

```
// à partir de Java 5 :
Triplet<String, String, Integer> t1 = new Triplet<String, String, Integer>("Marcel", "Durrand", 23);

// à partir de Java 7 :
Triplet<String, String, Integer> t2 = new Triplet<String, String, Integer>("Marcel", "Durrand", 23);

// à partir de Java 10 (si t3 est une variable locale) :
var t3 = new Triplet<String, String, Integer>("Marcel", "Durrand", 23);
```

Le type de **t1**, **t2** et **t3** est le type paramétrisé **Triplet<String, String, Integer>**.

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	
Compléments en 200	Aldric Degorre																			

- À l'usage, le type générique sert de constructeur de type : on remplace le paramètre par un type concret et on obtient un **type paramétrisé**.

Exemple : **List<String>** un des types paramétrés que **List** permet de construire.

- Le type concret substituant le paramètre doit être un type **référence** : **Triplet<int, boolean, char>** est interdit<sup>1</sup> !

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Introduction	
Compléments en 200	Aldric Degorre																			

1. Pour l'instant. Il semble qu'il soit prévu de permettre cela dans une prochaine version de Java.

Types génériques et types paramétrés		Parallèle entre type générique et méthode	
Compléments en 200 Aldric Degorre	Usage de base (4)	Compléments en 200 Aldric Degorre	La déclaration et l'utilisation des types génériques rappellent celles des méthodes.
<p>Utilisation de classe générique par extension non générique (<b>spécialisation</b>) :</p> <pre>class TroisChars extends Triplet&lt;Char, Char, Char&gt; {     public TroisChars(Char x, Char y, Char z) { super(x,y,z); }</pre> <p>TroisChars étend la classe paramétrée Triplet&lt;Char, Char, Char&gt;.</p> <p>Variante, spécialisation partielle :</p> <pre>class DeuxCharsEtAutre&lt;T&gt; extends Triplet&lt;Char, Char, T&gt; {     public DeuxCharsEtAutre(Char x, Char y, T z) { super(x,y,z); }</pre> <p>La classe générique DeuxCharsEtAutre&lt;T&gt; étend la classe générique partiellement paramétrée Triplet&lt;Char, Char, T&gt;.</p>	<p><b>Similitudes :</b></p> <ul style="list-style-type: none"> <li>introduction des paramètres (de type ou de valeur) dans l'en-tête de la déclaration;</li> <li>utilisation des noms des paramètres dans le corps de la déclaration seulement;</li> <li>pour utiliser le type générique ou appeler la méthode, on passe des <u>concrétisations des paramètres</u>.</li> </ul> <p><b>Principales différences :</b></p> <ul style="list-style-type: none"> <li>Les paramètres des génériques représentent des <u>types</u> alors que ceux des méthodes représentent des <u>valeurs</u>.</li> <li>Pour les paramètres de type, le « remplacement » 1 a lieu à la compilation.</li> <li>Pour les paramètres des méthodes, remplacement par une valeur à l'exécution.</li> </ul> <p>1. Rappel : remplacement oublié, effacé, aussitôt que la vérification du bon typage a été faite.</p>	<p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Introduction Effacement de type Collectives Optionnels Lambdpressions Les "streams" Instance des génératives vs. concrétes des tableaux Villecards Concurrence Interfaces graphiques</p>	<p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Introduction Effacement de type Collectives Optionnels Lambdpressions Les "streams" Instance des génératives vs. concrétes des tableaux Villecards Concurrence Interfaces graphiques</p>
<p>Types bruts</p>	<p>Compléments en 200 Aldric Degorre</p> <p>Un nom de type générique seul, sans paramètre (comme « Triplet »), est aussi un type légal, appelé un <b>type brut (raw type)</b>. Son utilisation est <b>fortement déconseillée</b>, mais elle est permise pour assurer la compatibilité ascendante<sup>1</sup>.</p> <p>Un type brut est <u>super-type direct</u><sup>2</sup> de tout type paramètre correspondant (ex : Triplet est supertype direct de Triplet&lt;Number, Object, String&gt;).</p> <p>Pour faciliter l'écriture, le <u>downcast implicite</u><sup>3</sup> est malgré tout possible :</p> <pre>List l1 = new ArrayList(); // déclaration de l1 avec raw type List&lt;Integer&gt; l2 = l1; // downcast implicite de l1 vers type paramétré</pre> <p>compile avec l'avertissement <b>unchecked conversion</b> sur la deuxième ligne.</p>	<p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Introduction Effacement de type Collectives Optionnels Lambdpressions Les "streams" Instance des génératives vs. concrétes des tableaux Villecards Concurrence Interfaces graphiques</p>	<p>Il est aussi possible d'introduire un paramètre de type dans la <u>signature</u> d'une méthode (possible aussi dans une classe non générique) :</p> <pre>static &lt;E&gt; List&lt;E&gt; inverserList(List&lt;E&gt; l) { ... ; E x = get(0); ... ; }</pre> <p>Dans l'exemple ci-dessus, on garantit que la liste renvoyée par <b>inverserList()</b> a le même type d'éléments que celle donnée en paramètre.</p> <p><b>Usages possibles :</b></p> <ul style="list-style-type: none"> <li>contraindre plusieurs types apparaissant dans la signature de la méthode à être le même type, sans pour autant dire lequel;</li> <li>introduire localement un nom de type utilisable dans le corps de la méthode (type non défini), mais dont les contraintes sont connues, ex : type intersection, voir plus loin).</li> </ul> <p><b>Remarque :</b> il est donc possible d'écrire une méthode statique générique et son corps (contexte statique) pourra utiliser le paramètre introduit, contrairement aux paramètres de type introduits au niveau de la classe ou de l'interface.</p>

	<a href="#">Introduction</a>	<a href="#">Généralités</a>	<a href="#">Style</a>	<a href="#">Objets et classes</a>	<a href="#">Types et polymorphisme</a>	<a href="#">Héritage</a>	<a href="#">Généricité</a>	<a href="#">Introduction</a>	<a href="#">Généralités</a>	<a href="#">Style</a>	<a href="#">Objets et classes</a>	<a href="#">Types et polymorphisme</a>	<a href="#">Héritage</a>	<a href="#">Généricité</a>
	Aldric Degorre								Aldric Degorre					

- Pour définir des bornes supérieures multiples (p. ex. pour implémenter de multiples interfaces), les supertypes sont séparés par le symbole « & » :
   
**class RunWithPriorityList<T extends Comparable<T> & Runnable> implements List<T>**

« **Comparable<T> & Runnable** » est un **type intersection**<sup>1</sup>, il est sous-type direct de **Comparable<T>** et de **Runnable**.

Ainsi, **T** est sous-type de l'intersection (et donc de de **Comparable<T>** et de **Runnable**).

- 
1. Remarque : c'est le seul contexte où on peut écrire un type intersection (type non dénotable). Ainsi, il n'est pas possible de déclarer explicitement une variable de type intersection. Implicitement, à l'aide d'une méthode générique et du mot-clé **var**, cela est cependant possible : **public static <T extends A & B> T intersectionFactory(...){ ... }**
- plus loin : **var x = intersectionFactory(...); //x est de type A & B**
- La technique assez « tirée par les cheveux » et d'utilité toute relative....

	<a href="#">Introduction</a>	<a href="#">Généralités</a>	<a href="#">Style</a>	<a href="#">Objets et classes</a>	<a href="#">Types et polymorphisme</a>	<a href="#">Héritage</a>	<a href="#">Généricité</a>	<a href="#">Introduction</a>	<a href="#">Généralités</a>	<a href="#">Style</a>	<a href="#">Objets et classes</a>	<a href="#">Types et polymorphisme</a>	<a href="#">Héritage</a>	<a href="#">Généricité</a>
	Aldric Degorre								Aldric Degorre					

- On peut prolonger l'analogie avec les méthodes et leurs paramètres : en effet, les paramètres des méthodes sont eux-mêmes « bornés » par les types déclarés dans la signature.

- L'idée principale du phénomène appelé **effacement de type** (ou **type erasure**) c'est que le système de types de la JVM ne connaît que les types réifiables. Autrement dit : la paramétrisation générique n'a pas d'impact à l'exécution. Plus de détails juste après.

- Effacement d'un type** : sur-approximation permettant d'obtenir un **type réifiable** (i.e. : « classique », façon Java 4) à partir de n'importe quel type. Plus précisément (JLS 4.6) :
- L'effacement d'un type générique ou paramétré de forme **G<...>**, est le type brut **G**.
  - L'effacement d'une variable de type est l'effacement de sa borne supérieure.
  - L'effacement de tout autre type **T** est **T**.

L'idée principale du phénomène appelé **effacement de type** (ou **type erasure**) c'est que le système de types de la JVM ne connaît que les types réifiables. Autrement dit : la paramétrisation générique n'a pas d'impact à l'exécution.

## Effacement de type (type erasure)

Explication

L'effacement de type commence en réalité dès la compilation.

**Descripteur de méthode** : information, dans la **table des constantes** d'une classe compilée, permettant d'identifier une méthode (peut-être surchargée) de façon unique. Tout appel de méthode<sup>1</sup> dans le code-octet fait référence à un tel descripteur. Or un descripteur consiste en un couple : (nom de méthode, types réifiables des paramètres).

**Conséquences :**

- aucun paramètre de type n'est réellement passé aux constructeurs (et méthodes)
  - les objets ne stockent donc pas les valeurs de leurs paramètres de types. Ils ne peuvent donc connaître que leur classe<sup>2</sup>. Les « objets paramétrés » n'existent pas.
1. `invokestatic`, `invokespecial` et `invokeinterface` prennent un index de la table des constantes comme paramètre.
2. Qui n'existe qu'en un seul exemplaire dans la mémoire, quelle que soit la paramétrisation.

## Effacement de type (type erasure)

Explication

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Introduction

Effacement de type

Collections

Outils

Lambdpressions

Les « strings »

Instanciation des génératrices vs. covariance des tableaux

Wildcards

Concurrence

Interfaces

graphiques

Annotations

Lambdpressions

Les « strings »

Instanciation des génératrices vs. covariance des tableaux

Wildcards

Concurrence

Interfaces

graphiques

Annotations

Lambdpressions

Les « strings »

Instanciation des génératrices vs. covariance des tableaux

Wildcards

Concurrence

Interfaces

graphiques

## Effacement de type (type erasure)

Conséquences concrètes

À l'exécution, il est impossible de savoir comment un paramètre de type a été concrétisé.

**En particulier :**

- Faut d'être raisonnablement exécutables, ces expressions ne compilent pas :
  - `x instanceof P` (avec `P` paramètre de type);
  - `x instanceof Type<Y>`<sup>1</sup> (avec `Y` type quelconque).

- On ne peut pas déclarer d'exception générique `Ex<T>` car `catch(Ex<X> ex)` ne serait pas non plus évaluable (même problème qu'`instanceof`).
 

```
// ne compile pas
public class GenericException<T> extends Exception { ... }
```

## Effacement de type (type erasure)

Explication

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Introduction

Effacement de type

Collections

Outils

Lambdpressions

Les « strings »

Instanciation des génératrices vs. covariance des tableaux

Wildcards

Concurrence

Interfaces

graphiques

Annotations

Lambdpressions

Les « strings »

Instanciation des génératrices vs. covariance des tableaux

Wildcards

Concurrence

Interfaces

graphiques

Annotations

Lambdpressions

Les « strings »

Instanciation des génératrices vs. covariance des tableaux

Wildcards

Concurrence

Interfaces

graphiques

## Effacement de type (type erasure)

Conséquences concrètes

Autres conséquences directes de l'effacement dans les descripteurs :

- Dans une classe, on ne peut pas définir plusieurs méthodes dont les signatures seraient identiques après effacement (leurs descripteurs seraient identiques).
 

```
// ne compile pas
public class A {
    List<Integer> f() { return null; }
    List<String> f() { return null; }
}
```

Autres conséquences directes de l'effacement dans les descripteurs :

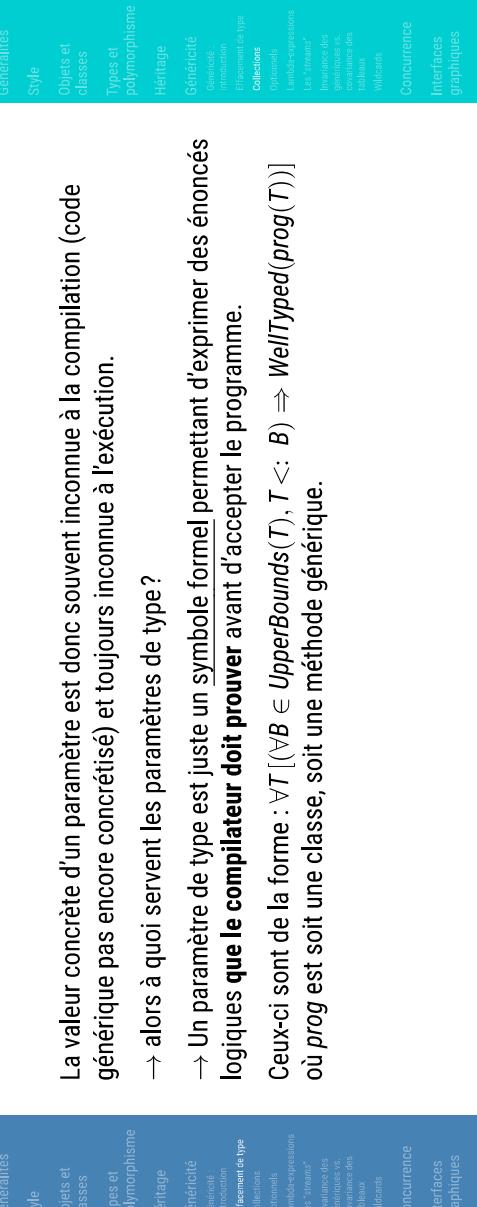
- Une classe ne peut pas implémenter plusieurs fois une interface générique avec des paramètres différents (on se retrouverait dans le cas précédent).
 

```
// ne compile pas
public class A implements ArrayList<Integer> implements List<String> { ... }
```

1. Mais `x instanceof Type` compile, c'est un des rares cas où on tolère le raw type.

## Effacement de type (type erasure)

Mais quand les paramètres de type servent-ils alors ?

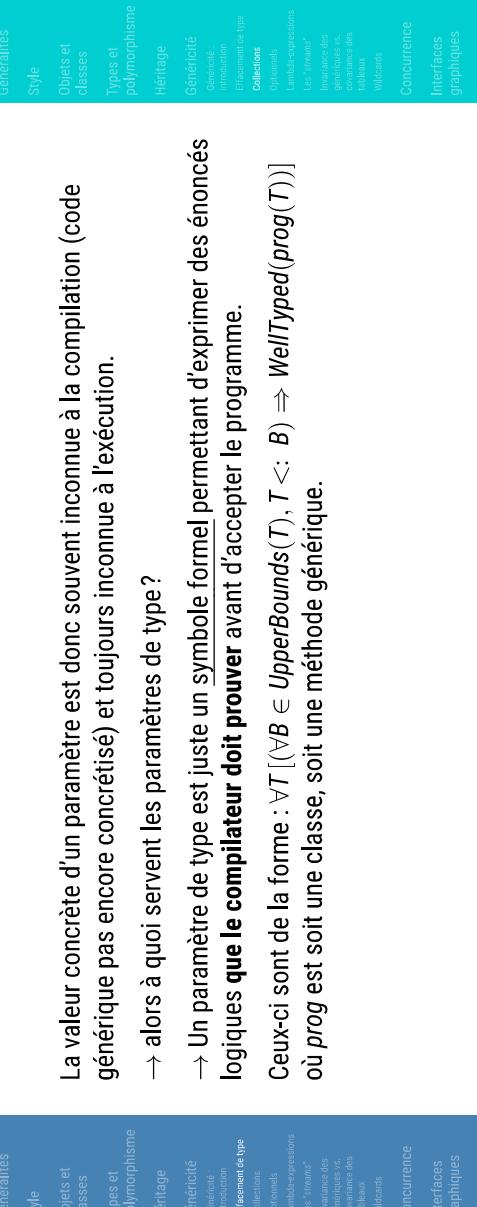


La valeur concrète d'un paramètre est donc souvent inconnue à la compilation (code générique pas encore concrétisé) et toujours inconnue à l'exécution.

→ alors à quoi servent les paramètres de type ?

→ Un paramètre de type est juste un symbole formel permettant d'exprimer des énoncés logiques que le compilateur doit prouver avant d'accepter le programme.

Ceux-ci sont de la forme :  $\forall T \ [(\forall B \in UpperBounds(T), T <: B) \Rightarrow WellTyped(prog(T))]$  où *prog* est soit une classe, soit une méthode générique.



La valeur concrète d'un paramètre est donc souvent inconnue à la compilation (code générique pas encore concrétisé) et toujours inconnue à l'exécution.

→ alors à quoi servent les paramètres de type ?

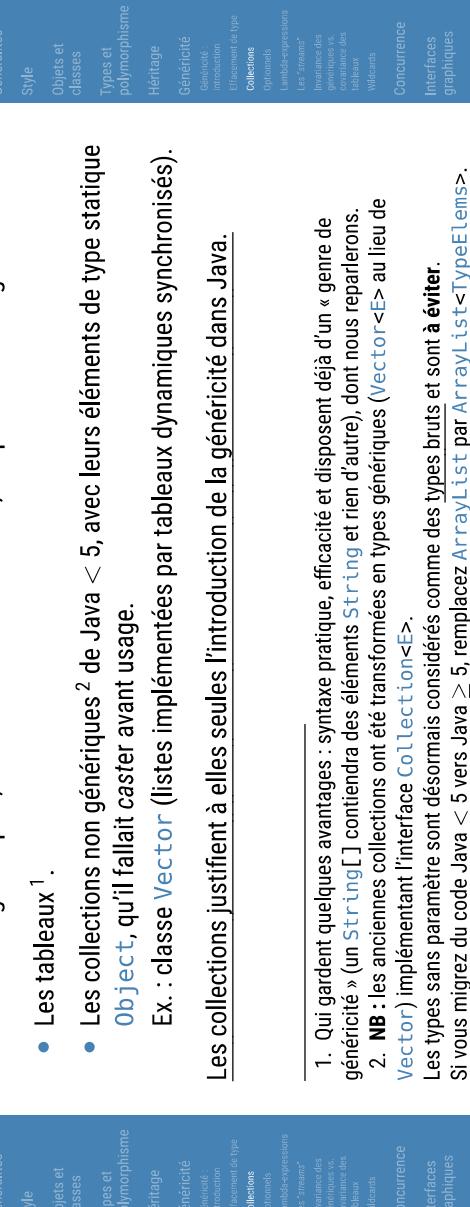
→ Un paramètre de type est juste un symbole formel permettant d'exprimer des énoncés logiques que le compilateur doit prouver avant d'accepter le programme.

Ceux-ci sont de la forme :  $\forall T \ [(\forall B \in UpperBounds(T), T <: B) \Rightarrow WellTyped(prog(T))]$  où *prog* est soit une classe, soit une méthode générique.

## Collections : pourquoi ?

### La hiérarchie des sous-interfaces d'Iterable

Interfaces de java.util et java.util.concurrent<sup>1</sup>



Les collections génériques, introduites dans Java SE 5, remplacent avantageusement :

- Les tableaux<sup>1</sup>.

- Les collections non génériques<sup>2</sup> de Java < 5, avec leurs éléments de type statique **Object**, qu'il fallait caster avant usage.

Ex. : classe **Vector** (listes implémentées par tableaux dynamiques synchronisés).

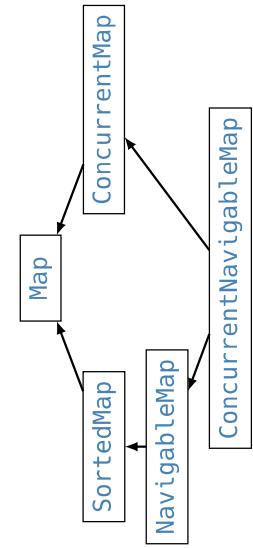
Les collections justifient à elles seules l'introduction de la généricité dans Java.

- Qui gardent quelques avantages : syntaxe pratique, efficacité et disposent déjà d'un « genre de généricité » (un `String[]` contiendra des éléments `String` et rien d'autre), dont nous reparlerons.
- NB :** les anciennes collections ont été transformées en types génériques (`Vector<E>` au lieu de `Vector`) implémentant l'interface `Collection<E>`.

Les types sans paramètre sont désormais considérés comme des types bruts et sont à éviter.  
Si vous migrez du code Java < 5 vers Java  $\geq$  5, remplacez `ArrayList` par `java.util.List<TypeElements>`.

<sup>1</sup>. Autres sous-interfaces dans `java.nio.file` et `java.beans.beancontext`.

## La hiérarchie des sous-interfaces de Map

Interfaces de java.util et java.util.concurrent<sup>1</sup>

Chacune de ces interfaces possède une ou plusieurs implémentations.

- 1. Autres dans javax.script, javax.xml.ws.handler et javax.xml.ws.handler.soap.

## Compléments en 200

Aldric Degorre

Introduction	Introduction
Généralités	Généralité
Style	Introduction
Objets et classes	Éléments de type
Types et polymorphisme	collections
Héritage	Objet
Généricité	Lambda-expressions
Généralité	Les "strangs"
Introduction	Instance des généraux vs. covariance des tableaux
Éléments de type	Villecards
collections	Concurrent
Objet	Interfaces graphiques

Introduction	Introduction
Généralités	Généralité
Style	Introduction
Objets et classes	Éléments de type
Types et polymorphisme	collections
Héritage	Objet
Généricité	Lambda-expressions
Généralité	Les "strangs"
Introduction	Instance des généraux vs. covariance des tableaux
Éléments de type	Villecards
collections	Concurrent
Objet	Interfaces graphiques

## Compléments en 200

Aldric Degorre

Introduction	Introduction
Généralités	Généralité
Style	Introduction
Objets et classes	Éléments de type
Types et polymorphisme	collections
Héritage	Objet
Généricité	Lambda-expressions
Généralité	Les "strangs"
Introduction	Instance des généraux vs. covariance des tableaux
Éléments de type	Villecards
collections	Concurrent
Objet	Interfaces graphiques

## Itérateurs ?

### Un itérateur :

- sert à parcourir un itérable et est habituellement utilisé implicitement;
- s'instancie en appelant la méthode `iterator` sur l'objet à parcourir;
- est un objet respectant l'interface suivante :

```

public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); // opération optionnelle
}
  
```

`remove`, si implementée, permet de supprimer un élément en cours de parcours sans provoquer `ConcurrentModificationException` (au contraire des méthodes de l'itérable). Cette possibilité justifie de créer une variable pour manipuler explicitement l'itérateur (sinon, on préfère `for-each`).

## L'interface Iterable (2)

### soit en utilisant explicitement l'itérateur (rare, mais utile pour accès en écriture) :

```

Iterator<String> it = monIterable.iterator();
while (it.hasNext()) {
    String s = it.next();
    if (s.equals("À enlever"))
        it.remove();
    else System.out.println("On garde: " + s);
}
  
```

Remarque : la construction `for-each` et la méthode `forEach` ne permettent qu'un parcours en lecture seule.

- soit en réduisant un `Stream` (cf. chapitre dédié) basé sur cet `Iterable` :

```

miaCollection.stream()
    .filter(x -> !x.equals("À enlever"))
    .forEach(System.out::println());
  
```

Les paramètres des méthodes de `Stream` sont typiquement des `lambda-expressions`.

- En réalité, pour des raisons assez obscures, la méthode `stream` n'existe que dans la sous-interface `Collection`. Mais il est facile de programmer une méthode équivalente pour `Iterable`.

## L'interface Set

## L'interface Collection

Compléments en 200

Aldric Degorre

Une **Collection** est un **Iterable** muni des principales opérations ensemblistes :

```
public interface Collection<E> extends Iterable<E> {  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element); // opération optionnelle  
    boolean remove(Object element); // opération optionnelle  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<?> extends E> c); // opération optionnelle  
    boolean removeAll(Collection<?> c); // opération optionnelle  
    boolean retainAll(Collection<?> c); // opération optionnelle  
    void clear();  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
    default Stream<E> stream() { ... }  
}
```

L'API ne fournit pas d'implémentation directe de **Collection**, mais plutôt des collections spécialisées, décrites dans la suite.

## L'interface Set (1)

Aldric Degorre

- Pas de méthodes autres que celles héritées de **Collection**.
- **Déférence : le contrat de Set garantit l'unicité de ses éléments (pas de doublon).**

**Exemple :**

```
Set<Integer> s = new HashSet<Integer>();  
s.add(1); s.add(2); s.add(3);  
for (int i : s) System.out.print(i + " , ");
```

Ceci affichera : **1, 2, 3,**  
La classe **HashSet** est une des implémentations de **Set** fournies par Java. C'est celle que vous utiliserez le plus souvent.

**Unicité ?** un élément **x** est unique si pour tout autre élément **y**, **x.equals(y)** retourne **false**.  
⇒ importance d'avoir une redéfinition correcte de **equals()**.

## L'interface List (1)

Aldric Degorre

**List** : c'est une **Collection** ordonnée avec possibilité de doublons. C'est ce qu'on utilise le plus souvent. Permet d'abstraire les notions de tableau et de liste chainée.

**Fonctionnalités principales :**

- **acès positionnel** (on peut accéder au *i*-ième élément)
- **recherche** (si on connaît un élément, on peut demander sa position)

```
public interface List<E> extends Collection<E> {  
    // Positional access  
    E get(int index);  
    E set(int index, E element); // optional  
    boolean add(E element); // optional  
    void add(int index, E element); // optional  
    E remove(int index); // optional  
    boolean addAll(int index, Collection<? extends E> c); // optional  
    // Search  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
    ...
```

## L'interface SortedSet

Aldric Degorre

Comme **Set**, mais les éléments sont triés.  
... ce qui permet d'avoir quelques méthodes en plus.

```
public interface SortedSet<E> extends Set<E> {  
    // Range-view  
    SortedSet<E> subSet(E fromElement, E toElement);  
    SortedSet<E> headSet(E toElement);  
    SortedSet<E> tailSet(E fromElement);  
  
    // Endpoints  
    E first();  
    E last();  
  
    // Comparator access  
    Comparator<? super E> comparator();  
}
```

**Implémentation typique : classe TreeSet.**

Compléments en 200

Aldric Degorre

**List** : c'est une **Collection** ordonnée avec possibilité de doublons. C'est ce qu'on utilise le plus souvent. Permet d'abstraire les notions de tableau et de liste chainée.

**Fonctionnalités principales :**

- **acès positionnel** (on peut accéder au *i*-ième élément)
- **recherche** (si on connaît un élément, on peut demander sa position)

```
public interface List<E> extends Collection<E> {  
    // Positional access  
    E get(int index);  
    E set(int index, E element); // optional  
    boolean add(E element); // optional  
    void add(int index, E element); // optional  
    E remove(int index); // optional  
    boolean addAll(int index, Collection<? extends E> c); // optional  
    // Search  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
    ...
```

## L'interface List (1)

Aldric Degorre

**List** : c'est une **Collection** ordonnée avec possibilité de doublons. C'est ce qu'on utilise le plus souvent. Permet d'abstraire les notions de tableau et de liste chainée.

**Fonctionnalités principales :**

- **acès positionnel** (on peut accéder au *i*-ième élément)
- **recherche** (si on connaît un élément, on peut demander sa position)

```
public interface List<E> extends Collection<E> {  
    // Positional access  
    E get(int index);  
    E set(int index, E element); // optional  
    boolean add(E element); // optional  
    void add(int index, E element); // optional  
    E remove(int index); // optional  
    boolean addAll(int index, Collection<? extends E> c); // optional  
    // Search  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
    ...
```

## L'interface List (1)

Aldric Degorre

**List** : c'est une **Collection** ordonnée avec possibilité de doublons. C'est ce qu'on utilise le plus souvent. Permet d'abstraire les notions de tableau et de liste chainée.

**Fonctionnalités principales :**

- **acès positionnel** (on peut accéder au *i*-ième élément)
- **recherche** (si on connaît un élément, on peut demander sa position)

```
public interface List<E> extends Collection<E> {  
    // Positional access  
    E get(int index);  
    E set(int index, E element); // optional  
    boolean add(E element); // optional  
    void add(int index, E element); // optional  
    E remove(int index); // optional  
    boolean addAll(int index, Collection<? extends E> c); // optional  
    // Search  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
    ...
```

**Mais aussi :**

- itérateurs plus riches (peuvent itérer en arrière)
- « vues » de sous-listes<sup>1</sup>

```
...
// Iteration
ListIterator<E> listIterator();
ListIterator<E> listIterator(int index);

// Range-view
List<E> subList(int from, int to);
}
```

1. Vue d'un objet `o` : objet `v` donnant accès à une partie des données de `o` sans en être une copie (partielle), les modifications des 2 objets restent liées.

Ceci affichera :

```
1, 2, 3, 1
3e élément: 3
Nouveau 3e élément: 9
```

Un itérateur de liste sert à parcourir une liste. Il fait la même chose qu'un itérateur ; mais aussi quelques autres opérations, comme :

- parcourir à l'envers
- ajouter/modifier des éléments en passant
- un itérateur de liste est un objet respectant l'interface suivante :

```
public interface ListIterator<E> extends Iterator<E> {
    void add(E e);
    boolean hasPrevious();
    int nextIndex();
    E previous();
    int previousIndex();
    void set(E e);
}
```

Une Queue représente typiquement une collection d'éléments en attente de traitement (typiquement FIFO : *first in, first out*).

Opérations de base : insertion, suppression et inspection.

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

Exemple : la classe `PriorityQueue` présente ses éléments selon l'ordre naturel de ses éléments (ou un autre ordre si spécifié).

Deque = « double ended queue ».

C'est comme une Queue, mais enrichie afin d'accéder à la collection aussi bien par le début que par la fin.

Le même Deque peut ainsi aussi bien servir de structure FIFO que LIFO (last in, first out).

```
public interface Queue<E> extends Collection<E> {
    boolean addFirst(E e);
    boolean addLast(E e);
    Iterator<E> descendingIterator();
    E getFirst();
    E getLast();
    boolean offerFirst(E e);
    boolean offerLast(E e);
    E peekFirst();
    E peekLast();
}
```

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Introduction

Effacement de type

collections

Généralité

Introduction

Effacement de type

collections

Généralité

Introduction

Objet

Listes

Lambda-expressions

Les "strangs"

Inégalités vs. covariance des tableaux

Wildcards

...

```
...
    E pollFirst();
    E pollLast();
    E pop();
    void push(E e);
    E removeFirst();
    E removeLast();
    E removeFirstOccurrence(Object o);
    E removeLastOccurrence(Object o);
}
// plus méthodes héritées
```

**Implémentations typiques :** [ArrayDeque](#), [LinkedList](#)

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Introduction

Effacement de type

collections

Généralité

Introduction

Objet

Listes

Lambda-expressions

Les "strangs"

Inégalités vs. covariance des tableaux

Wildcards

...

```
...
    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();
    // Collection views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Une Map est un ensemble d'associations (clé → valeur), où chaque clé ne peut être associée qu'à une seule valeur.

Nombreuses méthodes communes avec l'interface Collection, mais particularités.

```
public interface Map<K,V> {
    // Basic operations
    V put(K key, V value);
    V get(K key, V value);
    V remove(K key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
}
```

## L'interface SortedMap

Compléments  
en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Introduction

Élément de type

Collections

Options

Lambdagepressions

Les "strangs"

Innatenet des génératrices vs covariance des tableaux

Villecards

Concurrence

Interfaces graphiques

Aldric Degorre

**SortedMap** est à **Map** ce que **SortedSet** est à **Set** : ainsi les associations sont ordonnées par rapport à l'ordre de leurs les clés.

```
public interface SortedMap<K, V> extends Map<K, V>
{
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

Implémentation typique : **TreeMap**.

Fabriques statiques du JDK → alternative intéressante aux constructeurs de collections :

- Nombreuses dans la classe **Collections**<sup>1</sup> : collections vides, conversion d'un type de collection vers un autre, création de vues avec telle ou telle propriété, ...
- Pour obtenir une liste depuis un tableau : **Arrays.asList(tableau)**.
- Fabriques statiques de collections immuables, nommées « **of** », dans les interfaces **List**, **Set** et **Map** (Java ≥ 9) :

```
List<String> semaine = List.of("lundi", "mardi", "mercredi", "jeudi",
"vendredi", "samedi", "dimanche");
Map<String, String> instruments = Map.of("guitare", "cordes", "piano", "cordes",
"clarinette", "vent");
Set<Integer> premiers = Set.of(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31);
```

Appeler une fabrique plutôt qu'un constructeur évite de choisir une implémentation : on fait confiance à la fabrique pour choisir la meilleure pour les paramètres donnés.

1. Noter le 's'

Compléments  
en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Introduction

Élément de type

Collections

Options

Lambdagepressions

Les "strangs"

Innatenet des génératrices vs covariance des tableaux

Villecards

Concurrence

Interfaces graphiques

Aldric Degorre

**Le problème** : représenter de façon non ambiguë le fait qu'une méthode puisse retourner (ou qu'une variable puisse contenir) aussi bien une valeur qu'une absence de valeur.

**Exemples :**

- résultat du dépilement d'une pile (pile peut-être vide)
- recherche d'un élément satisfaisant un certain critère dans une liste (liste ne contenant pas forcément un tel élément)
- identité de la personne ayant réservé un certain siège dans un avion (siège peut-être pas encore réservé)

## Les optionnels

Compléments  
en POO

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Introduction

Élément de type

Collections

Options

Lambdagepressions

Les "strangs"

Innatenet des génératrices vs covariance des tableaux

Villecards

Concurrence

Interfaces graphiques

Aldric Degorre

- Avantages des tableaux** : syntaxe légère et efficacité.
- Avantages des collections génériques** :
- polyvalence (plein de collections adaptées à des cas différents)
  - polymorphisme via les interfaces de collections
  - sûreté du typage (« vraie » généricité)

**Conclusion, utilisez les collections**, sauf :

- si vous prototyppez un programme très rapidement et vous appréciez la simplicité
- si vous souhaitez optimiser la performance au maximum<sup>1,2</sup>

1. Cela dit, les méthodes du *collection framework*, sont écrites et optimisées par des experts et déjà testées par des milliers de programmeurs. Pensez-vous faire mieux ? (peut-être, si besoin très spécifique)
2. Mais pourquoi programmez-vous en Java alors ?

## Solutions (pas très bonnes) :

- retourner une valeur qui peut être **null** (« **nullable** »). **Inconvénients :**
  - si `getVal()` retourne une valeur nullable, l'appel `getVal().doSomething()` peut causer une `NullPointerException`. En toute généralité, cette exception peut se déclencher bien plus loin dans le programme (débogage difficile).
  - null** peut aussi représenter une variable pas encore initialisée (ambiguïté)
  - lancer une exception pour l'absence de valeur.

- Inconvénient :** obligation d'utiliser des **try catch** (lourdeur syntaxique, s'intègre mal au flot du programme) ; mécanisme coûteux à l'exécution.
- Utiliser une liste à 0 ou 1 élément.
  - Inconvénient :** le type liste autorise aussi les listes à 2 éléments ou plus.
1. L'invention de **null** a été qualifiée *a posteriori* par son auteur, Tony Hoare, d'*« erreur à un milliard de dollars »*, ce n'est pas peu dire !

Compléments  
en 200

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Introduction

Effacement de type

Collections

Optionnals

Lambdagepressions

Les "strangs"

Instanciation des généricités vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Introduction

Effacement de type

Collections

Optionnals

Lambdagepressions

Les "strangs"

Instanciation des généricités vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Introduction

Effacement de type

Collections

Optionnals

Lambdagepressions

Les "strangs"

Instanciation des généricités vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Introduction

Effacement de type

Collections

Optionnals

Lambdagepressions

Les "strangs"

Instanciation des généricités vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

## Exemple :

```
Optional<Client> maybeRes = seat.getReservation();
if (maybeRes.isPresent()) {
    Client res = maybeRes.get(); // on est sûr qu'il n'y a pas d'exception
    res.sendReminder(); // aucun risque de NPE car res est résultant de get()
}
```

**Remarque :** cela peut aussi s'écrire

```
seat.getReservation().ifPresent(res -> res.sendReminder());
```

## Une solution pas trop mauvaise :<sup>1</sup> la classe `java.util.Optional`<sup>2</sup>

- une instance de `Optional<T>` est une valeur représentant soit une instance présente de `T` soit l'absence d'une instance (par définition, de façon non ambiguë).
- ainsi, instance de `Optional<T>` contient juste un champ de type `T`
- La présence d'un élément se teste en appelant `isPresent`.
- On accède à la valeur de l'élément via la méthode `get` qui ne retourne jamais **null** mais lance `NoSuchElementException` si l'élément est absent.

Bien qu'`Optional<T>` n'implémente pas `Collection<T>`, il est pertinent d'imaginer `Optional<T>` comme un type représentant des collections de 0 ou 1 élément.

- Les valeurs nullables gardent quelques avantages sur `Optional` : pas besoin d'appeler `isPresent` et `get`). De plus, même une expression de type `Optional` est elle-même nullable...

Des alternatives existent (hors Java : notamment systèmes de types contenant des types non nullables, en Java : annotations `@NotNull` et `@Nullable` + outil d'analyse statique).

- Inspirée des langages fonctionnels : la classe `Option` en Scala, la monade `Maybe` en Haskell

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Introduction

Effacement de type

Collections

Optionnals

Lambdagepressions

Les "strangs"

Instanciation des généricités vs. covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

## Exemple :

```
Optional<Client> maybeRes = seat.getReservation();
if (maybeRes.isPresent()) {
    Client res = maybeRes.get(); // on est sûr qu'il n'y a pas d'exception
    res.sendReminder(); // aucun risque de NPE car res est résultant de get()
}
```

**Remarque :** cela peut aussi s'écrire

```
seat.getReservation().ifPresent(res -> res.sendReminder());
```

## Les optionnels

Ah... et comment les instancier au fait ?

La classe `Optional` est munie de 2 fabriques statiques principales :

- <`T`> `Optional<T> of(T elem)` : si `elem` est non **null**, retourne un optional contenant `elem` (sinon `NullPointerException`)
- <`T`> `Optional<T> empty()` : retourne un optional vide du type désiré (en fonction du contexte)

## Exemple :

```
public static Optional<Integer> findIndex(int[] elems, int elem) {
    for (int i = 0; i < elems.length; i++) {
        if (elems[i] == elem) return Optional.of(i);
    }
    return Optional.empty();
}
```

## Les optionnels

Optionnels spécialisés

Introduction		Fonctions de première classe et fonctions d'ordre supérieur	
Généralités		Le besoin (1)	
Style	Objets et classes	Appeler 5 fois une méthode <code>f</code> déjà connue :	
		<code>f(); f(); f(); f(); f();</code>	
Types et polymorphisme	Héritage	Appeler <code>f</code> un nombre de fois inconnu à l'avance :	
collectives	Généricité	<code>// Facile ! On ajoute un paramètre int : public static void repeat(int n) { for (int i = 0; i &lt; n; i++) f(); }</code>	
Lambda-expressions	Introduction		
Les "strangs"	Élément de type		
Initialisation des génériques vs. covariance des tableaux	Collectives		
Villecards	Outils		
Concurrence	Lambda-expressions		
Interfaces graphiques	Heritage		

Une dernière remarque sur le sujet :

- En plus de la classe générique `Optional<T>`, `java.util` contient aussi les classes non génériques `OptionalInt`, `OptionalLong` et `OptionalDouble`;
- celles-ci sont sémantiquement équivalentes à, respectivement `Optional<Integer>`, `Optional<Long>` et `Optional<Double>`.

L'intérêt est d'économiser les indirections (le fait de suivre 2 pointeurs pour obtenir une valeur primitive) et les allocations multiples (celle de l'`Optional` et celle du `Integer` par exemple) pour le cas des types primitifs.

Introduction		Fonctions de première classe et fonctions d'ordre supérieur	
Généralités		Le besoin (1)	
Style	Objets et classes	Appeler 5 fois une méthode <code>f</code> déjà connue :	
		<code>f(); f(); f(); f(); f();</code>	
Types et polymorphisme	Héritage	Appeler <code>f</code> un nombre de fois inconnu à l'avance :	
collectives	Généricité	<code>// Hm... il faudrait passer une méthode en paramètre ? Tentative : public static void repeat5(?? f) { // quel type pour f ? for (int i = 0; i &lt; 5; i++) f(); // si f une variable, "f()" -&gt; erreur de syntaxe }</code>	
Lambda-expressions	Introduction		
Les "strangs"	Élément de type		
Initialisation des génériques vs. covariance des tableaux	Collectives		
Villecards	Outils		
Concurrence	Lambda-expressions		
Interfaces graphiques	Heritage		

Appeler 5 fois une méthode `f` déjà connue :

```
f(); f(); f(); f(); f();
```

Appeler `f` un nombre de fois inconnu à l'avance :

```
// Facile ! On ajoute un paramètre int :  
public static void repeat(int n) { for (int i = 0; i < n; i++) f(); }
```

Appeler 5 fois une méthode inconnue à l'avance ?

```
// Hm... il faudrait passer une méthode en paramètre ? Tentative :  
public static void repeat5(?? f) { // quel type pour f ?  
for (int i = 0; i < 5; i++) f(); // si f une variable, "f()" -> erreur de syntaxe  
}
```

- `repeat5` = fonction avec paramètre fonction = **fonction d'ordre supérieur** (FOS).
- Pour que cela existe, il faut des fonctions considérées comme des valeurs (passables en paramètre) par le langage : des **fonctions de première classe** (FPC).

Introduction		Fonctions de première classe et fonctions d'ordre supérieur	
Généralités		Le besoin (2)	
Style	Objets et classes	Compléments en 200	
		Aldric Degorre	
Types et polymorphisme	Héritage	Introduction	
collectives	Généricité	Généralités	
Lambda-expressions	Introduction	Style	
Les "strangs"	Élément de type	Objets et classes	
Initialisation des génériques vs. covariance des tableaux	Collectives	Type et polymorphisme	
Villecards	Outils	Heritage	
Concurrence	Lambda-expressions	Généricité	
Interfaces graphiques	Heritage	Introduction	

Une FPC peut être affectée à une variable, être le paramètre d'une FOS, ou bien sa valeur de retour : c'est une valeur comme une autre.

Avec des **valeurs fonction**, il devient possible de manipuler des instructions sans les exécuter/évaluer immédiatement (**évaluation paresseuse**). Elles peuvent ainsi :

- être transformées, composées avant d'être évaluées;
- être évaluées plus tard, une, plusieurs fois ou pas du tout, en fonction de critères programmables ; (condition, répétition, déclenchement par événement ultérieur, ...);
- exécutées dans un autre contexte (p. ex. autre thread<sup>1</sup>).

De telles modalités d'exécution sont programmables en tant que FOS qui se comportent, en gros, comme de nouvelles structures de contrôle<sup>2</sup>.

Introduction		Fonctions de première classe et fonctions d'ordre supérieur	
Généralités		Le besoin (1)	
Style	Objets et classes	Compléments en 200	
		Aldric Degorre	
Types et polymorphisme	Héritage	Introduction	
collectives	Généricité	Généralités	
Lambda-expressions	Introduction	Style	
Les "strangs"	Élément de type	Objets et classes	
Initialisation des génériques vs. covariance des tableaux	Collectives	Type et polymorphisme	
Villecards	Outils	Heritage	
Concurrence	Lambda-expressions	Généricité	
Interfaces graphiques	Heritage	Introduction	

Bloc **if/else** :

```
// types et syntaxe d'appel des FPC toujours fantaisistes dans cet exemple  
public static void ifElse(? condition, ?? ifBlock, ??? elseBlock) {  
    if (condition()) ifBlock();  
    else elseBlock();  
}
```

Impossible d'écrire la signature d'une méthode mimant le bloc **if/else** sans paramètres FPC. Une telle méthode est nécessairement une FOS.

1. Voir chapitre programmation concurrente. La programmation concurrente a probablement été un argument primordial pour l'introduction des lambda-expressions en Java.  
2. À comparer avec `while(...){...}`, `for(...){...}`, `switch(...){...}`, `if (...){...}` ... `else ...,...`

Compléments en P00	Fonctions de première classe et fonctions d'ordre supérieur	Fonctions de première classe et fonctions d'ordre supérieur
Plus d'exemples (3)		Plus d'exemples (2)
Aldric Degorre	Aldric Degorre	Aldric Degorre
Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Introduction Effacement de type Collections Outils Lambda-expressions Les "strangs" Inertie des génératives vs. corréance des tableaux Villards Concurrence Interfaces graphiques	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Introduction Effacement de type Collections Outils Lambda-expressions Les "strangs" Inertie des génératives vs. corréance des tableaux Villards Concurrence Interfaces graphiques	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Introduction Effacement de type Collections Outils Lambda-expressions Les "strangs" Inertie des génératives vs. corréance des tableaux Villards Concurrence Interfaces graphiques
<p><b>Évidemment, plus intéressant d'écrire de nouveaux blocs de contrôle, par exemple :</b></p> <p><b>Bloc retry :</b></p> <pre>// pareil, ne faites pas ça à la maison ! public static void retry(??? instructions, int tries) {     while (tries &gt; 0) {         try { instructions(); return; }         catch (Throwable t) { tries--; }     }     throw new RuntimeException("Failure persisted after all tries."); }</pre>	<p><b>Encore un exemple :<sup>1</sup></b></p> <pre>// toujours en syntaxe fantaisiste --- SURTOUT NE PAS RECOPIER OU MÊME RETENIR ! public static &lt;U&gt; V&gt; List&lt;V&gt; map(List&lt;U&gt; l, ??? f) {     List&lt;V&gt; ret = new ArrayList&lt;V&gt;();     for (U x : l) ret.add(f(x));     return ret; }</pre>	<p><b>Ou encore :</b><sup>2</sup></p> <pre>public static readPacket(Socket s, ??? callback) {     ... }</pre>
		<p><b>callback = FPC pour traiter le prochain paquet reçu = fonction de rappel/callback.</b></p> <p>1. L'API <code>java.util.stream</code> contient plein de méthodes de traitement par lot dans ce genre.  2. Lecture asynchrone, similaire à ce qu'on trouve dans l'API <code>java.nio</code>.</p>
Compléments en P00	Programmation fonctionnelle	Compléments en P00
Aldric Degorre	Aldric Degorre	Aldric Degorre
Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Introduction Effacement de type Collections Outils Lambda-expressions Les "strangs" Inertie des génératives vs. corréance des tableaux Villards Concurrence Interfaces graphiques	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Introduction Effacement de type Collections Outils Lambda-expressions Les "strangs" Inertie des génératives vs. corréance des tableaux Villards Concurrence Interfaces graphiques	Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Introduction Effacement de type Collections Outils Lambda-expressions Les "strangs" Inertie des génératives vs. corréance des tableaux Villards Concurrence Interfaces graphiques
<p><b>Concepts de FPC et de FOS essentiels pour la programmation fonctionnelle (PF) :</b></p> <ul style="list-style-type: none"> <li>• PF = <u>paradigme de programmation</u>, au même titre que la P00.</li> <li>• <b>Idée de base</b> : on conçoit un programme comme une fonction mathématique, elle-même obtenue par composition d'un certain nombre d'autres fonctions.</li> <li>• Or pour pouvoir composer les fonctions, il faut supporter les FOS et donc les FPC.</li> <li>• Langages fonctionnels connus : Lisp (et variantes : Scheme, Emacs Lisp, Clojure...), ML (et variantes : OCaml, F#...), Haskell, Erlang...</li> <li>• Rien n'empêche d'être à la fois objet et fonctionnel (Javascript, Scala, OCaml, Common Lisp ...). Les langages sont souvent <u>multi-paradigme</u> (avec préférence).</li> </ul> <p><b>Java (<math>\geq 8</math>) possède quelques concepts fonctionnels<sup>1</sup>.</b></p>	<p><b>Principalement 3 choses :</b></p> <ul style="list-style-type: none"> <li>• si langage à typage statique, un <u>système de types</u> permettant d'écrire les types des FPC</li> <li>• une <u>yntaxe adaptée</u> pour les expressions décrivant les FPC.</li> <li>• Notamment, il faut des littéraux fonctionnels (appelés aussi <b>lambda-expressions</b><sup>1</sup> ou encore <u>fonctions anonymes</u>).</li> <li>• une <u>représentation en mémoire</u> adaptée pour les FPC (en Java, forcément des objets particuliers)</li> </ul>	<p>1. Mais n'est pas un vrai langage de PF pour autant (on verra plusieurs raisons). Remarque : quasiment tous les langages modernes supportent les FPC, bien qu'ils ne soient pas tous des LPF.</p> <p>Pourquoi « lambda » ? Référence au lambda-calcul d'Alonzo Church : la fonction <math>x \mapsto f(x)</math> s'y écrit <math>\lambda x.f(x)</math>.</p>

## Introduction

## Généralités

## Style

## Objets et classes

## Types et polymorphisme

## Héritage

## Généricité

## Généralité

## Introduction

## Effacement du type

## Collections

## Outils

## Lambda-expressions

## Les "strangs"

## Instanciation des généraux vs covariance des tableaux

## Villeards

## Concurrence

## Interfaces graphiques

**Bilan pour les FPC avant Java 8 :**

- **typage** : au cas par cas, rien de prévu, pas de standard : chaque méthode peut spécifier une interface différente pour la fonction passée en argument.
- à partir de Java 8 : le package `java.util.function` propose une série d'**interfaces fonctionnelles** standard.

Simon, rien n'est changé au système de type de Java (même sa syntaxe).

• **syntaxe** : lourde et peu pratique.

→ à partir de Java 8, on peut écrire des **lambda-expressions**.

• **représentation en mémoire** : instance d'une classe contenant juste une méthode.

Dans tout LOO, une FPC est représentable par un objet ayant la **fonction comme méthode**.

En Java (toute version) on implémente et instance une interface à méthode unique.

Typiquement, pour créer un **thread** à l'aide d'une classe anonyme :

```
new Thread( /* début de l'expression-fonction */ / new Runnable {
    @Override public void run() { /* choses à faire dans l'autre thread */ }
} /* fin de l'expression-fonction */ ).start()
```

Ici, on passe une fonction (décrise par la méthode `run`) au constructeur de `Thread`.

**Inconvénients :**

- **syntaxe lourde et peu lisible**, même avec classes anonymes,
- **obligation de se rappeler et d'écrire des informations sans rapport avec la fonction qu'on décrit** (nom de l'interface : `Runnable` ; et de la méthode implémentée : `run`).

## Introduction

## Généralités

## Style

## Objets et classes

## Types et polymorphisme

## Héritage

## Généricité

## Généralité

## Introduction

## Effacement du type

## Collections

## Outils

## Lambda-expressions

## Les "strangs"

## Instanciation des généraux vs covariance des tableaux

## Villeards

## Concurrence

## Interfaces graphiques

**Bilan pour les FPC avant Java 8 :**

- **typage** : au cas par cas, rien de prévu, pas de standard : chaque méthode peut spécifier une interface différente pour la fonction passée en argument.
- à partir de Java 8 : le package `java.util.function` propose une série d'**interfaces fonctionnelles** standard.

Simon, rien n'est changé au système de type de Java (même sa syntaxe).

• **syntaxe** : lourde et peu pratique.

→ à partir de Java 8, on peut écrire des **lambda-expressions**.

• **représentation en mémoire** : instance d'une classe contenant juste une méthode.

## Introduction

## Généralités

## Style

## Objets et classes

## Types et polymorphisme

## Héritage

## Généricité

## Généralité

## Introduction

## Effacement du type

## Collections

## Outils

## Lambda-expressions

## Les "strangs"

## Instanciation des généraux vs covariance des tableaux

## Villeards

## Concurrence

## Interfaces graphiques

**Bilan pour les FPC avant Java 8 :**

- **typage** : au cas par cas, rien de prévu, pas de standard : chaque méthode peut spécifier une interface différente pour la fonction passée en argument.
- à partir de Java 8 : le package `java.util.function` propose une série d'**interfaces fonctionnelles** standard.

Simon, rien n'est changé au système de type de Java (même sa syntaxe).

• **syntaxe** : lourde et peu pratique.

→ à partir de Java 8, on peut écrire des **lambda-expressions**.

• **représentation en mémoire** : instance d'une classe contenant juste une méthode.

## Bilan

**Bilan pour les FPC avant Java 8 :**

- **typage** : au cas par cas, rien de prévu, pas de standard : chaque méthode peut spécifier une interface différente pour la fonction passée en argument.  
→ à partir de Java 8 : le package `java.util.function` propose une série d'**interfaces fonctionnelles** standard.
- **Simon, rien n'est changé au système de type de Java (même sa syntaxe).**
- **syntaxe** : lourde et peu pratique.  
→ à partir de Java 8, on peut écrire des **lambda-expressions**.
- **représentation en mémoire** : instance d'une classe contenant juste une méthode.  
→ comme c'est une idée raisonnable, ça ne change pas.

<p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Généralité : Introduction Effacement de type Collections Outils</p> <p>Lambda-expressions Les "stream"</p> <p>Intégration des génériques vs covariance des tableaux Wildcards</p> <p>Concurrence Interfaces graphiques</p>	<p>Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Généralité : Introduction Effacement de type Collections Outils</p> <p>Lambda-expressions Les "stream"</p> <p>Intégration des génériques vs covariance des tableaux Wildcards</p> <p>Concurrence Interfaces graphiques</p>
<p><b>Interface fonctionnelle</b> = interface avec 1 seule méthode abstraite<sup>1</sup></p> <ul style="list-style-type: none"> <li>• <b>type SAM</b> (<i>single abstract method</i>) : type défini par une interface fonctionnelle.</li> <li>• En fait, il en existait déjà plein avant Java 8 (ex. : interfaces <code>Comparable</code>, <code>Comparator</code>, <code>Runnable</code>, <code>Callable</code>, <code>ActionListener</code>...).</li> <li>• L'API <code>java.util.function</code> en ajoute quelques dizaines, afin de standardiser les types des FPC. Cf. les 2 pages suivantes.</li> <li>• Les types SAM sont ainsi les types des FPC de Java (celles dont la signature est la même que celle de la méthode du type SAM).</li> <li>• Conséquence : une FPC peut être représentée par plusieurs types SAM (interfaces fonctionnelles de noms différents mais méthode de même signature). Le système de types de Java reste scrupuleusement <u>nominal</u>.</li> </ul> <p>1. ... mais autant de méthodes <code>static</code>, <code>default</code> ou <code>private</code> que l'on souhaite !</p>	

**Catalogue des interfaces fonctionnelles de Java 8**dans le package `java.util.function` (1)dans le package `java.util.function` (2)**java.util.function** contient toute une série d'interfaces fonctionnelles standard.**Interfaces génériques :**

Interface	Type représenté	Méthode unique
<code>BiConsumer&lt;T, U&gt;</code>	$T \times U \rightarrow \{\}\}$	<code>void accept(T, U)</code>
<code>BiFunction&lt;T, U, R&gt;</code>	$T \times U \rightarrow R$	<code>R apply(T, U)</code>
<code>BinaryOperator&lt;T&gt;</code>	$T \times T \rightarrow T$	$T \text{ apply}(T, T)$
<code>BiPredicate&lt;T, U&gt;</code>	$T \times U \rightarrow \{\perp, \top\}$	<code>boolean test(T, U)</code>
<code>Consumer&lt;T&gt;</code>	$T \rightarrow \{\}\}$	<code>void accept(T)</code>
<code>Function&lt;T, R&gt;</code>	$T \rightarrow R$	$R \text{ apply}(T)$
<code>Predicate&lt;T&gt;</code>	$T \rightarrow \{\perp, \top\}$	<code>boolean test(T)</code>
<code>Supplier&lt;T&gt;</code>	$\{\}\rightarrow T$	$T \text{ get}()$
<code>UnaryOperator&lt;T&gt;</code>	$T \rightarrow T$	$T \text{ apply}(T)$

De plus, ce package contient aussi des interfaces pour les fonctions prenant ou retournant des types primitifs `int`, `long`, `double` ou `boolean` (page suivante).

**Interfaces spécialisées :**

Interface	Type représenté	Méthode unique
<code>BooleanSupplier</code>	$\{\}\rightarrow \{\perp, \top\}$	<code>boolean getAsBoolean()</code>
<code>DoubleBinaryOperator</code>	$\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$	<code>double applyAsDouble(double, double)</code>
<code>DoubleConsumer</code>	$\mathbb{R} \rightarrow \{\}\}$	<code>void accept(double)</code>
<code>DoubleFunction&lt;R&gt;</code>	$\mathbb{R} \rightarrow R$	<code>R apply(double)</code>
<code>DoublePredicate</code>	$\mathbb{R} \rightarrow \{\perp, \top\}$	<code>void test(double)</code>
<code>DoubleSupplier</code>	$\{\}\rightarrow \mathbb{R}$	<code>double getAsDouble()</code>
<code>DoubleToIntFunction</code>	$\mathbb{R} \rightarrow \mathbb{Z}$	<code>int applyAsInt(double)</code>
...	...	...

Cf. javadoc de `java.util.function` pour liste complète.

Intérêt des interfaces spécialisées : programmes mieux optimisés<sup>1</sup> qu'avec les types « emballés » (`Int`, `Long`, ...).

1. Moins d'allocations et d'indirections.

	<pre>public static void repeat5(Runnable f) { for (int i = 0; i &lt; 5; i++) f.run(); }</pre>
	<pre>public static void ifElse(BooleanSupplier cond, Runnable ifBlock, Runnable elseBlock) {     if (cond.get() == Boolean.TRUE)         ifBlock.run();     else         elseBlock.run(); }</pre>
	<pre>public static void retry(Runnable instructions, int tries) {     while (tries &gt; 0) {         try {             instructions.run();         } catch (Throwable t) {             tries--;         }     } }</pre>
	<pre>public static &lt;U, V&gt; List&lt;V&gt; map(List&lt;U&gt; l, Function&lt;U, V&gt; f) {     List&lt;V&gt; ret = new ArrayList&lt;V&gt;();     for (U x : l) ret.add(f.apply(x));     return ret; }</pre>
	<p><b>Attention, catalogue incomplet :</b></p> <ul style="list-style-type: none"> <li>L'interface <code>java.lang.Runnable</code> reste le standard pour les « fonctions »<sup>1</sup> de <math>\{()\} \rightarrow \{()\}</math> (<b>void</b> vers <b>void</b>).</li> <li>Pas d'interfaces standard pour les fonctions à plus de 2 paramètres → il faut définir les interfaces soi-même :</li> </ul> <pre>@FunctionalInterface public interface TriConsumer&lt;T, U, V&gt; {     void apply(T t, U u, V v); }</pre> <p>L'annotation facultative <code>@FunctionalInterface</code> demande au compilateur de signaler une erreur si ce qui suit n'est pas une définition d'interface fonctionnelle.</p> <p>1. Ces fonctions sont intéressantes pour leurs effets de bord et non pour la transformation qu'elles représentent. En effet, en mathématiques, <math>\text{card}(\{()\} \rightarrow \{()\}) = 1</math>.</p>

	<p><b>Écrire une fonction anonyme par lambda-abstraction :</b></p> <pre>&lt;paramètres&gt; -&gt; &lt;corps de la fonction&gt;</pre>
	<p><b>Exemples :</b></p> <ul style="list-style-type: none"> <li><math>x \rightarrow x + 2</math></li> <li>(raccourci pour <code>(int x) -&gt; { return x + 2; }</code>)</li> <li><math>(a, b) \rightarrow \{</math></li> <li><math>(x, y) \rightarrow x + y</math></li> <li>(raccourci pour <code>(int x, int y) -&gt; { return x + y; }</code>)</li> <li><math>(a, b) \rightarrow \{</math></li> <li><math>\text{int } q = 0;</math></li> <li><math>\text{while } (a &gt;= b) \{ q++; a -= b; \}</math></li> <li><math>\text{return } q;</math></li> </ul>

	<p><b>Syntaxe des lambda-expressions</b></p> <p>lambda-abstraction : par l'exemple</p>
	<p><b>Écrire une fonction anonyme par lambda-abstraction :</b></p> <pre>&lt;paramètres&gt; -&gt; &lt;corps de la fonction&gt;</pre>
	<p><b>Exemples :</b></p> <ul style="list-style-type: none"> <li><math>x \rightarrow x + 2</math></li> <li>(raccourci pour <code>(int x) -&gt; { return x + 2; }</code>)</li> <li><math>(a, b) \rightarrow \{</math></li> <li><math>(x, y) \rightarrow x + y</math></li> <li>(raccourci pour <code>(int x, int y) -&gt; { return x + y; }</code>)</li> <li><math>(a, b) \rightarrow \{</math></li> <li><math>\text{int } q = 0;</math></li> <li><math>\text{while } (a &gt;= b) \{ q++; a -= b; \}</math></li> <li><math>\text{return } q;</math></li> </ul>

	<p><b>Interfaces fonctionnelles</b></p> <p>Écrire les types des FPC, c'est bien. Mais comment exécuter une fonction ?</p>
	<p>Comme on a déjà pu voir sur les exemples :</p> <ul style="list-style-type: none"> <li>Il n'y a <u>pas de syntaxe réservée</u> pour exécuter une expression fonctionnelle.</li> <li>Il faut donc à chaque fois appeler explicitement la méthode de l'interface fonctionnelle concernée (et donc <u>connaître son nom...</u>).</li> </ul> <p><b>Exemple :</b></p> <pre>Function&lt;Integer, Integer&gt; carre = n -&gt; n * n; System.out.println(carre.apply(5)); // &lt;-- ici c'est apply</pre>
	<p>mais...</p> <pre>Predicate&lt;Integer&gt; estPair = n -&gt; (n % 2 == 0; System.out.println(estPair.test(5)); // &lt;-- là c'est test</pre>
	<p><b>Introduction</b> Généralités Style Objets et classes Types et polymorphisme Héritage Généricité : introduction Effacement de type Collections Outils Lambda-expressions Les « friannes » Instance des génératrices vs covariance des tableaux Wildcards Concurrence Interfaces graphiques</p>

<h2>Syntaxe des lambda-expressions</h2> <p>lambda-abstraction : syntaxe formelle</p>	<p>&lt;paramètres&gt; -&gt; &lt;corps de la fonction&gt;</p> <ul style="list-style-type: none"> <li><b>Syntaxe en détails :</b> <ul style="list-style-type: none"> <li>&lt;paramètres&gt; : liste de paramètres formels (de 0 à plusieurs), de la forme (<b>int</b> <i>x</i>, <b>int</b> <i>y</i>, <b>String</b> <i>s</i>)</li> <li>Mais juste (<i>x</i>, <i>y</i>, <i>s</i>) fonctionne aussi (type des paramètres inféré).</li> </ul> </li> <li>Et parenthèses facultatives quand il y a un seul paramètre.</li> <li>Il est aussi possible (Java ≥ 11) de remplacer les noms de types par <b>var</b>.</li> <li>&lt;corps de la fonction&gt;, au choix : <ul style="list-style-type: none"> <li>une simple expression, p. ex. (<i>x==y</i>)?<i>s</i>: ""</li> <li>une liste d'instructions entre accolades, contenant une instruction <b>return</b> si type de retour non <b>void</b></li> </ul> </li> </ul>	<p>Compléments en 200 Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Généricité : introduction Effacement de type Collections Objets et classes Lambda-expressions Les "strangs" Instance des génériques vs covariance des tableaux Wildcards Concurrence Interfaces graphiques</p>	<h2>Syntaxe des lambda-expressions</h2> <p>Référence de méthode : les cas de figure</p> <p>Supposons la classe suivante définie :</p> <pre>class C {     int val;     C(int val) { this.val = val; }     static int f(int n) { return n; }     int g(int n) { return val + n; } }</pre> <p>La notation « référence de méthode » se décline pour différents cas de figure :</p> <ul style="list-style-type: none"> <li><u>Méthode statique</u> → <i>C::f</i> pour <i>n</i> -&gt; <i>C.f(n)</i></li> <li><u>Méthode d'instance avec récepteur donné</u> → avec <i>x</i> = <b>new</b> <i>C()</i>, on écrit <i>x::g</i> pour <i>n</i> -&gt; <i>x.g(n)</i></li> <li><u>Méthode d'instance sans récepteur donné</u> → <i>C::g</i> pour (<i>x</i>, <i>n</i>) -&gt; <i>x.g(n)</i></li> <li><u>Constructeur</u> → <i>C::new</i> pour <i>n</i> -&gt; <b>new</b> <i>C(n)</i></li> </ul> <p>En cas de surcharge, Java déduit la méthode référencée du type attendu.</p>
<h2>Syntaxe des lambda-expressions</h2> <p>Référence de méthode</p>	<p>Pour créer une lambda-expression contenant juste l'appel d'une méthode existante :</p> <ul style="list-style-type: none"> <li>on peut utiliser la lambda-abstraction : <ul style="list-style-type: none"> <li><i>x -&gt; Math.sqrt(x)</i></li> </ul> </li> <li>mais il existe une notation encore plus compacte : <ul style="list-style-type: none"> <li><b>Math::sqrt</b></li> </ul> </li> </ul> <p>Ceci s'appelle une <b>référence de méthode</b></p>	<p>Compléments en 200 Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Généricité : introduction Effacement de type Collections Objets et classes Lambda-expressions Les "strangs" Instance des génériques vs covariance des tableaux Wildcards Concurrence Interfaces graphiques</p>	<h2>Utiliser les méthodes (FOS) de nos exemples...</h2> <p>... en leur passant des lambda-expressions</p> <pre>// Dire 5 fois "Bonjour !" repeat5(() -&gt; { System.out.println("Bonjour !"); });  // Tirer pile ou face ifElse( () -&gt; Math.random() &gt; 0.5,        () -&gt; { System.out.println("pile"); },        () -&gt; { System.out.println("face"); });  // Essayer d'ouvrir un fichier jusqu'à 3 fois retry(() -&gt; { ouvre("monFichier.txt"); },        3);  // Calculer les racines carrées des nombres d'une liste List&lt;Double&gt; racines = map(maListe, Math::sqrt);</pre>
<h2>Syntaxe des lambda-expressions</h2> <p>Référence de méthode</p>	<p>&lt;paramètres&gt; -&gt; &lt;corps de la fonction&gt;</p> <ul style="list-style-type: none"> <li><b>Syntaxe en détails :</b> <ul style="list-style-type: none"> <li>&lt;paramètres&gt; : liste de paramètres formels (de 0 à plusieurs), de la forme (<b>int</b> <i>x</i>, <b>int</b> <i>y</i>, <b>String</b> <i>s</i>)</li> <li>Mais juste (<i>x</i>, <i>y</i>, <i>s</i>) fonctionne aussi (type des paramètres inféré).</li> </ul> </li> <li>Et parenthèses facultatives quand il y a un seul paramètre.</li> <li>Il est aussi possible (Java ≥ 11) de remplacer les noms de types par <b>var</b>.</li> <li>&lt;corps de la fonction&gt;, au choix : <ul style="list-style-type: none"> <li>une simple expression, p. ex. (<i>x==y</i>)?<i>s</i>: ""</li> <li>une liste d'instructions entre accolades, contenant une instruction <b>return</b> si type de retour non <b>void</b></li> </ul> </li> </ul>	<p>Compléments en 200 Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité Généricité : introduction Effacement de type Collections Objets et classes Lambda-expressions Les "strangs" Instance des génériques vs covariance des tableaux Wildcards Concurrence Interfaces graphiques</p>	<h2>Utiliser les méthodes (FOS) de nos exemples...</h2> <p>... en leur passant des lambda-expressions</p> <pre>// Dire 5 fois "Bonjour !" repeat5(() -&gt; { System.out.println("Bonjour !"); });  // Tirer pile ou face ifElse( () -&gt; Math.random() &gt; 0.5,        () -&gt; { System.out.println("pile"); },        () -&gt; { System.out.println("face"); });  // Essayer d'ouvrir un fichier jusqu'à 3 fois retry(() -&gt; { ouvre("monFichier.txt"); },        3);  // Calculer les racines carrées des nombres d'une liste List&lt;Double&gt; racines = map(maListe, Math::sqrt);</pre>

## Ce qui se cache derrière les lambda-expressions

### Ce qui se cache derrière les lambda-expressions

Concernant le type (1)

#### Nous avons montré :

- d'une part, les types qui sont utilisés pour les FPC en Java (interfaces fonctionnelles)
- d'autre part, la syntaxe permettant d'écrire des FPC (lambda-expressions)

#### Deux questions se posent alors :

- À la compilation, étant donnée une lambda-expression, quel type le compilateur lui donne-t-il?
- À l'exécution, comment est évaluée une lambda-expression ?

Introduction	Compléments en 200
Généralités	Aldric Degorre
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Introduction	
Effacement de type	
Collections	
Objetnulls	
Lambda-expressions	
Les "strangs"	
Instance des généraux vs covariance des tableaux	
Wildcards	
Concurrence	
Interfaces graphiques	

### Ce qui se cache derrière les lambda-expressions

Concernant le type (1)

- Hors contexte, une lambda-expression n'a pas de type (plusieurs types possibles).
- En contexte, sous réserve de compatibilité, son type est le type attendu à son emplacement dans le programme (**inférence de type**).

#### Compatibilité si :

- le type attendu est défini par une interface fonctionnelle (= est un type SAM)
  - et la méthode abstraite de cette interface est redéfinissable par une méthode qui aurait la même signature et le même type de retour que la lambda-expression.

Exemple, on peut écrire `Function<Integer, Double> f = x -> Math.sqrt(x);` car l'interface `Function` est comme suit :

```
public interface Function<T,R> { R apply(T); } // apply a une signature compatible
```

- Ou bien, dans le cas où les types des arguments ne sont pas précisés, s'il existe une façon de les ajouter qui rend la signature compatible.

## Ce qui se cache derrière les lambda-expressions

Concernant le type (3)

**Attention :** n'importe quel type SAM peut être le type d'une lambda-expression. Pas seulement ceux définis dans `java.util.function`.

Partout où une expression de type SAM attendue, on peut utiliser une lambda-expression même si ce type (ou la méthode qui l'attend) date d'avant Java 8.

Ainsi, en Swing, à la place de la classique « invocation magique » :

```
SwingUtilities.invokeLater(
    new Runnable() {
        public void run() { MonIG.build(); }
    });
;
```

on peut écrire : `SwingUtilities.invokeLaterLater(() -> MonIG.build());` ou encore mieux : `SwingUtilities.invokeLaterAfter(MonIG::build);`

## Ce qui se cache derrière les lambda-expressions

Concernant le type (2) – petits pièges

Le fait de préciser le type des paramètres d'une lambda-expression restreint les possibilités d'utilisation.

#### Ces exemples compliment :

```
Function<Integer,Double> f = x -> Math.sqrt(x);
Function<Integer,Double> f = (Integer x) -> Math.sqrt(x);
```

#### Mais ceux-ci ne compliment pas :

```
Function<Integer,Double> f = (Double x) -> Math.sqrt(x);
IntFunction<Double> f = (double x) -> Math.sqrt(x); // pourtant la lambda-expression accepte double (plus large que int).)
```

**Remarquablement, ceci compile (malgré le fait que sqrt ait un paramètre double) :**

```
Function<Integer,Double> f = Math::sqrt;
```

## Ce qui se cache derrière les lambda-expressions

Concernant le type (3)

**Attention :** n'importe quel type SAM peut être le type d'une lambda-expression. Pas seulement ceux définis dans `java.util.function`.

Partout où une expression de type SAM attendue, on peut utiliser une lambda-expression même si ce type (ou la méthode qui l'attend) date d'avant Java 8.

Ainsi, en Swing, à la place de la classique « invocation magique » :

```
SwingUtilities.invokeLater(
    new Runnable() {
        public void run() { MonIG.build(); }
    });
;
```

on peut écrire : `SwingUtilities.invokeLaterLater(() -> MonIG.build());` ou encore mieux : `SwingUtilities.invokeLaterAfter(MonIG::build);`

Introduction	Compléments en 200
Généralités	Aldric Degorre
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Introduction	
Effacement de type	
Collections	
Objetnulls	
Lambda-expressions	
Les "strangs"	
Instance des généraux vs covariance des tableaux	
Wildcards	
Concurrence	
Interfaces graphiques	

## Ce qui se cache derrière les lambda-expressions

Concernant l'évaluation

À l'évaluation, Java construit un objet singleton (instance d'une classe anonyme) qui implémente l'interface fonctionnelle en utilisant la fonction décrite dans la lambda-expression.

Toujours dans le même exemple, `javac` sait alors qu'il doit compiler l'instruction

```
Function<Integer, Double> f = x -> Math.sqrt(x);
```

de la même façon<sup>1</sup> que :

```
Function<Integer, Double> f = new Function<Integer, Double>() {
    @Override public Double apply(Integer x) {
        return Math.sqrt(x);
    }
};
```

1. En fait, pour les lambdas, la JVM construit la classe anonyme à l'exécution seulement. À cet effet, javac a en fait compilé l'expression en écrivant l'instruction **invokedynamic** (introduite dans Java 8 dans ce but). Autrement, les classes, même anonymes, sont créées à la compilation et existent déjà dans le code octet.

- Valeur d'une lambda-expression = instance de classe locale (anonyme).
- Ainsi, une lambda-expression de Java ne peut utiliser que les variables locales effectivement **final**.<sup>1</sup>

- **Comparaison avec OCaml** : en OCaml, cette « limitation » n'est pas perçue car, en effet, les « variables » ne sont pas réaffectables<sup>2</sup> (tout est « **final** »).
- Comme Java, OCaml se contente de recopier les valeurs des variables locales dans la closure de la fonction ( $\simeq$  l'instance de la classe locale).

**1. Rappel** : dans une classe locale, on a accès aux variables locales seulement si elles sont **effectivement final** (c.-à-d. jamais modifiées après leur initialisation). Cette restriction permet d'éviter les incohérences (modifications locales non partagées).

**2. On simule des données locales modifiables en manipulant des « références » mutables :**

```
let ref x = 42 in x := !x + 1; x;
```

Dans l'exemple, `x` n'est pas réaffectable, mais la valeur stockée à l'adresse contenue dans `x` l'est. L'équivalent en Java serait un objet-boîte contenant un unique attribut non **final**. D'ailleurs, rien n'empêche d'utiliser cette technique en Java ; il faut juste l'écrire « à la main ».

Exemple :

- Supposons qu'on veuille définir une fonction récursive, comme en OCaml :

```
let rec fact = function
| 0 -> 1
| n -> n * fact (n - 1);
```

- Sachant qu'il n'existe pas l'équivalent de **let rec** en Java, peut-on définir ?

```
IntUnaryOperator fact = n -> (n==0)?1:(n*fact.applyAsInt(n-1));
```

- Problème : la variable **fact** n'est pas encore déclarée quand on compile la lambda-expression  $\Rightarrow$  la compilation échoue.

Il existe des dizaines de façons de contourner cette limite, mais la seule qui soit élégante consiste à définir d'abord une méthode récursive.

Et correct aussi :

```
class IntRef { int val; IntRef(int val) { this.val = val; } }
final IntRef a = new IntRef(12);
Function<Integer, Integer> f = x -> { return a.val += x; /* modification de a */ };
```

## Fonctions récursives

### Fonctions récursives

Compléments  
en 200

Aldric Degorre

Pour définir cette FPC, il faudrait donc faire en 2 temps :

- ① initialiser `fact` (à `null` par exemple)
- ② écrire la lambda-expression (utilisant `fact`) et l'affecter à `fact`.

Il faudrait donc que la variable `fact` soit réaffectable... donc `fact` ne pourrait pas être locale.

→ Il faudrait que `fact` soit un attribut, mais alors attention à l'encapsulation.

Une possibilité, respectant l'encapsulation, à l'aide d'une classe locale auxiliaire :

```
class FunRef { IntUnaryOperator val; }
final FunRef factAux = new FunRef();
factAux.val = n -> (n==0 ?1 : (n*factAux.val).applyAsInt(n-1));
IntUnaryOperator fact = factAux.val;
```

**Inconvénient** : niveau d'indirection supplémentaire.

**Alternative** : déclarer la factorielle comme méthode privée récursive, puis manipuler une référence vers celle-ci.

```
class Autre2 {
    ...
    private int fact(int n) { return (n==0)?1:(n*fact(n-1)); }
    ...
    IntUnaryOperator fact = Autre2::fact;
    ...
}
```

Et si on veut tout encapsuler correctement, on revient à une classe anonyme classique :

```
IntUnaryOperator fact = new IntUnaryOperator() {
    @Override public int applyAsInt(int n) { return (n==0)?1:(n*applyAsInt(n-1)); }
}
```

Cette dernière technique n'a pas d'inconvénient<sup>1</sup>. C'est donc celle qu'il faut privilégier.

1. si, un : on troque la syntaxe des lambda-expressions contre celle, plus verbuse, des classes anonymes

## Bilan des FPC dans Java 8

### Bilan des FPC dans Java 8

Le bon

Compléments  
en 200

Aldric Degorre

• Pas de notation (flèche) dédiée aux types fonctionnels, juste interfaces classiques.

• Plusieurs interfaces possibles pour une même fonction.

• Pas de syntaxe réservée, unique, pour exécuter une FPC.

À la place : appel de la méthode de l'interface, dont le nom peut varier.

• Clôture contenant variables effectivement finales seulement.

Implication : nécessité de « contourner » pour capturer un état mutable.<sup>1</sup>  
(⇒ Impossible de définir simplement une lambda-expression réursive<sup>2</sup>.)

• Malgré les apports de Java 8 à 15, le JDK contient peu d'APIs dans le style fonctionnel (On peut néanmoins citer `Stream` et `CompletableFuture`).

Java n'est toujours pas un langage de PF, mais juste un LOO avec support limité des FPC.

1. Cela dit, on évite d'utiliser un état mutable en PF.

2. De plus Java n'optimise pas la récursivité terminale. Ainsi, l'appel d'une méthode récursive sur des données de taille modérément grande risque facilement de provoquer un `StackOverflowError`.  
Ainsi rien n'est fait pour encourager la programmation récursive.

## Bilan des FPC dans Java 8

Le mauvais

Compléments  
en 200

Aldric Degorre

• Java supporte les FPC et les FOS.

Or les FPC sont amenées à jouer un rôle de plus en plus important, notamment pour la programmation concurrente<sup>1</sup>, qui devient de plus en plus incontournable<sup>2</sup>.

• Les API `Stream` et `CompletableFuture` sont d'excellents exemples d'API concurrentes, introduites dans Java 8 et utilisant les FOS.

• D'anciennes API se retrouvent immédiatement utilisables avec les lambdas-expressions car utilisant déjà des interfaces fonctionnelles (e.g. JavaFX).

→ nouvelle concision, « gratuite ».

Malgré ses défauts, le support des FPC et FOS dans Java est un apport indéniable.

1. Quel est le rapport entre FPC/FOS et programmation concurrente ? Plusieurs réponses :

• la programmation concurrente incite à utiliser des structures immuables pour garantir la correction du programme.

• Or le style fonctionnel est naturel pour travailler avec les structures immuables.

• en programmation concurrente, on demande souvent l'exécution asynchrone d'un morceau de code. Pour ce faire, ce dernier doit être passé en argument d'une fonction (FOS) sous la forme d'une FPC.

2. En particulier à cause de la multiplication du nombre de cœur dans les microprocesseurs.

## Opérations d'agrégation

Définition et quelques exemples

## Opérations d'agrégation

Quelques exemples (1)

Compléments en POO

Aldric Degorre

**Opération d'agrégation** : traitement d'une séquence de données de même type qui produit un résultat synthétique<sup>1</sup> dépendant de toutes ces données.

**Exemples :**

- calcul de la taille d'une collection
- concaténation des chaînes d'une liste de chaînes
  - transformation d'une liste de chaînes en la liste de ses longueurs  
(ex : "bonjour", "le", monde → 7, 2, 5)
  - recherche d'un élément satisfaisant un certain critère

Tous ces calculs pourraient s'écrire à l'aide de boucles **for** très similaires...

1. synthèse = résumé

## Opérations d'agrégation

Quelques exemples (2)

## Opérations d'agrégation

Généralisation

Compléments en POO

Aldric Degorre

Transformation d'une liste de chaînes en la liste de ses longueurs :

```
public static List<Integer> lengths(List<String> dataSource) {  
    List<Integer> acc = new LinkedList<>();  
    for (String e : dataSource) acc.add(e.length());  
    return acc;  
}
```

Recherche d'un élément satisfaisant un certain critère<sup>1</sup> :

```
public static E find(List<E> dataSource, Predicate<E> criterion) {  
    E acc = null;  
    for (E e : dataSource) acc = (criterion.test(e)?e:null);  
    return acc;  
}
```

Voyez-vous le motif commun?

1. Remarque : on peut optimiser cette boucle, mais cette présentation illustre mieux le propos.

## Opérations d'agrégation

Quelques exemples (1)

Compléments en POO

Aldric Degorre

**Calcul de la taille d'une collection :**

```
public static int size(Collection<?> dataSource) {  
    int acc = 0;  
    for (Object e : dataSource) acc++;  
    return acc;  
}
```

**Concaténation des chaînes d'une liste de chaînes :**

```
public static String concat(List<String> dataSource) {  
    String acc = "";  
    for (String e : dataSource) acc += e.toString();  
    return acc;  
}
```

**Calcul de la taille d'une collection :**

```
public static int size(Collection<?> dataSource) {  
    int acc = 0;  
    for (Object e : dataSource) acc++;  
    return acc;  
}
```

**Concaténation des chaînes d'une liste de chaînes :**

```
public static String concat(List<String> dataSource) {  
    String acc = "";  
    for (String e : dataSource) acc += e.toString();  
    return acc;  
}
```

## Opérations d'agrégation

Généralisation

Compléments en POO

Aldric Degorre

On garde ce qui est commun dans une méthode prenant en argument ce qui est différent :

```
public static <E, R> R fold(Iterable<E> dataSource, R zero, ??? op) {  
    R acc = zero;  
    for (E e : dataSource) acc = op(acc, e); // comment on écrit ça déjà ?  
    return acc;  
}
```

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Interaction

Éffacement de type

Collectives

Opérations

Lambdageprogrammes

Les "streams"

Immutabilité des données et covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

...

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Interaction

Éffacement de type

Collectives

Opérations

Lambdageprogrammes

Les "streams"

Immutabilité des données et covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

...

Introduction	Généralités
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Généricité : Introduction Effacement de type Collections Outils	
Lambdpressions	
Les "streams"	
Instance des génératrices vs. concurrenties des tableaux Vidéos	
Concurrence	
Interfaces graphiques	

On garde ce qui est commun dans une méthode prenant en argument ce qui est différent :

```
public static <E, R> R fold(Iterable<E> dataSource, R zero, ?? op) {
    R acc = zero;
    for (E e : dataSource) acc = op(acc, e); // comment on écrit ça déjà ?
    return acc;
}
```

... et on se rappelle le cours sur les fonctions de première classe (FPC) et les fonctions d'ordre supérieur (FOS) :

```
public static <E, R> R fold(Iterable<E> dataSource, R zero, BiFunction<R, E, R> op) {
    R acc = zero;
    for (E e : dataSource) acc = op.apply(acc, e);
    return acc;
}
```

Introduction	Généralités
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Généricité : Introduction Effacement de type Collections Outils	
Lambdpressions	
Les "streams"	
Instance des génératrices vs. concurrenties des tableaux Vidéos	
Concurrence	
Interfaces graphiques	

On peut alors écrire :

```
public static int size(Collection<?> dataSource) {
    return fold(dataSource, 0, (acc, e) -> acc + 1);
}

public static String concat(List<String> dataSource) {
    return fold(dataSource, "", (acc, e) -> acc + e.toString());
}

public static List<Integer> lengths(List<String> dataSource) {
    return fold(dataSource, new LinkedList<?>(),
              (acc, e) -> { acc.add(e.length()); return acc; });
}

// "Bof" : on modifie l'argument de op dans op (incorrect si fold est concurrent).
// On doit pouvoir faire mieux (à méditer en TP... ) !
public static <E> E find(List<E> dataSource, Predicate<E> criterion) {
    return fold(dataSource, null, (acc, e) -> (criterion.test(e) ? e : null));
}
```

Introduction	Généralités
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Généricité : Introduction Effacement de type Collections Outils	
Lambdpressions	
Les "streams"	
Instance des génératrices vs. concurrenties des tableaux Vidéos	
Concurrence	
Interfaces graphiques	

## L'API Streams

Ce dont il s'agit.

**Streams** : API introduite dans Java 8 pour effectuer des opérations d'agrégation.

- API dans le style fonctionnel, avec fonctions d'ordre supérieur;
- distincte de l'API des collections (nouvelle interface `Stream`, au lieu de méthodes ajoutées à `Collection`<sup>1</sup>);
- optimisée pour les grands volumes de données : **évaluation paresseuse** (calculs effectués seulement au dernier moment, seulement lorsqu'ils sont nécessaires);
- qui sait utiliser les CPUs multi cœur pour accélérer ses calculs (implémentation parallèle multi-threadée).

## Opérations d'agrégation

Généralisation

Introduction	Généralités
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Généricité : Introduction Effacement de type Collections Outils	
Lambdpressions	
Les "streams"	
Instance des génératrices vs. concurrenties des tableaux Vidéos	
Concurrence	
Interfaces graphiques	

## L'API Streams

Ce dont il s'agit.

Pour écrire des traitements similaires à ces exemples, on aimeraient une API fournissant des FOS analogues à `fold` pour les principaux schémas d'itération<sup>1</sup>. C'est justement ce que fait l'API `stream`.<sup>2</sup>

**Avertissement** : ce chapitre traite du package `java.util.stream` introduit dans Java 8. Ces streams n'ont aucun rapport avec les classes `InputStream` et `OutputStream` de `java.io`.

1. Similaires aux fonctions de manipulation de liste en OCaml.
2. Mais pas seulement...

## Opérations d'agrégation

Avec les streams

Exemples

Avec l'API `stream`, une telle opération se décompose sous forme d'un **pipeline** d'étapes successives, selon le schéma suivant :

- ➊ sélection d'une **source**<sup>1</sup>. Depuis cette source on obtient un **stream**.
- ➋ un certain nombre (0 ou plus) d'**opérations intermédiaires**. Ces opérations transforment un `stream` en un autre `stream`.
- ➌ une **opération terminale** qui transforme le `stream` en un résultat final (qui n'est plus un `stream`).

Les calculs sont effectués à l'appel de l'opération terminale seulement. Et seuls les calculs nécessaires le sont.

1. Souvent une collection, mais peut aussi être un tableau, une fonction productrice d'éléments, un canal d'entrées/sorties

### Quelques streams :

- Pour toute collection `coll`, le `stream` associé est `coll.stream()`.
- `Stream.of(4, 39, 2, 12, 32)` représente la séquence 4, 39, 2, 12, 32.
- `Stream.of(4, 39, 2, 12, 32).map(x -> 2 * x)` représente la séquence 8, 78, 4, 24, 64.

Inversement, on peut ensuite obtenir une collection depuis un `stream` :

```
Stream.of(4, 39, 2, 12, 32).map(x -> 2 * x).collect(Collectors.toList)
```

→ on obtient un `List<Integer>` (`collect`, opération terminale, force le calcul de la séquence).

Introduction	Généralités	Style	Objets et classes	Types et polymorphisme	Héritage	Généricité	Généricité	Introduction	Effacement de type	Collectors	Options	Lambdas-expressions	Les "streams"	Instance des généricités vs. covariance des tableaux	Villejards	Concurrence	Interfaces graphiques
Compléments en POO	Aldric Degorre																

Compléments en POO	Aldric Degorre	Caractéristiques principales
<b>Les objets stream</b>		

Qu'est-ce qu'un `stream`? → **2 points de vue** :

- ➊ la représentation implicite d'une séquence d'éléments finie ou infinie
- ➋ la description d'une suite d'opérations permettant d'obtenir cette séquence.

### Remarques importantes :

- Un objet `stream` n'est pas une collection : il ne contient qu'une référence vers une source d'éléments (parfois une collection, souvent un autre `stream`) et la description d'une opération à effectuer.
- Un objet `stream` n'est pas le résultat d'un calcul, mais la description d'un calcul à effectuer<sup>1</sup>.

1. Pour les fans de programmation fonctionnelle : le type `Stream<T>` muni des opérations `of` et `flatMapMap` est une monade.

Compléments en POO	Aldric Degorre	Stream versus Iterator
<b>Les objets stream</b>		

Les streams et les `iterators` ont beaucoup en commun :

- intermédiaires techniques pour parcourir les collections
- contiennent juste l'information pour faire cela ; pas les éléments eux-mêmes ;
- usage unique (après le premier parcours de la source, l'objet ne peut plus servir)

Et une grosse différence :

- streams : opérations agissant sur l'ensemble des éléments (itération implicite). Ce sont les méthodes fournies dans le JDK qui gèrent l'itération (et proposent notamment une implémentation en parallèle sur plusieurs `threads`<sup>1</sup>).
- iterators : 1 opération (`next`) = lire l'élément suivant (→ itération explicite avec **for ou while**)

1. Le `stream` construit sur une collection utilise en fait le `splitter` de celle-ci : sorte d'itérateur évolué capable, en plus d'itérer séquentiellement, de couper une collection en morceaux ("split") pour partager le travail entre plusieurs `threads`.

## Quelques exemples de traitements réalisables en utilisant les streams

### Streams et parallélisme

Problème à résoudre

Compléments en POO

Aldric Degorre

15 nombres entiers aléatoires positifs inférieurs à 100 en ordre croissant :

```
Stream<generate(Math::random) // on obtient un Stream<Double>
    .limit(15) //
    .map(x -> (int)(100 * x)) //
    .sorted() //
    .collect(Collectors.toList()); //
```

Nombre d'usagers d'une bibliothèque ayant emprunté un livre d'Alexandre Dumas :

```
bibli.getLivres() //
    .stream() //
    .filter(livre -> livre.getAuteur().equals("Dumas, Alexandre")) //
    .flatMap(livre -> livre.getEmprunteurs(), stream() //
    .distinct() //
    .count()); //
```

## Streams et parallélisme

Problème à résoudre... une solution ?

### Streams et parallélisme

Streams séquentiels et streams parallèles

Compléments en POO

Aldric Degorre

**Les streams, semblent une réponse naturelle à ce problème.** En effet :

- leurs opérations ne modifient pas le contenu de leur source;
- l'objet de type Stream est lui-même à usage unique.

→ protection naturelle maximale contre les accès en compétition.  
→ Ce serait bien que les opérations d'agrégation d'un stream puissent être réparties sur plusieurs threads (lors de l'appel à l'opération terminale)...

... et bien justement :

Java permet de lancer les opérations d'agrégation en parallèle<sup>1</sup>, sans presque rien changer à l'invocation du même traitement en séquentiel :

- Il suffit de créer le stream avec `maCollection.parallelStream()` à la place de `maCollection.stream()`.
- Alternative : à partir d'un stream séquentiel, on peut obtenir un stream parallèle avec la méthode `parallel`, et vice-versa avec la méthode `sequential`.

Un stream est soit (entièrement) parallèle, soit (entièrement) séquentiel. L'opération terminale prend seulement en compte le dernier appel à sequential ou parallel<sup>2</sup>.

1. En utilisant (de façon cachée) `ForkJoinPool.ForkJoinTask`. Sauf mention contraire, le thread pool par défaut `ForkJoinPool.commonPool()` est utilisé.

2. Rappel : l'effectuation des calculs étant seulement déclenchée par l'opération terminale, il est logique que ses modalités concrètes d'exécution ne soient prises en compte qu'à ce moment.

- Les calculs d'un *pipeline* parallèle sont répartis sur plusieurs *threads*.
- Leur ordre d'exécution peut être sans rapport avec celui des éléments de la source.
- L'opération terminale retourne après que tous les calculs parallèles sont terminés (synchronisation!).
- Certaines opérations garantissent que les éléments sont traités dans l'ordre, s'ils en avaient un (p. ex. : `forEachOrdered`), d'autres non (`forEach`).
- Imposer l'ordre demande plus de synchronisation, impliquant moins de parallélisme.
- Certaines opérations sont optimisées pour le traitement parallèle (p. ex. `.collect(Collectors.toConcurrentMap(...))` plus efficace que `.collect(Collectors.toMap(...))`).

Introduction	Compléments en 200
Généralités	Aldric Degorre
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Introduction	
Effacement de type	
Collectives	
Options	
Lambda-expressions	
Les "streams"	
Intégration des génératrices vs. covariance des tableaux	
Wildcards	
Concurrence	
Interfaces graphiques	

Introduction	Compléments en 200
Généralités	Aldric Degorre
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Introduction	
Effacement de type	
Collectives	
Options	
Lambda-expressions	
Les "streams"	
Intégration des génératrices vs. covariance des tableaux	
Wildcards	
Concurrence	
Interfaces graphiques	

Introduction	Compléments en 200
Généralités	Aldric Degorre
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Introduction	
Effacement de type	
Collectives	
Options	
Lambda-expressions	
Les "streams"	
Intégration des génératrices vs. covariance des tableaux	
Wildcards	
Concurrence	
Interfaces graphiques	

Introduction	Compléments en 200
Généralités	Aldric Degorre
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Introduction	
Effacement de type	
Collectives	
Options	
Lambda-expressions	
Les "streams"	
Intégration des génératrices vs. covariance des tableaux	
Wildcards	
Concurrence	
Interfaces graphiques	

Introduction	Compléments en 200
Généralités	Aldric Degorre
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Introduction	
Effacement de type	
Collectives	
Options	
Lambda-expressions	
Les "streams"	
Intégration des génératrices vs. covariance des tableaux	
Wildcards	
Concurrence	
Interfaces graphiques	

Introduction	Compléments en 200
Généralités	Aldric Degorre
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Introduction	
Effacement de type	
Collectives	
Options	
Lambda-expressions	
Les "streams"	
Intégration des génératrices vs. covariance des tableaux	
Wildcards	
Concurrence	
Interfaces graphiques	

## Appendice : les méthodes de l'API stream

Les transparents qui suivent :

- sont un résumé des méthodes proposées dans l'API stream.
  - ne sont pas détaillés en cours magistral
  - doivent servir de référence pour les TPs et pour la relecture approfondie du cours.
- (Il existe aussi `DoubleStream`, `IntStream` et `LongStream`.)
- Cette interface contient un grand nombre de méthodes. 3 catégories :
- des méthodes statiques<sup>1</sup> servant à créer des streams depuis des sources diverses.
  - des méthodes d'instance transformant des streams en streams (pour les opérations intermédiaires)
  - des méthodes d'instance transformant des streams en autre chose (pour les opérations terminales).

1. Rappel : oui, c'est possible depuis Java 8.

## Fabriquer un stream depuis une source

Compléments en POO	Aldric Degorre	<p>Depuis une <u>collection</u> : méthode <code>Stream&lt;T&gt; stream()</code> de <code>Collection&lt;T&gt;</code>.</p> <p>À l'aide d'une des <u>méthodes statiques</u> de <code>Stream</code> :</p> <ul style="list-style-type: none"><li>&lt;<code>T</code>&gt; <code>Stream&lt;T&gt; empty()</code> : retourne un stream vide</li><li>&lt;<code>T</code>&gt; <code>Stream&lt;T&gt; generate(Supplier&lt;T&gt; s)</code> : retourne la séquence des éléments générés par <code>s.get()</code> (Rappel : <code>Supplier&lt;T&gt;</code> = fonction de <code>{() -&gt; T}</code>).</li><li>&lt;<code>T</code>&gt; <code>Stream&lt;T&gt; iterate(T seed, UnaryOperator&lt;T&gt; f)</code> : retourne la séquence des éléments <code>seed, f.apply(seed), f.apply(f.apply(seed)) ...</code></li><li>&lt;<code>T</code>&gt; <code>Stream&lt;T&gt; of(T... values)</code> : retourne le stream constitué de la liste des éléments passés en argument (méthode d'arité variable).</li></ul> <p>En utilisant un <u>builder</u><sup>1</sup> (<code>Stream.Builder</code>) :</p> <ul style="list-style-type: none"><li>Un <code>Stream.Builder</code> est un objet mutable servant à construire un stream.</li><li>On instancie un <code>builder</code> vide avec l'appel statique <code>b = Stream.builder()</code></li><li>On ajoute des éléments avec les appels <code>b.add(T e)</code> ou <code>b.accept(T e)</code>.</li><li>On finalise en créant le stream contenant ces éléments : appel <code>s = b.build()</code></li></ul> <p>1. On parle du patron de conception <code>builder</code> (ou "monteur") ici appliquée aux streams. Ainsi, par exemple, il existe une classe <code>StringBuilder</code> jouant le même rôle pour les <code>String</code>. Voir le TP sur le patron <code>builder</code>.</p>
--------------------	----------------	--

## Transformer un stream : les opérations intermédiaires (1)

Compléments en POO	Aldric Degorre	<p>Pour <code>this</code> instance de <code>Stream&lt;T&gt;</code> :</p> <ul style="list-style-type: none"><li>&lt;<code>T</code>&gt; <code>Stream&lt;T&gt; distinct()</code> retourne un stream qui parcourt les éléments de <code>this</code> sans les doublons.</li><li>&lt;<code>T</code>&gt; <code>Stream&lt;T&gt; filter(Predicate&lt;? super T&gt; p)</code> retourne le stream parcourant les éléments <code>x</code> de <code>this</code> qui satisfont <code>p.test(x)</code></li><li>&lt;<code>R</code>&gt; <code>Stream&lt;R&gt; flatMap(Function&lt;? super T, ? extends Stream&lt;? super R&gt; mapper)</code> retourne la concaténation des streams <code>mapper.apply(x)</code> pour tout <code>x</code> dans <code>this</code>.</li><li>&lt;<code>T</code>&gt; <code>Stream&lt;T&gt; limit(long n)</code> tronque le stream après <code>n</code> éléments.</li><li>&lt;<code>U</code>&gt; <code>Stream&lt;U&gt; map(Function&lt;T, U&gt; f)</code> retourne le stream des éléments <code>f.apply(x)</code> pour tout <code>x</code> élément de <code>this</code>.</li></ul>
--------------------	----------------	--

## Transformer un stream : les opérations intermédiaires (2)

Compléments en POO	Aldric Degorre	<p>Transformer un stream avec les mêmes éléments que <code>this</code>. À l'étape terminale, pour chaque élément <code>x</code> parcouru, <code>c.consume(x)</code> sera exécuté<sup>1</sup>.</p> <p>&lt;<code>T</code>&gt; <code>Stream&lt;T&gt; peek(Consumer&lt;? super T&gt; c)</code> retourne le suffixe de la séquence en sautant les <code>n</code> premiers éléments.</p> <p>&lt;<code>T</code>&gt; <code>Stream&lt;T&gt; skip(long n)</code> retourne la séquence, triée dans l'ordre naturel.</p> <p>&lt;<code>T</code>&gt; <code>Stream&lt;T&gt; sorted()</code> idem mais en suivant l'ordre fourni.</p>
--------------------	----------------	---

## Calculer et extraire un résultat : les opérations terminales

Compléments en POO	Aldric Degorre	<p>Opérations spécialisées (1)</p> <p>Introduction</p> <p>Généralités</p> <p>Style</p> <p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Intégration</p> <p>Élément de type</p> <p>Collections</p> <p>Options</p> <p>Lambda-expressions</p> <p>Les "streams"</p> <p>Instance des génériques vs covariance des tableaux</p> <p>Wildcards</p> <p>Concurrence</p> <p>Interfaces graphiques</p>
--------------------	----------------	---

## Calculer et extraire un résultat : les opérations terminales

Compléments en POO	Aldric Degorre	<p>boolean <code>allMatch(Predicate&lt;? super T&gt; p)</code> retourne vrai si et seulement si <code>p</code> est vrai pour tous les éléments du stream.</p> <p>boolean <code>anyMatch(Predicate&lt;? super T&gt; p)</code> retourne vrai si et seulement si <code>p</code> est vrai pour au moins un élément du stream.</p> <p>long <code>count()</code> retourne le nombre d'éléments dans le stream.</p> <p>Optional&lt;T&gt; <code>findFirst()</code> : retourne un élément (quelconque) du stream ou rien si stream vide (voir interface <code>Optional&lt;T&gt;</code>).</p> <p>Optional&lt;T&gt; <code>findFirst()</code> : pareil, mais premier élément.</p>
--------------------	----------------	---

## Calculer et extraire un résultat : les opérations terminales

Compléments en POO	Aldric Degorre	<p>1. Remarque : <code>c</code> ne sert que pour ses effets de bord. <code>peek</code> peut notamment être utile pour le débogage.</p>
--------------------	----------------	--

## Calculer et extraire un résultat : les opérations terminales

Compléments en 200  
Aldric Degorre  
Opérations spécialisées (2)

Introduction	Compléments en 200 Aldric Degorre
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Généricité : introduction	
Effacement de type	
Collections	
Options	
Lambdas-expressions	
Les "streams"	
Instance des génériques vs covariance des tableaux	
Wildcards	
Concurrence	
Interfaces graphiques	

- **void forEach(Consumer<? super T> action) :** applique `action` à chaque élément.
- **void forEachOrdered(Consumer<? super T> action) :** pareil en garantissant de traiter les éléments dans l'ordre de la source si elle en avait un.
- **Optional<T> max(Comparator<? super T> comp) :** retourne le maximum.
- **Optional<T> min(Comparator<? super T> comp) :** retourne le minimum.
- **Object[] toArray() :** retourne un tableau contenant les éléments du `stream`.
- **<A> A[] toArray(IntFunction<A[]> generator) :** retourne un tableau contenant les éléments du `stream`. La fonction `generator` sert à instancier un tableau de la taille donnée par son paramètre.

1. Opération appelée `fold` dans d'autres langages. Les définitions varient...

Calculer et extraire un résultat : les opérations terminales  
Opérations "couteau suisse" :

Introduction	Compléments en 200 Aldric Degorre
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Généricité : introduction	
Effacement de type	
Collections	
Options	
Lambdas-expressions	
Les "streams"	
Instance des génériques vs covariance des tableaux	
Wildcards	
Concurrence	
Interfaces graphiques	

- **Optional<T> reduce(BinaryOperator<T> op) :** effectue la réduction du `stream` par l'opération d'accumulation associative `op`.
- **T reduce(T zero, BinaryOperator<T> op) :** idem, avec le zéro fourni.
- **<U> U reduce(U z, BiFunction<U, ? super T, U> acc, BinaryOperator<U> comb) :** idem avec accumulation vers autre type.<sup>1</sup>
- **<R> R collect(Supplier<R> z, BiConsumer<R, ? super T> acc, BiConsumer<R, R> comb) :** comme `reduce` avec accumulation dans objet mutable.
- **<R,A> R collect(Collector<? super T,A,R> collector) :** on parle juste après.

1. Opération appelée `fold` dans d'autres langages. Les définitions varient...

## La classe Collectors

Introduction	Compléments en 200 Aldric Degorre
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Généricité : introduction	
Effacement de type	
Collections	
Options	
Lambdas-expressions	
Les "streams"	
Instance des génériques vs covariance des tableaux	
Wildcards	
Concurrence	
Interfaces graphiques	

## L'interface Collector

Introduction	Compléments en 200 Aldric Degorre
Généralités	
Style	
Objets et classes	
Types et polymorphisme	
Héritage	
Généricité	
Généricité : introduction	
Effacement de type	
Collections	
Options	
Lambdas-expressions	
Les "streams"	
Instance des génériques vs covariance des tableaux	
Wildcards	
Concurrence	
Interfaces graphiques	

Un `Collector` est un objet servant à "réduire" un `stream` en un résultat concret (en effectuant le calcul). Pour ce faire,

- il initialise un accumulateur du type désiré (p. ex : liste vide),
- puis transforme et aggrège les éléments issus du calcul du `stream` dans l'accumulateur (ex : ajout à la liste)
- enfin il "finalise" l'accumulateur avant retour (p. ex : suppression des doublons).

Trois techniques pour fabriquer un tel objet :

- **(cas courants) utiliser une des fabriques statiques de la classe Collectors**
- utiliser la fabrique statique `Collector.of()` ("constructeur" généraliste)
- programmer à la main une classe qui implémente l'interface `Collector`

Cette classe, non instanciable, est une bibliothèque de fabriques statiques pour obtenir simplement les `collectors` les plus courants. Quelques exemples :

- `Collectors.toList()`, `Collectors.toSet()`, `Collectors.counting()`,
- `Collectors.groupingBy(..., Collectors.reducing(...))`,
- `Collectors.toConcurrentMap(...)`...

→ on retrouve des opérations équivalentes<sup>1</sup> à la plupart des réductions de l'interface `Stream`.

Ainsi, autre façon d'avoir la taille d'un `stream` :  
`monStream.collect(Collectors.counting())2`.

- 1. mais ici : implémentation "mutable", utilisant un attribut accumulateur, alors que dans `Stream`, les réductions utilisent de fonctions "pures"
- 2. ... mais le plus simple reste `monStream.count()` !

## Construire un collector via l'interface Collector

### Types génériques et sous-typage

Compléments en 200

Aldric Degorre

Au cas où la bibliothèque `Collectors` ne contient pas ce qu'on cherche, on peut créer un `Collector` autrement :

- créer et instancier une classe implémentant `Collector`.
- Méthodes à implémenter : `accumulator()`, `characteristics()`, `combiner()`, `finished()` et `supplier()`.
- sinon, créer directement l'objet grâce à la méthode statique `Collector.of()` :

```
c2 = Collector<Integer, List<Integer>, Integer> c2 = Collector.of(  
    Arrays.asList::new, List::add,  
    (l1, l2) -> {l1.addAll(l2); return l1}, List::size  
);
```

(façon... un peu alambiquée de calculer la taille d'un stream...)

Utiliser la méthode `of()` est plus "légèr" syntaxiquement, mais ne permet pas d'ajouter des champs ou des méthodes à l'objet fabriqué.

### Types génériques et sous-typage

Invariance des génériques

**Le problème :** examinons l'exemple suivant (qui ne compile pas).

```
public static void main(String[] args) {  
    List<VoitureSousPermis> listVoituresSP = new ArrayList<>() ;  
    // l1: ceci est en fait interdit... mais supposons que ça passe...  
    List<Voiture> listVoiture = listVoituresSP;  
    // l2: instruction bien typée (pour le compilateur), mais...  
    listVoiture.add(new Voiture());  
    // l3: ... logiquement ça afficherait "Voiture" (contradictoire)  
    System.out.println(listVoitureSP.get(0).getClass());  
}
```

S'il compilait, en l'exécutant, à la fin, `listVoitureSP = listVoiture` contiendrait des `Voiture` → **contredit la déclaration de listVoitureSP !**

Ainsi, Java interdit l1 : deux spécialisations différentes du même type générique sont incompatibles. Ont dit que les génériques de Java sont **invariants**.

### Tableaux – génériques ou pas ? (1)

Compléments en 200

Aldric Degorre

### Types génériques et sous-typage

Invariance des génériques

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Introduction

Effacement de type

collectives

Options

Lambda-expressions

Les "streams"

Invariance des génériques vs covariance des tableaux

Wildcards

Concurrency

Interfaces graphiques

### Tableaux – génériques ou pas ? (1)

Compléments en 200

Aldric Degorre

### Tableaux – génériques ou pas ? (1)

Invariance des génériques

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Introduction

Effacement de type

collectives

Options

Lambda-expressions

Les "streams"

Invariance des génériques vs covariance des tableaux

Wildcards

Concurrency

Interfaces graphiques

### Remarque : l'analogue à l'exemple précédent utilisant `Voiture[]` au lieu de `List<Voiture>` compile sans avertissement :

```
public static void main(String[] args) {  
    VoitureSousPermis[] listVoituresSP = new VoitureSousPermis[100];  
    // l1: ceci est autorisé !  
    Voiture[] listVoiture = listVoituresSP;  
    // l2: instruction bien typée (pour le compilateur), mais... ArrayStoreException !  
    listVoiture[0] = new Voiture();  
    // l3: on ne va pas jusque là  
    System.out.println(listVoitureSP[0].getClass());  
}
```

→ Les tableaux sont **covariants** (l1 autorisé) : `[A <: B] ⇒ [A[] <: B[]]`.  
Mais on crashe plus loin, lors de l'exécution de l2 (bien qu'il n'y ait pas eu de warning!).

### Note : cependant le compilateur détecte la conversion « louche » et signale un avertissement (warning) « unchecked conversion » pour la ligne l1'.

**Moralité :** en programmation générique, le paramètre de type fournit une garantie stricte et le compilateur refuse de compiler au moindre doute. Si on passe autre (`cast`), il nous avertit (à raison car `ClassCastException` peut se produire à l'exécution).

### Types génériques et sous-typage

Compléments en 200

Aldric Degorre

### Types génériques et sous-typage

Invariance des génériques

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Introduction

Effacement de type

collectives

Options

Lambda-expressions

Les "streams"

Invariance des génériques vs covariance des tableaux

Wildcards

Concurrency

Interfaces graphiques

### Types génériques et sous-typage

Invariance des génériques

Aldric Degorre

### Types génériques et sous-typage

Invariance des génériques

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Introduction

Effacement de type

collectives

Options

Lambda-expressions

Les "streams"

Invariance des génériques vs covariance des tableaux

Wildcards

Concurrency

Interfaces graphiques

## Tableaux – génériques ou pas ? (2)

### Génériques et tableaux

Différence de philosophie (1)

Compléments en P00  
Aldric Degorre

Introduction	Généralités	Compléments en P00
Généralités	Style	Aldric Degorre
Style	Objets et classes	
Objets et classes	Types et polymorphisme	
Types et polymorphisme	Héritage	
Héritage	Généricité	
Généricité	Introduction	
Introduction	Effacement de type	
Effacement de type	Collections	
Collections	Options	
Options	Lambda-expressions	
Lambda-expressions	Les "strangs"	
Les "strangs"	Instance des génériques vs. covariance des tableaux	
Instance des génériques vs. covariance des tableaux	Villefranche	
Villefranche	Concurrence	
Concurrence	Interfaces graphiques	
Interfaces graphiques	Concurrence	
Concurrence	Interfaces graphiques	

- covariance à la place d'invariance :  $\Rightarrow$  vérifications moins strictes à la compilation, rendant possibles des problèmes à l'exécution<sup>1</sup>
  - pour détecter les problèmes au plus tôt : à l'instanciation, un tableau enregistre le nom du type déclaré pour ses éléments (pas d'effacement de type)
  - cela permet à la JVM de déclencher **ArrayStoreException** lors de toute tentative d'y stocker un élément du mauvais type, au lieu de **ClassCastException** lors de son utilisation (donc bien plus tard).
  - « Genre de » généricité, mais conception obsolète : avec la généricité moderne, la compilation garantit une exécution sans erreur.
1. Raison : un tableau est à la fois producteur et consommateur. D'un point de vue théorique, une telle structure de données ne peut être qu'invariante, si on veut des garanties dès la compilation.

### Génériques et tableaux

Différence de philosophie (2)

Compléments en P00  
Aldric Degorre

Introduction	Généralités	Compléments en P00
Généralités	Style	Aldric Degorre
Style	Objets et classes	
Objets et classes	Types et polymorphisme	
Types et polymorphisme	Héritage	
Héritage	Généricité	
Généricité	Introduction	
Introduction	Effacement de type	
Effacement de type	Collections	
Collections	Options	
Options	Lambda-expressions	
Lambda-expressions	Les "strangs"	
Les "strangs"	Instance des génériques vs. covariance des tableaux	
Instance des génériques vs. covariance des tableaux	Villefranche	
Villefranche	Concurrence	
Concurrence	Interfaces graphiques	
Interfaces graphiques	Concurrence	
Concurrence	Interfaces graphiques	

- Génériques : (vérification à la compilation... puis plus rien)
- **usage normal**, le compilateur rejette toute tentative de conversion implicite de **Gen<A>** à **Gen<B>**, garantissant qu'à l'exécution toute instance de **Gen<T>** sera bien utilisée avec le type **T** → exécution cohérente et sans exception garantie.
- **usage anormal**, conversion forcée : (**Gen<B>**) **new Gen<A>()** compile avec un warning et... provoque des erreurs à retardement à l'exécution (très mal, mais on a été prévenu) ! Exemple :

```
List<String> ls = new ArrayList<String>();  
List<Integer> li = (List<Integer>) Object ls; // exécution ok ! (oh !)  
li.add(5); // toujours pas de crash... (double oh !)  
ls.get(0); // BOOM à retardement ! (ClassCastException)
```

### Génériques et tableaux

Et si on mélangeait ?

### Génériques et tableaux

Différence de philosophie (1)

Compléments en P00  
Aldric Degorre

Introduction	Généralités	Compléments en P00
Généralités	Style	Aldric Degorre
Style	Objets et classes	
Objets et classes	Types et polymorphisme	
Types et polymorphisme	Héritage	
Héritage	Généricité	
Généricité	Introduction	
Introduction	Effacement de type	
Effacement de type	Collections	
Collections	Options	
Options	Lambda-expressions	
Lambda-expressions	Les "strangs"	
Les "strangs"	Instance des génériques vs. covariance des tableaux	
Instance des génériques vs. covariance des tableaux	Villefranche	
Villefranche	Concurrence	
Concurrence	Interfaces graphiques	
Interfaces graphiques	Concurrence	
Concurrence	Interfaces graphiques	

- Tableaux : (vérification à l'exécution, mais le + tôt possible)
- **usage normal** : conversion sans warning de **SousType[]** à **SuperType[]** par upcasting (implicite). Possibilité d'**ArrayStoreException** à l'exécution.
- Object[] tab = new String[10];  
tab[0] = Integer.valueOf(3); // BOOM ! (ArrayStoreException)
- Pas idéal, mais aurait pu être pire : le crash évite que le programme continue avec une mémoire incohérente.
- **usage anormal**, avec cast explicite vers type incompatible :  
(String[]) (Object) (new Integer[10]) compile mais avec warning et fait ClassCastException quand on exécute (tout va bien : on avait été prévenu).

⇒ Tout ce qui est dans le tableau pouvant ensuite être affecté à une variable de type **List<Integer>**, la garantie promise par la généricité serait cassée.

<h2>Génériques et tableaux</h2> <p>Premier interdit : <code>new T[10]</code></p> <p><b>Supposons <code>T extends Up</code> paramètre de type.</b></p> <p><b><code>new T[10]</code> est aussi interdit.</b></p> <p><b>Raison :</b> après compilation, <code>T</code> est oublié et remplacé par <code>Up</code>. Au mieux <code>new T[10]</code> pourrait être compilé comme <code>new Up[10]</code>. Mais si c'était ce qui se passait, on pourrait trop facilement « polluer » la mémoire sans s'en rendre compte :</p> <pre> static &lt;T&gt; T[] makeArray(int size) { return new T[10]; /* interdit ! */ }  String[] tString = makeArray(10); // à l'exécution on affacterait un Object[]  Object[] tobject = tString; // toujours autorisé (covariance)  tobject[0] = 42; // et BOOM ! Maintenant tString contient un Integer !     </pre> <p>En pratique, pour faire compiler cela, il faut « tricher » avec cast explicite :</p> <pre> T[] tab = (T[]) new Up[10];     </pre> <p>Ça n'empêche pas le problème ci-dessus, mais au moins le compilateur affiche un warning (« <code>unchecked conversion</code> »).</p>	<p>Compléments en P00 Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité introduction Effacement de type collecte Objets Lambdalexpressions Les "strangs" Instance des génériques vs covariance des tableaux Wildcards Concurrence Interfaces graphiques</p>	<p>Compléments en P00 Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité introduction Effacement de type collecte Objets Lambdalexpressions Les "strangs" Instance des génériques vs covariance des tableaux Wildcards Concurrence Interfaces graphiques</p>
<h2>Génériques et tableaux</h2> <p>Deuxième interdit : <code>Gen&lt;T&gt;[1]</code> (1)</p> <p><b>Mauvais scenario, pouvant se produire si on autorisait les tableaux de génériques :</b></p> <pre> class Box&lt;T&gt; {     final T x;     Box(T x) { this.x = x; }      class Lophole {         public static void main(String[] args) {             Box&lt;String&gt;[] bsa = new Box&lt;String&gt;[3]; // supposons que cette ligne compile             Object[] oa = bsa; // autorisé car tableaux covariants              /* autorisé à la compilation (Box &lt; Object) le test à l'exécution est aussi ok             (parce que le tableau référencé par oa est celui instancié à la première ligne             et que le type enregistré dans la JVM est juste Box) */              oa[0] = new Box&lt;Integer&gt;(3);              /* ... et là, c'est le drame !             (ClassCastException, alors que l'instruction est bien typée !) */              String s = bsa[0].x;         }     } }     </pre>	<p>Compléments en P00 Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité introduction Effacement de type collecte Objets Lambdalexpressions Les "strangs" Instance des génériques vs covariance des tableaux Wildcards Concurrence Interfaces graphiques</p>	<p>Compléments en P00 Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité introduction Effacement de type collecte Objets Lambdalexpressions Les "strangs" Instance des génériques vs covariance des tableaux Wildcards Concurrence Interfaces graphiques</p>
<h2>Génériques et tableaux</h2> <p>Deuxième interdit : <code>Gen&lt;T&gt;[1]</code> (2)</p> <p><b><code>new Box&lt;Integer&gt;[10]</code> est interdit.</b></p> <p>En effet, le tableau instancié serait de type <code>Box[String][1]</code><sup>1</sup>, ce qui rendrait possible le scénario du transparent précédent.</p> <p>En revanche, Java autorise <code>new Box[10]</code>.</p> <p><b>Remarque</b>, du coup, là aussi, il existe une « triche » pour faire compiler l'exemple du transparent précédent : on remplace <code>new Box&lt;String&gt;[3]</code> par <code>new Box[3]</code>. Le compilateur émet heureusement un warning (« <code>unchecked conversion</code> »)… et à l'exécution, on a effectivement <code>ClassCastException</code></p> <p>Encore une fois, la « triche » permet de compiler, n'empêche pas l'exception à l'exécution, mais seulement on a été prévenu par le warning du compilateur !</p>	<p>Compléments en P00 Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité introduction Effacement de type collecte Objets Lambdalexpressions Les "strangs" Instance des génériques vs covariance des tableaux Wildcards Concurrence Interfaces graphiques</p>	<p>Compléments en P00 Aldric Degorre</p> <p>Introduction Généralités Style Objets et classes Types et polymorphisme Héritage Généricité introduction Effacement de type collecte Objets Lambdalexpressions Les "strangs" Instance des génériques vs covariance des tableaux Wildcards Concurrence Interfaces graphiques</p>

Invariance des génériques → garanties fortes : très bien, mais... très rigide à l'usage!

**Le besoin :** quand `B <: A`<sup>1</sup>, on aimerait pouvoir écrire `Gen<A> g = new Gen<B>();`

- Cela favoriserait le polymorphisme (par sous-typage).
- On le fait bien avec les tableaux (`Object[] t = new String[10];`).
- C'est souvent conforme à l'intuition (cf. tableaux).

**Mais on sait que ça risque d'être difficile :**

- On a vu un contre-exemple pathologique (on provoque facilement `ClassCastException` si on force le compilateur à outrepasser l'invariance).
- On a vu les problèmes que posent les tableaux covariants (`ArrayListException` possible même dans programme sans `warning`).

1. À cause de l'effacement de type, `Box<Integer>` n'existe pas à l'exécution. Toutes les spécialisations ont le même type : `Box` !

Pour les quelques pages qui suivent, **oublions que javac impose l'invariance.**  
**Question :** parmi les variables `x`, `y`, `z` et `t`, ci-dessous, lesquelles devrait-on, idéalement<sup>1</sup>, pouvoir affecter à quelles autres ?

```
class A {}  
class B extends A {}  
// Interface pour fonctions F -> U (extraite de java.util.function)  
interface Function<T,U> { U apply(T t); }  
class Test {  
    Function<A,B> x;  
    Function<A,B> y;  
    Function<B,A> z;  
    Function<B,B> t;  
}
```

**Le critère :** on cherche les cas où une instance `Function<X,Y>` fournit au moins le service d'une instance de `Function<Z,T>`.

1. par exemple dans un langage où les génériques pourraient ne pas être invariants

- Paramètre utilisé en entrée (= type de paramètre de méthode ou type d'attribut public modifiable) → encoché.
- Paramètre utilisé en sortie (= type de retour de méthode, type d'attribut public quelconque) → excroissance.

L'encoche (resp. excroissance) pour un type donné doit contenir celles de ses sous-types.  
**Exemple :**

`Function<T, U>`



(L'encoche à gauche représente `T` et l'excroissance à droite, `U`.)

---

1. Attention, on ne parle pas de Java, mais seulement d'un système de type « idéal ».

Compléments en 200	Aldric Degorre
<p><b>Introduction</b></p> <p>Généralités</p> <p>Style</p> <p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Généricité : introduction</p> <p>Élément de type</p> <p>Collectives</p> <p>Optionnels</p> <p>Lambda-expressions</p> <p>Les "streams"</p> <p>Instance des génériques vs covariance des tableaux</p> <p>Wildcards</p> <p>Concurrence</p> <p>Interfaces graphiques</p>	<p>Introduction</p> <p>Généralités</p> <p>Style</p> <p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Généricité : introduction</p> <p>Élément de type</p> <p>Collectives</p> <p>Optionnels</p> <p>Lambda-expressions</p> <p>Les "streams"</p> <p>Instance des génériques vs covariance des tableaux</p> <p>Wildcards</p> <p>Concurrence</p> <p>Interfaces graphiques</p>
<p>La variance souhaitée n'est donc pas la même pour tous les types génériques :</p> <ul style="list-style-type: none"> <li>intuitif et logique de vouloir <code>Function&lt;Object, Integer&gt; &lt;: Function&lt;Double, Number&gt;</code>.</li> </ul> <p><b>Justification</b> : le premier paramètre de type est utilisé uniquement pour l'argument de <code>apply</code> alors que l'autre est uniquement son type de retour.</p> <p>→ Emboîtement de <code>Function&lt;T, U&gt;</code> dans <code>Function&lt;V, W&gt;</code> possible dès que <code>T &gt; V</code> et <code>U &lt;: W</code>.</p> <p>Remarque : tailles de l'encoché et de l'excroissance de <code>Function&lt;T, U&gt;</code> indépendantes l'une de l'autre car elles représentent 2 paramètres différents. Si le même paramètre de type est utilisé en entrée et en sortie, ça ne marche plus (cf. page d'après).</p>	<p>La variance souhaitée n'est donc pas la même pour tous les types génériques :</p> <ul style="list-style-type: none"> <li>intuitif et logique de vouloir <code>Function&lt;Object, Integer&gt; &lt;: List&lt;Number&gt;</code> serait illogique.</li> </ul> <p><b>Justification</b> : Le même paramètre apparaît à la fois en <u>sortie</u> (méthode <code>get</code>) et en <u>entrée</u> (méthodes <code>set</code> et <code>add</code>)<sup>1</sup>. Donc tailles de l'encoche et de l'excroissance de <code>List&lt;T&gt;</code> liées car représentant le même <code>T</code></p> <p>→ impossible d'encastrer la pièce de <code>List&lt;x&gt;</code> dans le trou <code>List&lt;y&gt;</code> si <code>x ≠ y</code>.</p>

## Encore un peu de vocabulaire autour de la variance (1)

Compléments en 200	Aldric Degorre
<p><b>Introduction</b></p> <p>Généralités</p> <p>Style</p> <p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Généricité : introduction</p> <p>Élément de type</p> <p>Collectives</p> <p>Optionnels</p> <p>Lambda-expressions</p> <p>Les "streams"</p> <p>Instance des génériques vs covariance des tableaux</p> <p>Wildcards</p> <p>Concurrence</p> <p>Interfaces graphiques</p>	<p>Introduction</p> <p>Généralités</p> <p>Style</p> <p>Objets et classes</p> <p>Types et polymorphisme</p> <p>Héritage</p> <p>Généricité</p> <p>Généricité : introduction</p> <p>Élément de type</p> <p>Collectives</p> <p>Optionnels</p> <p>Lambda-expressions</p> <p>Les "streams"</p> <p>Instance des génériques vs covariance des tableaux</p> <p>Wildcards</p> <p>Concurrence</p> <p>Interfaces graphiques</p>

## Encore un peu de vocabulaire autour de la variance (2)

Compléments en 200	Aldric Degorre
<p>Dans <code>Function&lt;T, U&gt;, T et U ont des influences contraires</code> l'une de l'autre à cause de leur usage dans la méthode de <code>Function</code>.</p> <p>→ 2 catégories d'usage :</p> <ul style="list-style-type: none"> <li>en <b>position covariante</b> : utilisé comme type de retour de méthode (ou type d'attribut)</li> <li>en <b>position contravariante</b> : utilisé comme type d'un argument dans la signature d'une méthode (ou comme type d'un attribut non <b>final</b>)</li> </ul>	<p>Dans <code>Function&lt;T, U&gt;, T et U ont des influences contraires</code> l'une de l'autre à cause de leur usage dans la méthode de <code>Function</code>.</p> <p>→ 3 catégories de paramètres de type :</p> <ul style="list-style-type: none"> <li><b>paramètre covariant</b> (comme <code>U</code>) : utilisé seulement en position covariante → <u>plus</u> le paramètre effectif est petit, <u>plus</u> le type paramétré <u>devrait</u> être petit;</li> <li><b>paramètre contravariant</b> (comme <code>T</code>) : utilisé seulement en position contravariante → <u>plus</u> le paramètre effectif est petit, <u>plus</u> le type paramétré <u>devrait</u> être grand;</li> <li><b>paramètre invariant</b> : utilisé à la fois en position covariante et contravariante.</li> </ul> <p><b>Attention, ces concepts ne sont que théoriques.</b></p> <p>Il se trouve que ceux-ci n'ont <b>pas de sens pour le compilateur de Java</b> :appelez-vous qu'on avait dit que, pour l'instant, on oubliait l'invariance imposée par Java.</p>

## Gérer la variance dans un programme

2 approches principales pour prendre en compte le phénomène de la variance :

- **annotations de variance sur site de déclaration** (n'existent pas en Java)

Variance définie (définitivement) dans la déclaration du type générique.

Exemple en langage Kotlin, on utilise **in** (contravariance) et **out** (covariance) :

```
interface Function<in T, out U> { fun apply(t: T) : U }
class A
class B : A
```

Alors, dans cet exemple, **Function<A, B><: Function<B, A>**.

- **annotations de variance sur site d'usage**

Variance choisie lors de l'usage d'un type générique (dans déclarations de variables et signatures de méthodes).

C'est l'approche utilisée par Java, via le mécanisme des **wildcards**.

1. Le compilateur de Kotlin vérifie que les paramètres covariants (resp. contravariants) sont effectivement uniquement utilisés en position covariante (resp. contravariante).

### Compléments en 200

#### Introduction

#### Généralités

#### Style

#### Objets et classes

#### Types et polymorphisme

#### Héritage

#### Généricité

#### Introduction

#### Effacement de type

#### Collectives

#### Options

#### Lambda-expressions

#### Les "strangs"

#### Instance des génériques vs covariance des tableaux

#### Wildcards

#### Concurrence

#### Interfaces graphiques

#### Autres

### (On revient enfin à Java !)

#### Types et polymorphisme

#### Héritage

#### Généricité

#### Introduction

#### Effacement de type

#### Collectives

#### Options

#### Lambda-expressions

#### Les "strangs"

#### Instance des génériques vs covariance des tableaux

#### Wildcards

#### Concurrence

#### Interfaces graphiques

#### Autres

- Quand on écrit un type paramétré, les paramètres peuvent en fait être soit des types, soit le symbole « ? » (symbolisant un joker, un **wildcard**), parfois muni d'une  **borne**.
- Les types paramétrés dont le paramètre est compatible avec la borne du **wildcard** se comportent alors comme des sous-types du type contenant le **wildcard**.

Ainsi **List<Integer>** est sous-type de **List<?>**.

**Remarque :** « ? » tout seul n'est pas un type. Ce caractère ne peut être utilisé que pour écrire un type paramétré (entre « < » et « > »).

### Compléments en 200

#### Introduction

#### Généralités

#### Style

#### Objets et classes

#### Types et polymorphisme

#### Héritage

#### Généricité

#### Introduction

#### Effacement de type

#### Collectives

#### Options

#### Lambda-expressions

#### Les "strangs"

#### Instance des génériques vs covariance des tableaux

#### Wildcards

#### Concurrence

#### Interfaces graphiques

#### Autres

### Wildcards

#### bornes

Toute occurrence de « ? » peut se voir associer une borne.

Le principe est similaire aux bornes de paramètres de type, avec quelques différences :

- « ? » bornable à chaque usage (or, paramètres bornables juste à leur introduction).
- Les « ? » admettent des bornes supérieures (**T<?** **extends A**), **mais aussi des bornes inférieures** (**T<?** **super A**), imposant que toute concrétisation doit être un supertype de la borne.

- Pour un « ? », Java autorise une seule borne à la fois.

1. Si on veut plusieurs types concrets comme bornes supérieures, il est possible de contourner cette limite en introduisant un type intermédiaire : **Interface Borne extends Borne1, Borne2 ...**. Combiner plusieurs bornes inférieures concrètes (disons **A** et **B**) ne sert à rien : en effet **A** et **B** ont nécessairement un plus petit supertype commun, **C**, qui a un nom déjà connu quand on écrit le programme. **C** est le plus petit type contenant **A ∪ B** (qui n'est pas un type de Java). Ainsi **T?** **super C** serait équivalent à **? super A ∪ B** (syntaxe fictive).
- Si non, pour mixer des bornes qui sont elles-mêmes des paramètres, d'autres techniques basées sur l'introduction d'une variable de type supplémentaire sont envisageables.

## Wildcards

Vérification du type

## Wildcards

Vérification du type

### L'affectation suivante est-elle bien typée ?

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

Pour savoir, on vérifie si le terme droite de l'affectation a un type compatible avec son emplacement.

Son type est `ArrayList<String>`, or le type attendu à cet emplacement est `List<? extends Serializable>` (= type de la variable à affecter).

- D'une part, `String` satisfait la borne de ? (`String` implémente `Serializable`)
- et, d'autre part, `ArrayList<String> <: List<String>`.

Donc cette affectation est bien typée.

Compléments  
en POO

Aldric Degorre

## Wildcards

Vérification du type

### Généralisons :

- Soit `TE` une expression convertie par capture<sup>1</sup>, si applicable) de `expr`.
- Soit `TE` le type (déjà « converti par capture », si applicable) de `expr`.
- Supposons que le type attendu dans le contexte soit de la forme `TA<?` **borne**.
- Alors il est légal d'utiliser `expr` à cet endroit si et seulement si il existe un type `T` satisfaisant `borne`, tel que `TE <: TA<T>`.

- Expliquer un peu plus loin. Ceci concerne le cas où le type de l'expression contient un ?.

## Wildcards

utilisation pour signaler la variance

## Wildcards

Conversion par capture (1)

## Wildcards

Conversion par capture (1)

## Wildcards

Conversion par capture (1)

Pour revenir au problème initial, reprenons notre exemple :

```
interface Function<T, U> { U apply(T t); }
class A {}
class B extends A {}
class Test { Function<A, A> x; Function<A, B> y; Function<B, A> z; Function<B, B> t; }
```

`U` → position covariante; `T` → position contravariante. On « assouplit » donc `Test` :

```
class Test2 {
    Function<? super A, ? extends A> x; Function<? super A, ? extends B> y;
    Function<? super B, ? extends A> z; Function<? super B, ? extends B> t;
}
```

Maintenant, les affectations qu'on voulait écrire sont acceptées par le compilateur :

```
z = t; z = x; t = y; x = y; // vérifiez !
/* et aussi... */ a = b; a = x. apply(a); b = y. apply(b); a = z. apply(b); b = t. apply(b);
```

**Recette :** position covariante → **extends**, position contravariante → **super**.

Se rappeler **PECS** : « producer extends, consumer super ».

Inversez **super** et **extends** et vérifiez que les appels à **apply** ne fonctionnent plus.

## Wildcards

Vérification du type

1. Et comme on ne sait rien de ce type, vérifier que l'expression est à sa place c'est vérifier qu'elle est à sa place **pour toute** valeur de **QuelqueChose**.

Concrètement, lors de la vérification de type d'une expression, le compilateur effectue une opération appelée **conversion par capture**<sup>1</sup> :

- À chaque fois qu'un « ? » apparaît au premier niveau<sup>2</sup> du type d'une expression<sup>3</sup>, le compilateur le remplace par un nouveau type créé à la volée, recevant un nom de la forme **capture#1-of?** (**capture de wildcard**).
- Quand une telle capture est créée, le compilateur se souvient des bornes du « ? » qu'elle remplace (il peut s'en servir dans la suite de l'analyse de types).

1. Pour les logiciens, cette transformation s'apparente à la skolémisation : on remplace une variable quantifiée existentiellement par un nouveau symbole.
2. On ne regarde pas en profondeur : **List<?>** devient **Set<capture#1-of?>**, le compilateur se rappelant que **capture#1-of?** est sous-type de **Set<?>**.
3. Cela se produit quand l'expression est une variable typée avec des ?, un appel de méthode dont le type de retour contient des ?, ou une expression castée vers un tel type.

## Wildcards

### Conversion par capture (2)

Compléments  
en 200

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Généricité

Introduction

Effacement de type

Collections

Options

Lambdas-expressions

Les « ? »

Inheritance des généricités vs covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

### Exemple :

- soit une expression : **new HashMap<? super String, ? extends List<?>()**,
- son type « brut » : **HashMap<? super String, ? extends List<?>>**,
- son type après conversion par capture : **HashMap<capture#1-of?, capture#2-of?>**.
- Le compilateur se rappelle que **capture#1-of? >: String** et que **capture#2-of? <: List<?>**.

Cette conversion a lieu à chaque fois que le type d'une expression est évalué. Ainsi, une expression composite peut contenir plusieurs captures différentes accumulées depuis l'analyse du type de ses sous-expressions.

Conversion des exemples plus complexes ? (1)

Compléments  
en 200

Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Types et polymorphisme

Héritage

Généricité

Généricité

Introduction

Effacement de type

Collections

Options

Lambdas-expressions

Les « ? »

Inheritance des généricités vs covariance des tableaux

Wildcards

Concurrence

Interfaces graphiques

**Explication : à la 2e ligne,**

- la 1e occurrence de **l** est de type **List<capture#1-of?>**  $\Rightarrow$  **l.add(...)** attend un paramètre de type **capture#1-of?** ;
- la 2e occurrence de **l** est de type **List<capture#2-of?>** (**capture indépendante**)  $\Rightarrow$  **l.get(0)** est de type **capture#2-of?** ;
- or **capture#1-of?** et **capture#2-of?** sont, du point de vue du compilateur, deux types quelconques sans lien de parenté, d'où l'erreur de type.

On peut contourner en forçant une capture anticipée (via méthode auxiliaire) :

```
// méthode auxilliaire. Ici, tout est ok, car l.get(0) de type T, or l.add() prend du T.
<T> static void aux(List<T> l) { l.add(l.get(0)); }

// plus loin
List<? super String> l = new ArrayList<?>; aux(l); // encore ok
```

Les « ? » peuvent apparaître à différentes profondeurs, y compris dans les bornes :

```
List<A<? super String>> las = new ArrayList<?>();
List<?> extends A<? super Integer> lar = las;
```

Pour chaque niveau de <> on vérifie que le type (resp. l'ensemble de types) donné correspond à un élément (resp. un sous-ensemble) de l'ensemble attendu.

```
List<A<? super String>> las = new ArrayList<()>;
List<? extends A<? super Integer>> lar = las;
```

Dans l'exemple (2e ligne, à droite de =) :

- On veut comparer `List<A<? super String>>` et `List<? extends A<? super Integer>>` (type reçu = celui de `las`) et (type attendu = celui de `lar`).
- Au premier niveau, on a `List` et `List` → OK, vérifions les paramètres.
- Il faut que `A<? super String><: A<? super Integer>`.
- On a `A` des 2 côtés... jusqu'à tout va bien. Vérifions l'inclusion des paramètres.
- À l'intérieur on attend « `? super Integer` », mais on reçoit « `? super String` ».
- Les deux bornes sont dans le même sens, c'est bon signe.
- Malheureusement, on n'a pas `String >: Integer`. Donc « `? super String` » n'est pas inclus dans « `? super Integer` » (p. ex. : le 1er ensemble contient `String` mais pas le 2e). Donc erreur de type !