Exemple (de l'API) :

```
public interface Comparator<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. [1]

Types génériques du JDK :

- Les collections de Java ≥ 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 ≥ 8 (pour les lambda expressions).
- `Optionnal`, `Stream`, `Future`, `CompletableFuture`, `ForkJoinTask`, ...

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

Généricité

Pourquoi ? (1)

La généricité est un procédé permettant d'augmenter la réutilisabilité du code de façon maîtrisée [1] grâce à des relations fines entre les types utilisés.

Sur un exemple :

```
class Boite { // non polymorphe
  public int x;
  void echange(Boite autre) {
    int ech = x; x = autre.x; autre.x = ech;
  }
}
```

Inconvénient : définition qui ne marche que pour les boîtes à entiers.

Réutilisabilité : proche de zéro !

1. Par opposition au polymorphisme par sous-typage, où, par exemple, pour les arguments d'appel de méthode, tout sous-type fait l'affaire indépendamment des autres types utilisés en argument.

Généricité

Pourquoi ? (2)

Première solution : boîte universelle (polymorphisme par sous-typage)

```
class Boite { // très (trop ?) polymorphe
  public Object x; // contient des Object, 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 [1]) 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 `Boite` revient à abandonner le bénéfice du typage statique (pourtant une des forces de Java).

Cas d'utilisation problématique :

```
Boite b1 = new Boite(), b2 = new Boite(); b1.x = 6; b2.x = "toto";
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 [1].

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 aimerait éviter (en plus d'être une lourdeur à l'écriture du programme).

La bonne solution : boîte générique (→ **polymorphisme générique**)

```
class Boite<C> {
    public C x;
    void echange(Boite<C> autre) { C ech = x; x = autre.x; autre.x = ech; }
}
... // plus loin :
        Boite<Integer> b1 = new Boite<>(); Boite<String> b2 = new Boite<>();
        b1.x = 6; b2.x = "toto";
        System.out.println(7 * b1.x); // <- là  c'est toujours ok (et sans cast, SVP !)
        // 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 dès de la compilation [1] et uniquement à la compilation. Celle-ci est oubliée aussitôt [2] (**effacement de type** / *type erasure*).

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.

Types génériques et types paramétrés

Usage de base (1)

- **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.

Types génériques et types paramétrés

Usage de base (2)

- **À l'usage**, le type générique sert de constructeur de type : on remplace le paramètre par un type concret et on obtient un **type paramétré**.
  Exemple : `List` est un type générique, `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 [1] !

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

Types génériques et types paramétrés

Usage de base (3)

- Utilisation de classe générique par instanciation directe :

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

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

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

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

Types génériques et types paramétrés

Usage de base (4)

- Utilisation de classe générique par extension non générique (**spécialisation**) :

```
class TroisChars extends Triplet<Char, Char, Char> {
  public TroisChars(Char x, Char y, Char z) { super(x,y,z); }
}
```

`TroisChars` étend la classe paramétrée `Triplet<Char, Char, Char>`.

- Variante, spécialisation partielle :

```
class DeuxCharsEtAutre<T> extends Triplet<Char, Char, T> {
  public DeuxCharsEtAutre(Char x, Char y, T z) { super(x,y,z); }
}
```

La classe générique `DeuxCharsEtAutre<T>` étend la classe générique partiellement paramétrée `Triplet<Char, Char, T>`.

Parallèle entre type générique et méthode

La déclaration et l'utilisation des types génériques rappellent celles des méthodes.

Similitudes :

- introduction des paramètres (de type ou de valeur) dans l'en-tête de la déclaration ;
- utilisation des noms des paramètres dans le corps de la déclaration seulement ;
- pour utiliser le type générique ou appeler la méthode, on passe des concrétisations des paramètres.

Principales différences :

- Les paramètres des génériques représentent des types alors que ceux des méthodes représentent des valeurs.
- Pour les paramètres de type, le « remplacement »[1] a lieu à la compilation.
  Pour les paramètres des méthodes, remplacement par une valeur à l'exécution.

1. Rappel : remplacement oublié, effacé, aussitôt que la vérification du bon typage a été faite.

Types bruts

- Un nom de type générique seul, sans paramètre (comme « `Triplet` »), est aussi un type légal, appelé un **type brut** (***raw type***).
  Son utilisation est **fortement déconseillée**, mais elle est permise pour assurer la compatibilité ascendante [1].
- Un type brut est supertype direct [2] de tout type paramétré correspondant (ex : `Triplet` est supertype direct de `Triplet<Number, Object, String>`).
- Pour faciliter l'écriture, le *downcast* implicite [3] est malgré tout possible :

```
List l1 = new ArrayList(); // déclaration de l1 avec raw type
List<Integer> l2 = l1; // downcast implicite de l1 vers type paramétré
```

compile avec l'avertissement `unchecked conversion` sur la deuxième ligne.

1. Un source Java < 5 compile avec `javac` $\geq$ 5. Or certains types sont devenus génériques entre temps.
2. C'est une des règles de sous-typage relatives aux génériques, omises dans le début de ce cours.
3. Je crois que c'est l'unique occurrence de *downcast* implicite en Java.

Déclaration de paramètre de type niveau méthode

- Il est aussi possible d'introduire un paramètre de type dans la signature d'une méthode (possible aussi dans une classe non générique) :

```
static <E> List<E> inverseListe(List<E> l) { ... ; E x = get(0); ...; }
```

- Dans l'exemple ci-dessus, on garantit que la liste retournée par `inverseListe()` a le même type d'éléments que celle donnée en paramètre.
- **Usages possibles :**
  - contraindre plusieurs types apparaissant dans la signature de la méthode à être le même type, sans pour autant dire lequel ;
  - 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).

Remarque : 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.

Bornes de paramètres de type (1)

- Pour limiter les concrétisations autorisées, un paramètre de type admet des **bornes supérieures** [1] (se comportant comme supertypes du paramètre) :

```
class Calculator<Data extends Number>
```

Ici, `Data` devra être concrétisé par un sous-type de `Number` : une instance de `Calculator` travaillera sur nécessairement avec un certain sous-type de `Number`, celui choisi à son instanciation.

1. On verra dans la suite que les bornes inférieures existes aussi, mais elles ne s'appliquent qu'aux *wildcards* (et non aux paramètres de type).

Bornes de paramètres de type (2)

- 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** [1], 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... .

Bornes de paramètres de type (3)

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.

Effacement de type (*type erasure*)

De quoi il s'agit.

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 connait que les types réifiables.
Autrement dit : la paramétrisation générique n'a pas d'impact à l'exécution.

Plus de détails juste après.

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 [1] 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 [2]. Les « objets paramétrés » n'existent pas.

1. `invokestatic`, `invokespecial`, `invokevirtual` 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 parmétrisation.

Effacement de type (*type erasure*)

Explication

Remarque : le code-octet contient tout de même encore des types non réifiables. C'est le cas pour les « signatures [1] » des classes et de leurs membres. Cela est indispensable pour vérifier les types génériques lors de la compilation des classes dépendantes.

Il est donc faux que le code-octet ne sait déjà plus rien de la paramétrisation générique d'une classe.

Mais cette information est à destination du compilateur, et non pas de la JVM [2].

1. Signature au sens de la JVM. Pour la JVM, la signature contient toute l'information de typage d'une entité donnée. Par exemple, pour une méthode, c'est son type de retour et les types de ses paramètres. Cette notion est donc différente de la notion de signature dans un code source Java. Elle est aussi différente de la notion de descripteur tout juste évoquée.
2. Ceci dit, il est possible de lire les signatures pendant l'exécution (grâce à la réflexion). Mais cela ne permet en aucun cas de savoir quels paramètres de types effectifs ont été utilisés pour instancier un objet donné ou exécuter une méthode donnée.

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 :

- Faute d'être raisonnablement exécutables, ces expressions ne compilent pas :
  - `x instanceof P` (avec `P` paramètre de type) ;
  - `x instanceof TypeG<Y>` [1] (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 { ... }
```

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

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; }
}
```

- 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 extends ArrayList<Integer> implements List<String> { ... }
```

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.

$\rightarrow$ alors à quoi servent les paramètres de type ?

$\rightarrow$ 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 : $\forall T\,[(\forall B \in \mathit{UpperBounds}(T), T <: B) \Rightarrow \mathit{WellTyped}(\mathit{prog}(T))]$
où *prog* est soit une classe, soit une méthode générique.

Collections : pourquoi ?

- **Besoin :** représenter des « paquets », des « collections » d'objets similaires.
- **Plusieurs genres de paquets/collections :** avec ou sans doublon, accès séquentiel ou aléatoire (= par indice), avec ou sans ordre, etc.
- **Mais nombreux points communs :** peuvent contenir plusieurs éléments, possibilité d'itérer, de tester l'appartenance, l'inclusion etc.
- Pour chaque « genre » plusieurs représentations/implémentations de la structure (optimisant telle ou telle opération...).

Collections : pourquoi ?

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

- Les tableaux [1].
- Les collections non génériques [2] de Java < 5, avec leurs éléments de type statique `Object`, qu'il fallait *caste*r avant usage.

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

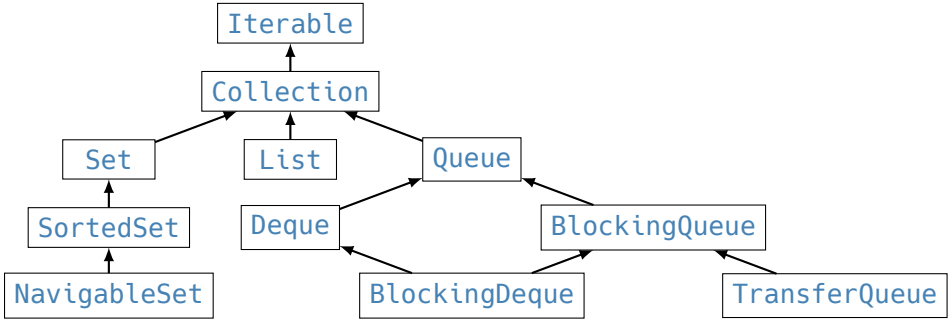
Les collections justifient à elles seules l'introduction de la généricité dans Java.

---

1. 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.
2. **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 ≥ 5, remplacez `ArrayList` par `ArrayList<TypeElems>`.

La hiérarchie des sous-interfaces d'`Iterable`

Interfaces de `java.util` et `java.util.concurrent`[1]

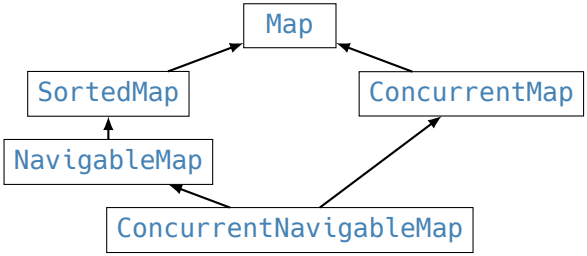


Chacune de ces interfaces possède une ou plusieurs implémentations.

1. 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`[1]



Chacune de ces interfaces possède une ou plusieurs implémentations.

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

L'interface `Iterable` (1)


```
public interface Iterable<E> {
    Iterator<E> iterator(); // cf. Iterator
    default Spliterator<E> spliterator() { ... } // pour les Stream
    default void forEach(Consumer<? super T> action) { ... } // utiliser avec lambdas
}
```

Un `Iterable` représente une séquence qu'on peut « itérer » (à l'aide d'un **itérateur**).

- soit avec la construction « *for-each* » (conseillé !) :

```
for ( Object o : monIterable ) System.out.println(o);
```

- soit avec la méthode `forEach` et une lambda-expression (cf. chapitre dédié) :

```
monIterable.forEach(System.out::println);
```

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`[1] :

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

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

1. 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`.

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 implémentée, 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 `Collection`

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();                               // opération optionnelle
    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

- Pas de méthodes autres que celles héritées de `Collection`.
- **Différence :** 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); s.add(1);
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 `SortedSet`

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`.

L'interface `List` (1)

`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 :

- acces positionnel (on peut accéder au *i*-ième élément)
- recherche (si on connait 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` (2)

Mais aussi :

- itérateurs plus riches (peuvent itérer en arrière)
- « vues » de sous-listes [1]

```
...

    // 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.

Itérateurs de liste ?

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);
}
```

L'interface `List` (3)

Implémentations principales : `ArrayList` (basée sur un tableau, avec redimensionnement dynamique), `LinkedList` (basée sur liste chainée).

Exemple :

```
ArrayList<Integer> l = new ArrayList<Integer>();
l.add(1); l.add(2); l.add(3); l.add(1);
for (int i : l) System.out.print(i + ", ");
System.out.println("\n3e element: " + l.get(2));
l.set(2,9);
System.out.println("Nouveau 3e element: " + l.get(2));
```

Ceci affichera :

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

L'interface Queue

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).

L'interface Deque (1)

`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();
...
```

L'interface Deque (2)


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

Implémentations typiques : `ArrayDeque`, `LinkedList`

L'interface Map (1)

Une `Map` est un ensemble d'associations (clé $\mapsto$ 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(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
...
```

L'interface Map (2)


```
...
   // 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);
    }
}
```

Implémentation la plus courante : la classe `HashMap`

L'interface `SortedMap`

`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

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

- Nombreuses dans la classe `Collections` [1] : 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 $\geq$ 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'

Collections et tableaux

Quand les utiliser

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 prototypez un programme très rapidement et vous appréciez la simplicité
- si vous souhaitez optimiser la performance au maximum [1] [2]

---

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 ?

Les optionnels

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)

Solutions (pas très bonnes) :

- retourner une valeur qui peut être `null` (« **nullable** »). **Inconvénients :** [1]
  - 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 (déboguage 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 !

Les optionnels

Une solution pas trop mauvaise : [1] la classe `java.util.Optional` [2]

- 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.

1. Les valeurs nullables gardent quelques avantages sur `Optional` : pas besoin d'allouer un conteneur supplémentaire, moins de lourdeurs syntaxiques (comme la nécessité 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).
2. Inspirée des langages fonctionnels : la classe `Option` en Scala, la monade `Maybe` en Haskell

Les optionnels

Pourquoi c'est plus sûr qu'un type nullable :

- On ne peut pas appeler directement les méthodes de `T` sur une expression de type `Optional<T>` : il faut d'abord extraire son contenu (méthode `get`).
- Ainsi pas de risque de `NullPointerException` (ni sur l'instance d'`Optional<T>` ni sur le résultat de `get()`).
- `get` peut bien lancer `NoSuchElementException`, mais ça se produit là où `get` est appelée. On voit donc tout de suite si et où on a oublié d'appeler `isPresent`.

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 résultat 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 optionnel contenant `elem` (sinon `NullPointerException`)
- `<T> Optional<T> empty()` : retourne un optionnel 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

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.

Fonctions de première classe et fonctions d'ordre supérieur

Le besoin (1)

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 repeatF(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).

Fonctions de première classe et fonctions d'ordre supérieur

Le besoin (2)

- 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 évènement ultérieur, ...) ;
  - exécutées dans un autre contexte (p. ex. autre *thread* [1]).

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

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 ...`, ...

Fonctions de première classe et fonctions d'ordre supérieur

Plus d'exemples (1)

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.

Fonctions de première classe et fonctions d'ordre supérieur

Plus d'exemples (2)

Évidemment, plus intéressant d'écrire de nouveaux blocs de contrôle, par exemple :

Bloc `retry` :

```
// pareil, ne faites pas ça à  la maison !
public static void retry(??? instructions, int tries) {
    while (tries > 0) {
        try { instructions(); return; }
        catch (Throwable t) { tries--; }
    }
    throw new RuntimeException("Failure persisted after all tries.");
}
```

Fonctions de première classe et fonctions d'ordre supérieur

Plus d'exemples (3)

Encore un exemple : [1]

```
// toujours en syntaxe fantaisiste --- SURTOUT NE PAS RECOPIER OU MÊME RETENIR !
public static <U, V> List<V> map(List<U> l, ??? f) {
    List<V> ret = new ArrayList<>();
    for (U x : l) ret.add(f(x));
    return ret;
}
```

Ou encore : [2]

```
public static readPacket(Socket s, ??? callback) {
    ...
}
```

`callback` = FPC pour traiter le prochain paquet reçu = **fonction de rappel**/***callback***.

1. L'API `java.util.stream` contient plein de méthodes de traitement par lot dans ce genre.
2. Lecture asynchrone, similaire à ce qu'on trouve dans l'API `java.nio`.

Programmation fonctionnelle

Concepts de FPC et de FOS essentiels pour la **programmation fonctionnelle** (PF) :

- PF = paradigme de programmation, au même titre que la POO.
- **Idée de base :** on conçoit un programme comme une fonction mathématique, elle-même obtenue par composition d'un certain nombre d'autres fonctions.
- Or pour pouvoir composer les fonctions, il faut supporter les FOS et donc les FPC.
- Langages fonctionnels connus : Lisp (et variantes : Scheme, Emacs Lisp, Clojure...), ML (et variantes : OCaml, F#...), Haskell, Erlang...
- Rien n'empêche d'être à la fois objet et fonctionnel (Javascript, Scala, OCaml, Common Lisp ...). Les langages sont souvent multi-paradigme (avec préférence).

Java ($\geq$ 8) possède quelques concepts fonctionnels [1].

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.

Que faut-il pour qu'un langage supporte les FPC ?

Principalement 3 choses :

- si langage à typage statique, un système de types permettant d'écrire les types des FPC
- une syntaxe adaptée pour les expressions décrivant les FPC.
  Notamment, il faut des littéraux fonctionnels (appelés aussi **lambda-expressions** [1] ou encore fonctions anonymes).
- une représentation en mémoire adaptée pour les FPC
  (en Java, forcément des objets particuliers)

1. En particulier en Java. C'est donc le nom « lambda-expression » que nous allons utiliser.
Pourquoi « lambda » ? Référence au lambda-calcul d'Alonzo Church : la fonction $x \mapsto f(x)$ s'y écrit $\lambda x.f(x)$.

FPC avant Java 8

Analyse de l'existant

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

En Java (toute version) on implémente et instancie 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 (décrite par la méthod `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`).

FPC avant Java 8

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.
- **syntaxe :** lourde et peu pratique.
- **représentation en mémoire :** instance d'une classe contenant juste une méthode.

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.
  Sinon, rien n'est changé au système de type de Java (même sa syntaxe).
- **syntaxe :** lourde et peu pratique.
- **représentation en mémoire :** instance d'une classe contenant juste une méthode.

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.
  Sinon, 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 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.
  Sinon, 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.

Interfaces fonctionnelles et types SAM

- **Interface fonctionnelle** = interface avec 1 seule méthode abstraite [1]
- **type SAM** (*single abstract method*) : type défini par une interface fonctionnelle.
- En fait, il en existait déjà plein avant Java 8 (ex. : interfaces `Comparable`, `Comparator`, `Runnable`, `Callable`, `ActionListener`...).
- L'API `java.util.function` en ajoute quelques dizaines, afin de standardiser les types des FPC. Cf. les 2 pages suivantes.
- 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).
- 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 nominal.

1. ... mais autant de méthodes **`static`**, **`default`** ou **`private`** que l'on souhaite !

Catalogue des interfaces fonctionnelles de Java 8

dans le package `java.util.function` (1)

`java.util.function` contient toute une série d'interfaces fonctionnelles standard.

Interfaces génériques :

Interface	Type représenté	Méthode unique
`BiConsumer<T,U>`	$T \times U \to \{()\}$	`void accept(T, U)`
`BiFunction<T,U,R>`	$T \times U \to R$	`R apply(T, U)`
`BinaryOperator<T>`	$T \times T \to T$	`T apply(T, T)`
`BiPredicate<T,U>`	$T \times U \to \{\bot, \top\}$	`boolean test(T, U)`
`Consumer<T>`	$T \to \{()\}$	`void accept(T)`
`Function<T,R>`	$T \to R$	`R apply(T)`
`Predicate<T>`	$T \to \{\bot, \top\}$	`boolean test(T)`
`Supplier<T>`	$\{()\} \to T$	`T get()`
`UnaryOperator<T>`	$T \to T$	`T 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).

Catalogue des interfaces fonctionnelles de Java 8

dans le package `java.util.function` (2)

Interfaces spécialisées :

Interface	Type représenté	Méthode unique
`BooleanSupplier`	$\{()\} \to \{\bot, \top\}$	`boolean getAsBoolean()`
`DoubleBinaryOperator`	$\mathbb{R} \times \mathbb{R} \to \mathbb{R}$	`double applyAsDouble(double, d`
`DoubleConsumer`	$\mathbb{R} \to \{()\}$	`void accept(double)`
`DoubleFunction<R>`	$\mathbb{R} \to R$	`R apply(double)`
`DoublePredicate`	$\mathbb{R} \to \{\bot, \top\}$	`void test(double)`
`DoubleSupplier`	$\{()\} \to \mathbb{R}$	`double getAsDouble()`
`DoubleToIntFunction`	$\mathbb{R} \to \mathbb{Z}$	`int applyAsInt(double)`
...	...	...

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

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

1. Moins d'allocations et d'indirections.

Catalogue des interfaces fonctionnelles de Java 8

dans le package `java.util.function`... mais pas seulement !

Attention, catalogue incomplet :

- L'interface `java.lang.Runnable` reste le standard pour les « fonctions »[1] de $\{()\} \to \{()\}$ (**`void`** vers **`void`**).
- Pas d'interfaces standard pour les fonctions à plus de 2 paramètres $\to$ il faut définir les interfaces soi-même :

```
@FunctionalInterface public interface TriConsumer<T, U, V> {
    void apply(T t, U u, V v);
}
```

L'annotation facultative `@FunctionalInterface` demande au compilateur de signaler une erreur si ce qui suit n'est pas une définition d'interface fonctionnelle.

1. Ces fonctions sont intéressantes pour leurs effets de bord et non pour la transformation qu'elles représentent. En effet, en mathématiques, $card(\{()\} \to \{()\}) = 1$.

Retour sur les exemples...

... avec les bons types et la bonne syntaxe


```java
public static void repeat5(Runnable f) { for (int i = 0; i < 5; i++) f.run(); }
```

```java
public static void ifElse(BooleanSupplier cond, Runnable ifBlock, Runnable elseBlock) {
    if (cond.getAsBoolean()) ifBlock.run();
    else elseBlock.run();
}
```

```java
public static void retry(Runnable instructions, int tries) {
    while (tries > 0) {
        try { instructions.run(); return; }
        catch (Throwable t) { tries--; }
    }
    throw new RuntimeException("Failure persisted after all tries.");
}
```

```java
public static <U, V> List<V> map(List<U> l, Function<U,V> f) {
    List<V> ret = new ArrayList<>();
    for (U x : l) ret.add(f.apply(x));
    return ret;
}
```

Interfaces fonctionnelles

Écrire les types des FPC, c'est bien. Mais comment exécuter une fonction ?

Comme on a déjà pu voir sur les exemples :

- Il n'y a pas de syntaxe réservée pour exécuter une expression fonctionnelle.
- Il faut donc à chaque fois appeler explicitement la méthode de l'interface fonctionnelle concernée (et donc connaître son nom...).

Exemple :

```
Function<Integer, Integer> carre = n -> n * n;
System.out.println(carre.apply(5)); // <--- ici c'est apply
```

mais...

```
Predicate<Integer> estPair = n -> (n % 2) == 0;
System.out.println(estPair.test(5)); // <--- là  c'est test
```

Syntaxe des lambda-expressions

lambda-abstraction : par l'exemple

Écrire une fonction anonyme par lambda-abstraction :

```
<paramètres> -> <corps de la fonction>
```

Exemples :

- ```
  x -> x + 2
  ```

  (raccourci pour `(int x)-> { return x + 2; }`)
- ```
  (x, y) -> x + y
  ```

  (raccourci pour `(int x, int y)-> { return x + y; }`)
- ```
  (a, b) -> {
    int q = 0;
    while (a >= b) { q++; a -= b; }
    return q;
  }
  ```
  ```

Syntaxe des lambda-expressions

lambda-abstraction : syntaxe formelle


```
<paramètres> -> <corps de la fonction>
```

Syntaxe en détails :

- <paramètres> : liste de paramètres formels (de 0 à plusieurs), de la forme `(int x, int y, String s)`
  Mais juste `(x,y,s)` fonctionne aussi (type des paramètres inféré).
  Et parenthèses facultatives quand il y a un seul paramètre.
  Il est aussi possible (Java $\geq$ 11) de remplacer les noms de types par **`var`**.
- <corps de la fonction>, au choix :
  - une simple expression, p. ex. `(x==y)?s:""`
  - une liste d'instructions entre accolades, contenant une instruction **`return`** si type de retour non **`void`**

Syntaxe des lambda-expressions

Référence de méthode

Pour créer une lambda-expression contenant juste l'appel d'une méthode existante :

- on peut utiliser la lambda-abstraction :
  `x -> Math.sqrt(x)`
- mais il existe une notation encore plus compacte :
  `Math::sqrt`
  Ceci s'appelle une **référence de méthode**

Remarque :
`Math::sqrt` est bien équivalent à `x -> Math.sqrt(x)`, et non à `(double x)-> Math.sqrt(x)`. Cela a une incidence pour l'inférence de type (cf. la suite).

Syntaxe des lambda-expressions

Référence de méthode : les cas de figure

Supposons la classe suivante définie :

```
class C {
    int val;
    C(int val) { this.val = val; }
    static int f(int n) { return n; }
    int g(int n) { return val + n; }
}
```

La notation « référence de méthode » se décline pour différents cas de figure :

- Méthode statique → `C::f` pour `n -> C.f(n)`
- Méthode d'instance avec récepteur donné → avec `x = new C()`, on écrit `x::g` pour `n -> x.g(n)`
- Méthode d'instance sans récepteur donné → `C::g` pour `(x, n)-> x.g(n)`
- Constructeur → `C::new` pour `n -> new C(n)`

En cas de surcharge, Java déduit la méthode référencée du type attendu.

Utiliser les méthodes (FOS) de nos exemples...

... en leur passant des lambda-expressions


```
// Dire 5 fois "Bonjour !" :
repeat5(() -> { System.out.println("Bonjour !"); });
```

```
// Tirer pile ou face
ifElse( () -> Math.random() > 0.5,
    () -> { System.out.println("pile"); },
    () -> { System.out.println("face"); }
);
```

```
// Essayer d'ouvrir un fichier jusqu'à  3 fois
retry(() -> { ouvre("monFichier.txt"); }, 3);
```

```
// Calculer les racines carrées des nombres d'une liste
List<Double> racines = map(maListe, Math::sqrt);
```

Ce qui se cache derrière les lambda-expressions

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 ?

Ce qui se cache derrière les lambda-expressions

Concernant le typage (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. [1]

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 t); }  // apply a une signature compatible
```

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

Ce qui se cache derrière les lambda-expressions

Concernant le typage (2) — petits pièges

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

Ces exemples compilent :

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

Mais ceux-ci ne compilent 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 typage (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.invokeLater(()-> MonIG.build());`

ou encore mieux : `SwingUtilities.invokeLater(MonIG::build);`

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 [1] 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.

Avertissement

Attention aux variables locales !

- 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`**.[1]
- **Comparaison avec OCaml :** en OCaml, cette « limitation » n'est pas perçue car, en effet, les « variables » ne sont pas réaffectables[2] (tout est « **`final`** »).

  Comme Java, OCaml se contente de recopier les valeurs des variables locales dans la clôture 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 ».

Avertissement

Exemple :

Incorrect :

```
int a = 1;
a++; // a réaffectée
Function<Integer, Integer> f = x -> { return x + a; };
```

Correct :

```
final int a = 1; // a final (non réaffectable)
Function<Integer, Integer> f = x -> { return x + a; };
```

Aussi correct :

```
int a = 1; // a effectivement final (non réaffectée)
Function<Integer, Integer> f = x -> { return x + a; };
```

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

- 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 $\Longrightarrow$ 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.

Fonctions récursives

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.

Fonctions récursives

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 = Autre::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 [1]. C'est donc celle qu'il faut privilégier.

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

Bilan des FPC dans Java 8

Le mauvais

- 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. [1]
  ($\Rightarrow$ Impossible de définir simplement une lambda-expression récursive [2].)
- 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 bon

- 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 [1], qui devient de plus en plus incontournable [2].
- 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 lambda-expressions car utilisant déjà des interfaces fonctionnelles (e.g. JavaFX).
  $\rightarrow$ 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 cœurs dans les microprocesseurs.

Opérations d'agrégation

Définition et quelques exemples

Opération d'agrégation : traitement d'une séquence de données de même type qui produit un résultat synthétique [1] 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 $\rightarrow$ 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 (1)

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

Quelques exemples (2)

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 [1] :

```
public static <E> 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

Généralisation

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;
}
```

Opérations d'agrégation

Généralisation

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;
}
```

Opérations d'agrégation

Généralisation

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));
}
```

Opérations d'agrégation

Généralisation

Pour écrire des traitements similaires à ces exemples, on aimerait une API fournissant des FOS analogues à `fold` pour les principaux schémas d'itération [1].

C'est justement ce que fait l'API *stream*. [2]

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

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`[1]) ;
- 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-thread*ée).

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. Heureusement on obtient facilement une instance de de `Stream` depuis une instance de `Collection`, grâce à la méthode `stream` de `Collection`.

Opérations d'agrégation

Avec les *streams*

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

1. sélection d'une **source** d'éléments [1]. Depuis cette source on obtient un ***stream***.
2. un certain nombre (0 ou plus) d'**opérations intermédiaires**. Ces opérations transforment un *stream* en un autre *stream*.
3. 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)
```

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

Les objets *stream*

Caractéristiques principales

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

1. la représentation implicite d'une séquence d'éléments finie ou infinie
2. 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** [1].

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

Les objets *stream*

`Stream` versus `Iterator`

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* [1]).
- *iterators* : 1 opération (`next`) = lire l'élément suivant ($\rightarrow$ itération explicite avec **`for`** ou **`while`**)

1. Le *stream* construit sur une collection utilise en fait le *spliterator* 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*

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

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

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

```
bibli.getLivres() //                                                        List<Livre>
    .stream() //                                                            Stream<Livre>
    .filter(livre -> livre.getAuteur().equals("Dumas, Alexandre")) //       Stream<Livre>
    .flatMap(livre -> livre.getEmprunteurs().stream()) //                   Stream<Usager>
    .distinct() //                                                          Stream<Usager>
    .count(); //                                                            long
```

Streams et parallélisme

Problème à résoudre

- la plupart des collections ne sont pas *thread safe* (comportement incorrect quand utilisées dans plusieurs *threads* en même temps, notamment à cause des accès en compétition)
- on peut y ajouter de la synchronisation (voir collections synchronisées), mais toujours risque de *dead-lock*.

Pourtant, accélérer le traitement les grandes collections, il est utile de profiter du parallélisme.

Streams et parallélisme

Problème à résoudre... une solution ?

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'aggrégation 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 [1], 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` [2].

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.

Streams et parallélisme

Quelques explications

- 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(...))`.

Streams et parallélisme

Piège à éviter : les effets de bord

En général, évitez les **effets de bord** [1] dans le *pipeline*, préférez les **fonctions pures** [2].

- Pour les entrées/sorties, au mieux, pas de contrôle sur leur ordre.
- Pour les modifications d'objets partagés :
  - sans synchronisation, risque d'accès en compétition. Or, dans ce cas, le modèle de concurrence de Java ne garantit rien ($\rightarrow$ résultats incorrects).
  - avec synchronisation : comme on ne contrôle pas l'ordre d'exécution des tâches du *pipeline*, risque de *dead lock*.
    Sinon, de toute façon, la synchronisation ralentit l'exécution.

Heureusement, habituellement [3], il est inutile de modifier des objets extérieurs dans les opérations d'un *stream*.

1. Effet de bord : tout effet externe d'une fonction, c'est à dire toute sortie physique ou modification de mémoire en dehors de son espace propre (= variables locales + champs des objets non partagés).
2. Fonction pure : fonction (méthode ou FPC) sans effet de bord.
3. À part à des fins de déboguage ou de *monitoring*.

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.

L'interface Stream

L'interface principale du package

```
public interface Stream<T> { // pour des éléments de type T
  ...
}
```

(Il existe aussi `DoubleStream`, `IntStream` et `LongStream`.)

Cette interface contient un grand nombre de méthodes. 3 catégories :

- des méthodes statiques [1] 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

- Depuis une collection : méthode `Stream<T> stream()` de `Collection<T>`.
- À l'aide d'une des méthodes statiques de `Stream` :
  - `<T> Stream<T> empty()` : retourne un *stream* vide
  - `<T> Stream<T> generate(Supplier<T> s)` : retourne la séquence des éléments générés par `s.get()` (Rappel : `Supplier<T>` = fonction de $\{()\} \to T$.).
  - `<T> Stream<T> iterate(T seed, UnaryOperator<T> f)` : retourne la séquence des éléments `seed`, `f.apply(seed)`, `f.apply(f.apply(seed))` ...
  - `<T> Stream<T> of(T... values)` : retourne le *stream* constitué de la liste des éléments passés en argument (méthode d'arité variable).
- En utilisant un *builder* [1] (`Stream.Builder`) :
  - Un `Stream.Builder` est un objet mutable servant à construire un *stream*.
  - On instancie un *builder* vide avec l'appel statique `b = Stream.builder()`
  - On ajoute des éléments avec les appels `b.add(T e)` ou `b.accept(T e)`.
  - On finalise en créant le *stream* contenant ces éléments : appel `s = b.build()`

1. On parle du patron de conception *builder* (ou "monteur"), ici appliqué aux *streams*. Ainsi, par exemple, il existe une classe `StringBuilder` jouant le même rôle pour les `String`. Voir le TP sur le patron *builder*.

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

Pour **`this`** instance de `Stream<T>` :

- ```
  Stream<T> distinct()
  ```
  retourne un *stream* qui parcourt les éléments de **`this`** sans les doublons.
- ```
  Stream<T> filter(Predicate<? super T> p)
  ```
  retourne le *stream* parcourant les éléments `x` de **`this`** qui satisfont `p.test(x)`
- ```
  <R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)
  ```
  retourne la concaténation des *streams* `mapper.apply(x)` pour tout `x` dans **`this`**.
- ```
  Stream<T> limit(long n)
  ```
  tronque le *stream* après `n` éléments.
- ```
  <U> Stream<U> map(Function<T, U> f)
  ```
  retourne le *stream* des éléments `f.apply(x)` pour tout `x` élément de **`this`**.
  ```

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

- ```
  Stream<T> peek(Consumer<? super T> c)
  ```

  retourne un *stream* avec les mêmes éléments que **`this`**. À l'étape terminale, pour chaque élément `x` parcouru, `c.consume(x)` sera exécuté [1].

- ```
  Stream<T> skip(long n)
  ```

  retourne le suffixe de la séquence en sautant les `n` premiers éléments.

- ```
  Stream<T> sorted()
  ```

  retourne la séquence, triée dans l'ordre naturel.

- ```
  Stream<T> sorted(Comparator<? super T> comparator)
  ```

  idem mais en suivant l'ordre fourni.

1. Remarque : `c` ne sert que pour ses effets de bord. `peek` peut notamment être utile pour le déboguage.

Calculer et extraire un résultat : les opérations terminales

Opérations spécialisées (1)

- `boolean allMatch(Predicate<? super T> p)` retourne vrai si et seulement si `p` est vrai pour tous les éléments du *stream*.
- `boolean anyMatch(Predicate<? super T> p)` retourne vrai si et seulement si `p` est vrai pour au moins un élément du *stream*.
- `long count()` retourne le nombre d'éléments dans le *stream*.
- `Optional<T> findAny()` : retourne un élément (quelconque) du *stream* ou rien si *stream* vide (voir interface `Optional<T>`).
- `Optional<T> findFirst()` : pareil, mais premier élément.

Calculer et extraire un résultat : les opérations terminales

Opérations spécialisées (2)

- `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.

Calculer et extraire un résultat : les opérations terminales

Opérations “couteau suisse” :

- `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. [1]
- `<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...

L'interface `Collector`

interface `Collector<T,A,R>`

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`

La classe `Collectors`

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(...)`...

$\rightarrow$ on retrouve des opérations équivalentes [1] 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`

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(
    ArrayList<Integer>::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<VoitureSansPermis> listVoitureSP = new ArrayList<>();

  // l1: ceci est en fait interdit... mais supposons que ça passe...
  List<Voiture> listVoiture = listVoitureSP;

  // 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**.

Types génériques et sous-typage

Invariance des génériques

type erasure → vérification **à la compilation seulement**. Court-circuitons-la, pour voir :

```
// l1': version avec "triche" (pas d'exception car type erasure)
List<Voiture> listVoiture = (List<Voiture>)(Object) listVoitureSP;

/* l2: */ listVoiture.add(new Voiture());

// l3: ça affiche effectivement "Voiture" (oooh !)
System.out.println(listVoitureSP.get(0).getClass());

// l4: et pour la forme, une petite ClassCastException :
VoitureSansPermis vsp = listVoitureSP.get(0);
```

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 outre (*cast*), il nous avertit (à raison car `ClassCastException` peut se produire à l'exécution).

Tableaux – génériques ou pas ? (1)

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

```
public static void main(String[] args) {
  VoitureSansPermis[] listVoitureSP = new VoitureSansPermis[100];

  // l1: ceci est autorisé !
  Voiture[] listVoiture = listVoitureSP;

  // 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) : $[A <: B] \implies [A[] <: B[]]$.

Mais on crashe plus loin, lors de l'exécution de l2 (bien qu'il n'y ait pas eu de *warning* !).

Tableaux – génériques ou pas ? (2)

- covariance à la place d'invariance : ⇒ vérifications moins strictes à la compilation, rendant possibles des problèmes à l'exécution [1]
- 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 (1)

- 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).

Génériques et tableaux

Différence de philosophie (2)

- 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 $T \rightarrow$ exécution cohérente et sans exception garantie.
  - **usage anormal,** conversion forcée : `(Gen<B>)(Object)(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 ?

Tableaux et génériques ne font pas bon ménage : les uns ont besoin de tout savoir à l'exécution, alors que les autres veulent tout oublier !

- Avec `T`, paramètre de type, **`new T[taille]` est impossible**.
  Raison : pour instancier un tableau, Java doit connaître dès la compilation le type concret des éléments du tableau.
  Or à la compilation, `T` n'est pas associé à un type concret.
- Les types tableau de types paramétrés, comme ~~`List<Integer>[]`~~, **sont illégaux**.
  Raison : à l'exécution, Java ne sait pas distinguer `List<Integer>` et `List<String>` et donc ne peut pas accepter de mettre des `List<Integer>` dans un tableau sans aussi accepter `List<String>`.
  $\Longrightarrow$ 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.

Génériques et tableaux

Premier interdit : ~~`new T[10]`~~

Supposons `T extends Up` paramètre de type.

~~`new T[10]`~~ est aussi interdit.

Raison : après compilation, `T` est oublié et remplacé par `Up`. Au mieux `new T[10]` pourrait être compilé comme `new Up[10]`. Mais si c'était ce qui se passait, on pourrait trop facilement « polluer » la mémoire sans s'en rendre compte :

```
static <T> T[] makeArray(int size) { return new T[10]; /* interdit ! */ }
static {
    String[] tString = makeArray(10); // à  l'exécution on affecterait un Object[]
    Object[] tObject = tString; // toujours autorisé (covariance)
    tObject[0] = 42; // et BOOM ! Maintenant tString contient un Integer !
}
```

En pratique, pour faire compiler cela, il faut « tricher » avec *cast* explicite :
`T[] tab = (T[])new Up[10];`.
Ça n'empêche pas le problème ci-dessus, mais au moins le compilateur affiche un *warning* (« `unchecked conversion` »).

Génériques et tableaux

Deuxième interdit : ~~Gen<T>[]~~ (1)

Mauvais scenario, pouvant se produire si on autorisait les tableaux de génériques :

```
class Box<T> {
    final T x;
    Box(T x) { this.x = x; }
}

class Loophole {
    public static void main(String[] args) {
        Box<String>[] bsa = new Box<String>[3];     // supposons que cette ligne compile
        Object[] oa = bsa;                          // autorisé car tableaux covariants

        /* autorisé à  la compilation (Box < 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<Integer>(3);

        /* ... et là , c'est le drame !
        (ClassCastException, alors que l'instruction est bien typée !) */
        String s = bsa[0].x;
    }
}
```

Génériques et tableaux

Deuxième interdit : ~~Gen<T>[]~~ (2)

~~`new Box<Integer>[10]`~~ est interdit.

En effet, le tableau instancié serait de type `Box[]` [1], ce qui rendrait possible le scénario du transparent précédent.

En revanche, Java autorise `new Box[10]`.

Remarque, du coup, là aussi, il existe une « triche » pour faire compiler l'exemple du transparent précédent : on remplace `new Box<String>[3]` par `new Box[3]`.
Le compilateur émet heureusement un warning (« unchecked conversion »)... et à l'exécution, on a effectivement `ClassCastException`

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 !

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

Les génériques invariants c'est bien mais...

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

Le besoin : quand $B <: A$ [1], 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 (`ArrayStoreException` possible même dans programme sans *warning*).

1. Ou bien, peut.-être parfois, quand $B <: A$.

Considérations autour de la variance (1)

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 [1], 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,A> 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

Considérations autour de la variance (2)

Réponse : affecter `u` à `v` a un sens si la méthode `apply` de `u` peut remplacer celle de `v` (en toute situation). C.-à-d. :

- si elle accepte tous les paramètres effectifs acceptés par celle-ci
- et si les valeurs retournées appartiennent à un type au moins aussi restreint.

(En résumé : une instance de `Function<X,Y>` peut remplacer une instance de `Function<Z,T>` si `X :> Z` et `Y <: T`.)

$\rightarrow$ en appliquant ce principe, on voudrait donc que le compilateur accepte :

```
z = t; z = x; t = y; x = y; z = y;
```

Considérations autour de la variance (3)

Représentation graphique : type générique → pièce de puzzle.

- Paramètre utilisé en entrée (= type de paramètre de méthode ou type d'attribut public modifiable) → encoche.
- 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.

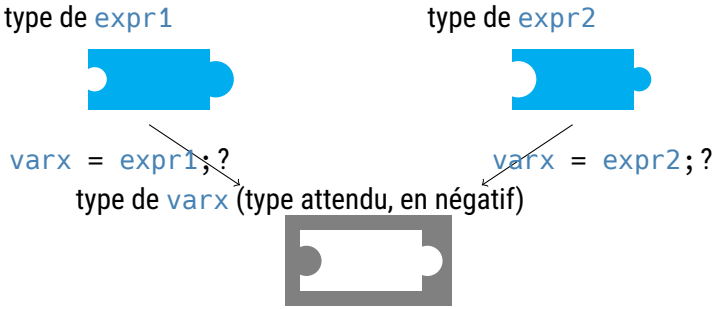
Exemple :

`Function<T, U>`

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

Considérations autour de la variance (4)

Ainsi, inclusion des formes si et seulement s'il y a sous-typage :



→ seul `varx = expr2;` doit fonctionner [1] (pas de chevauchement) :

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

Considérations autour de la variance (5)

La variance souhaitée n'est donc pas la même pour tous les types génériques :

- intuitif et logique de vouloir
  `Function<Object, Integer>` <: `Function<Double, Number>`.
  Justification : le premier paramètre de type est utilisé uniquement pour l'argument de `apply` alors que l'autre est uniquement son type de retour.
  → Emboîtement de `Fuction<T, U>` dans `Function<V,W>` possible dès que `T` :> `V` et `U` <: `W`.

Remarque : tailles de l'encoche et de l'excroissance de `Fuction<T, U>` 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).

Considérations autour de la variance (6)

- mais ~~`List<Integer>`~~ $\not<:$ ~~`List<Number>`~~ serait illogique.

  Justification : Le même paramètre apparait à la fois en sortie (méthode `get`) et en entrée (méthodes `set` et `add`) [1]. Donc tailles de l'encoche et de l'excroissance de `List<T>` liées car représentant le même `T`
  $\rightarrow$ impossible d'encastrer la pièce de `List<X>` dans le trou `List<Y>` si $X \neq Y$.

1. Même topo pour `Integer[]` $<:$ `Number[]` avec les opérations `x = t[i]` et `t[i] = x` (... mais ça c'est autorisé : en contrepartie, il est nécessaire de faire des vérifications à l'exécution, avec risque de `ArrayStoreException`).

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

Dans `Function<T, U>`, `T` et `U` ont des **influences contraires** l'une de l'autre à cause de leur usage dans la méthode de `Function`.

→ 2 catégories d'usage :

- en **position covariante** : utilisé comme type de retour de méthode (ou type d'attribut)
- en **position contravariante** : utilisé comme type d'un argument dans la signature d'une méthode (ou comme type d'un attribut non `final`)

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

→ 3 catégories de paramètres de type :

- **paramètre covariant** (comme U) : utilisé seulement en position covariante
  → plus le paramètre effectif est petit, plus le type paramétré devrait être petit ;
- **paramètre contravariant** (comme T) : utilisé seulement en position contravariante
  → plus le paramètre effectif est petit, plus le type paramétré devrait être grand ;
- **paramètre invariant** : utilisé à la fois en position covariante et contravariante.

Attention, ces concepts ne sont que théoriques.

Il se trouve que ceux-ci n'ont **pas de sens pour le compilateur de Java** : rappelez-vous qu'on avait dit que, pour l'instant, on oubliait l'invariance imposée par Java.

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>`.[1]

- **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).

Wildcards

principe de base (1)

(On revient enfin à Java !)

- 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 « > »).

Wildcards

principe de base (2)

- **Exemple de déclaration de méthode :**

  ```
  static double somme(List<? extends Number> liste){ ... }
  ```

  Cette méthode annonce pouvoir faire la somme des éléments d'une liste de n'importe quoi, tant que ce n'importe quoi est un sous-type de nombre.

  Dans ce cas, c'est équivalent à :

  ```
  static <T> double somme(List<T extends Number> liste) { ... }
  ```

- **Exemple de déclarations de variables :**

  ```
  C<? extends A> v1;
  C<? super B> v2;
  ```

  on peut alors affecter à `v1` (resp. `v2`) toute valeur de type `C<X>` pour peu que `X` soit sous type (resp. supertype) de `A` (resp. `B`).

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]

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 $A \cup B$ (qui n'est pas un type de Java). Ainsi `T<? super C>` serait équivalent à `<? super A U B>` (syntaxe fictive).
Sinon, 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

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.

Généralisons :

- Soit une expression `expr` utilisées dans un certain contexte (appel de méthode, affectation, ...).
- Soit `TE` le type (déjà « converti par capture »[1], 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>`.

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

Wildcards

utilisation pour signaler la variance

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

Conversion par capture (1)

En réalité, pour une expression, avoir le type `Gen<?>` veut dire qu'**il existe** [1] un type `QuelqueChose` (inconnu mais fixé) tel que cette expression est de type `Gen<QuelqueChose>`.

Il faut interpréter le type d'une expression à *wildcards* comme un type inconnu appartenant à l'ensemble des types respectant les contraintes trouvées.

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`.

Wildcards

Conversion par capture (2)

Concrètement, lors de la vérification de type d'une expression, le compilateur effectue une opération appelée **conversion par capture** [1] :

- À chaque fois qu'un « ? » apparaît au premier niveau [2] du type d'une expression [3], le compilateur le remplace par un nouveau type créé à la volée, recevant un nom de la forme `capture#i-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<? extends Set<?>>` devient `List<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.

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.

Wildcards

Exemple pathologique

```
List<? super String> l = new ArrayList<>(); l.add("toto"); //ok
l.add(l.get(0)); // Mal typé ! Mais pourquoi ?
```

Explication : à la 2e ligne,

- la 1e occurrence de `l` est de type `List<capture#1-of?>` $\Longrightarrow$ `l.add(...)` attend un paramètre de type `capture#1-of?`;
- la 2e occurrence de `l` est de type `List<capture#2-of?>` (capture indépendante !) $\Longrightarrow$ `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<>(); l.add("toto"); aux(l); // encore ok
```

Wildcards

Quid des exemples plus complexes ? (1)

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 élémént (resp. un sous-ensemble) de l'ensemble attendu.

Wildcards

Quid des exemples plus complexes (2) ?

```
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>>` (type reçu = celui de `las`) et `<? extends A<? super Integer>>` (type attendu = celui de `lar`).
- Au premier niveau, on a `List` et `List` $\rightarrow$ OK, vérifions les paramètres.
- Il faut que `A<? super String>` $<:$ `A<? super Integer>`.
- On a `A` des 2 côtés... jusque là 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 !