

# LES EXPRESSIONS LAMBDA

Voir les deux liens suivants :

<http://www.lambdafaq.org/why-are-lambda-expressions-being-added-to-java/>

## **I. Justification et historique**

### **a) Les classes anonymes**

Il existe en Java des classes dites anonymes, utilisées pour fournir un moyen d'implémenter des classes exploitables une seule fois dans une application. L'exemple typique est celui des gestionnaires d'événements graphiques. Plutôt que de construire des classes séparées pour chacun des clics sur un bouton, il est possible d'écrire :

```
Button button = new Button("Test Button");
button.setAction(new EventHandler() {
    @Override
    public void handle(ActionEvent actionEvent) {
        //... traitement du bouton
    }
});
```

Si cette technique n'existait pas, il faudrait construire une classe implémentant EventHandler pour chaque événement rattaché par exemple à un bouton. En créant la classe là où elle est nécessaire, rend le code un peu plus facile à lire, même si celui-ci n'est pas particulièrement élégant (beaucoup d'instructions pour définir une seule méthode).

### **b) Les interfaces fonctionnelles**

Le code de l'interface EventHandler est le suivant :

```
public interface EventHandler<T extends Event> extends EventListener {

    public void handle(T event);

}
```

Il s'agit d'une interface composée d'une seule méthode : une telle interface est connue comme étant une interface fonctionnelle ("functional interface"). Et utiliser des interfaces fonctionnelles est un pattern très fréquent en Java, il suffit par exemple de penser à des interfaces comme `java.lang.Runnable` ou encore `Comparator`. Java a permis de les améliorer et de les utiliser avec des expressions lambda, de manière à en simplifier l'utilisation.

En effet, chaque expression lambda peut être implicitement affectable à une interface appelée interface fonctionnelle ("functional interface"). Par exemple, il est possible de créer la référence d'une interface fonctionnelle au moyen d'une expression fonctionnelle par :

```
Runnable r = () -> System.out.println("hello world");
```

Ce type de conversion est automatiquement pris en charge par le compilateur lorsque l'on écrit:

```
new Thread(() -> System.out.println("hello world")).start();
```

Dans le code ci-dessus, le compilateur déduit automatiquement que l'expression lambda peut être convertie en une interface Runnable, grâce à l'entête du constructeur :

```
public Thread(Runnable r) { }.
```

Voici d'autres exemples d'expressions lambda et de leurs interfaces fonctionnelles :

```
Consumer<Integer> c = (int x) -> { System.out.println(x) };
```

```
BiConsumer<Integer, String> b =  
    (Integer x, String y) -> System.out.println(x + " : " + y);
```

```
Predicate<String> p = (String s) -> { s == null };
```

<b>Les principales interfaces fonctionnelles (java.util.function)</b>
---

<b>Interfaces fonctionnelles</b>	<b>Types des paramètres</b>	<b>Type de la valeur de retour</b>	<b>Méthodes abstraites</b>	<b>Description</b>	<b>Autres méthodes</b>
Supplier<T>	Aucun	T	get	Renvoie une valeur de type T	
Runnable	Aucun	void	run	Exécute une action sans paramètre ni valeur de retour	
Consumer<T>	T	void	accept	Consomme une valeur de type T	chain
BiConsumer<T,U>	T, U	void	accept	Consomme deux valeurs de type T et U	chain
Function<T,R>	T	R	apply	Une fonction avec un argument de type T	compose, andThen, identity
BiFunction<T,U,R>	T, U	R	apply	Une fonction avec deux arguments de type T et U	andThen
UnaryOperator<T>	T	T	apply	Un opérateur unaire à appliquer au type T	compose, andThen, identity
BinaryOperator<T>	T, T	T	apply	Un opérateur binaire à appliquer au type T	andThen
Predicate<T>	T	boolean	test	Une fonction retournant un booléen	and, or, negate, isEqual
BiPredicate<T>	T, U	boolean	test	Une fonction retournant un booléen avec deux paramètres	and, or, negate

### c) Création d'une interface fonctionnelle

L'annotation [`@FunctionalInterface`](#) est une nouvelle annotation utilisée pour indiquer que la déclaration d'un type d'interface doit être comprise comme étant celle d'une interface fonctionnelle. Java 8 fournit par ailleurs un certain nombre d'interfaces fonctionnelles pouvant être utilisées par des expressions lambda.

L'annotation `@FunctionalInterface` peut être utilisée comme marqueur d'erreur par le compilateur quand l'interface annotée n'est pas une interface fonctionnelle valide. Par exemple, l'interface fonctionnelle suivante est valide :

```
@FunctionalInterface
public interface WorkerInterface {

    public void f();

}
```

Et celle-ci ne l'est pas, et produit l'erreur indiquée à la suite :

```
@FunctionalInterface
public interface WorkerInterface {

    public void f();

    public void g();

}
Error:
Unexpected @FunctionalInterface annotation
@FunctionalInterface ^ WorkerInterface is not a functional interface multiple
non-overriding abstract methods found in interface WorkerInterface 1 error
```

Une fois l'interface fonctionnelle définie, il est possible de l'utiliser n'importe où dans le code. Par exemple :

```
//define a functional interfDéfinition d'une interface fonctionnelle
@FunctionalInterface
public interface WorkerInterface {

    public void traitement();

}
```

```

public class WorkerInterfaceTest {

    public static void execute(WorkerInterface worker) {
        worker.doSomeWork();
    }

    public static void main(String [] args) {
        //invocation par une classe anonyme
        execute(new WorkerInterface() {
            @Override
            public void traitement () {
                System.out.println("Utilisation d'une classe anonyme");
            }
        });

        //invocation par une expression lambda
        execute( () -> System.out.println("Utilisation d'une expression Lambda") );
    }
}

```

#### **d) Invocation d'une expression lambda**

On pourrait légitimement s'attendre à une notation ressemblant à :

```

{ int x -> x + 1 }.invoke(10)
ou encore
int sum = { int x, int y -> x + y }.invoke(3, 4);

```

Ce n'est absolument pas le cas de Java 8; en effet, le langage préfère travailler à l'aide des interfaces fonctionnelles. Demander à exécuter une lambda signifie "instancier une interface fonctionnelle".

```

interface Runnable { void run(); }

Runnable r = () -> { System.out.println("hello"); };
Thread t = new Thread (r);
t.start();

```

## II. Les expressions lambda

### a) Syntaxe

La syntaxe d'une lambda est une expression formée de trois parties :

Liste d'arguments encadrée par des ()	Flèche ->	Corps de l'expression : instructions
--	--------------	---

- La première partie donne la liste des paramètres utilisés par l'expression lambda (comme ceux d'une fonction classique); par exemple (int x, int y). Il peut ne pas y avoir de paramètres ou y en avoir autant qu'on le souhaite. Noter que le type des paramètres peut être explicite ou être inféré par le contexte; par exemple (a). S'il n'y a qu'un seul paramètre inféré, les parenthèses ne sont pas obligatoires, comme par exemple dans :

```
a -> return a*a
```

- La deuxième partie est une symbolique Java pour exprimer une lambda (->)
- La troisième partie forme le corps de la lambda; il peut s'agir d'une simple expression ou encore d'un bloc. Par exemple, x + y.

```
(int x, int y) -> x + y
```

S'il s'agit d'une simple expression, le corps est évalué et retourné. S'il s'agit d'un bloc, le corps est évalué comme celui d'une méthode : return renvoie une valeur au programme appelant.

Si le corps n'a qu'une seule instruction, les accolades ne sont pas obligatoires et le type de la valeur de retour est le type de l'expression du corps.

Si le corps a plusieurs instructions, les accolades sont obligatoires et le type de la valeur de retour est le type de l'expression retournée ou void si rien n'est retourné.

En voici quelques exemples :

(int x, int y) -> x + y : expression prenant deux arguments entiers et retournant x+y, en utilisant la forme expression

() -> 42 : expression ne prenant pas d'arguments et retournant 42, en utilisant la forme expression

(String s) -> { System.out.println(s); } : expression prenant un argument chaîne et l'affichant, en utilisant la forme bloc.

Constructions possibles	Constructions impossibles
() -> {...}	-> {...}
(a)-> {...}	int a-> {...}
(a,b)->{...}	a,b->{...}
(int a, int b)->{...}	(int a, b)->{...}
a->{...}	
(...)->System.out.println("hello");	
(...)-> {  System.out.println("hello");  System.out.println("world");  }	(...)-> {  break; // interdit si pas dans une boucle  }
(...)-> {  return a + b;  }	
(...)-> return a + b;	

## b) Références de méthodes

Il existe quelquefois une méthode qui fait le traitement que l'on aimerait passer à un bout de code. Par exemple, si l'on souhaite afficher un objet event à chaque fois qu'un bouton est utilisé :

```
button.setOnAction(event ->System.out.println(event));
```

Java a introduit dans ce cas une simplification de notation :

```
button.setOnAction(System.out::println);
```

L'expression `System.out::println` est une "référence de méthode" équivalente à l'expression lambda : `event ->System.out.println(event)`

Comme il est possible de le voir, l'opérateur :: sépare le nom de la méthode du nom de l'objet ou de la classe.

Les syntaxes générales sont :

objet :: méthode_d_instance	Référence de méthode équivalente à la lambda fournissant les paramètres à la méthode.  Exemples:  System.out::println <=> x->System.out.println(x)  Math::pow <=> (x,y)->Math.pow(x,y)
Classe::méthode_statique	Idem
Classe::méthode_d_instance	Dans ce cas, le premier paramètre devient l'objet sur lequel on déclenche la méthode  String::compareToIgnoreCase <=> (x,y)->x.compareToIgnoreCase(y)
this::méthode_d_instance	Extension du premier cas à this  this::equals <=> x->this.equals(x)
Super::méthode_d_instance	Invoke la version de la méthode appartenant à la classe de base

Il est intéressant de noter qu'il existe une version un peu équivalente, appelé constructeur de référence, dans laquelle le nom de la méthode est remplacé par new. Par exemple, JButton::new est la référence au constructeur de la classe JButton.

Exemple :

```
List<String> labels = ...  
Stream<JButton> stream = labels.stream().map(JButton::new);  
List<JButton> buttons = stream.collect(Collectors.toList());
```

Cette utilité existe aussi en version tableau: int[]::new est l'équivalent de x->new int[x].



### c) Capture des variables

Un problème qui se pose est celui de la manipulation des variables locales dans les expressions lambda; en effet, comme celles-ci peuvent être utilisées bien après la fin de la fonction contenant leur définition (un peu comme les classes anonymes), il n'est pas possible qu'elles manipulent des variables, normalement supprimées en fin de fonction.

Les règles suivantes sont donc appliquées :

- les données membres de classes ou les données statiques sont accessibles,
- les paramètres de la fonction contenant la définition de la lambda à condition qu'ils soient "final"
- les variables construites dans le corps de la fonction à condition qu'elles soient "final"
- les paramètres déclarés dans l'expression lambda
- les variables déclarées dans le corps de la lambda

Il faut comprendre ici que même si les variables ne sont pas déclarées en final, le mot-clé leur sera automatiquement attribué par le compilateur, et que donc il ne sera plus possible de modifier leur valeur après leur initialisation.

```
class FiltreDeFichiers {

    private static final String extUn = ".c";
    private static String extDeux = ".cpp";
    private String extTrois;
    private String extQuatre; // Can not be captured!

    public FiltreDeFichiers(String extTrois) {
        this.extTrois = extTrois;
    }

    public String[] doFilter(String filtre) {

        String ext = ".bla";

        File repertoire = new File(System.getProperty("user.home"));
        String[] nomsFichiers = repertoire
            .list((File dirToFilter, String filename) -> {

                String autreFiltre = ".yaf";

                return filename.endsWith(extUn) ||
                    filename.endsWith(extDeux) ||
                    filename.endsWith(extTrois) ||
                    filename.endsWith(ext) ||
                    filename.endsWith(filtre) ||
                    filename.endsWith(autreFiltre);
            });
    }
}
```

```

        return nomsFichiers;
    }

    public void setQuatre(String extQuatre) {
        this.extQuatre = extQuatre;
    }
}

```

#### d) Le pointeur this

Il suffit de comprendre que this désigne l'objet qui serait normalement référencé à l'extérieur de la lambda.

#### e) Exemples

##### Lambda Runnable

```

6 public class RunnableTest {
7     public static void main(String[] args) {
8
9         System.out.println("=== RunnableTest ===");
10
11         // Anonymous Runnable
12         Runnable r1 = new Runnable() {
13
14             @Override
15             public void run() {
16                 System.out.println("Hello world one!");
17             }
18         };
19
20         // Lambda Runnable
21         Runnable r2 = () -> { System.out.println("Hello world two!"); };
22
23         // Run em!
24         r1.run();
25         r2.run();
26
27     }
28 }

```

##### Lambda Comparator

```

9 public class Person {
10     private String givenName;
11     private String surName;
12     private int age;
13     private Gender gender;
14     private String eMail;
15     private String phone;
16     private String address;
17

```

```

10 public class ComparatorTest {
11
12     public static void main(String[] args) {
13
14         List<Person> personList = Person.createShortList();
15
16         // Sort with Inner Class
17         Collections.sort(personList, new Comparator<Person>() {
18             public int compare(Person p1, Person p2) {
19                 return p1.getSurName().compareTo(p2.getSurName());
20             }
21         });
22
23         System.out.println("=== Sorted Asc SurName ===");
24         for(Person p:personList){
25             p.printName();
26         }
27
28         // Use Lambda instead
29         Comparator<Person> SortSurNameAsc = (Person p1, Person p2) ->
p1.getSurName().compareTo(p2.getSurName());
30         Comparator<Person> SortSurNameDesc = (p1, p2) ->
p2.getSurName().compareTo(p1.getSurName());
31
32         // Print Asc
33         System.out.println("=== Sorted Asc SurName ===");
34         Collections.sort(personList, SortSurNameAsc);
35
36         for(Person p:personList){
37             p.printName();
38         }
39
40         // Print Desc
41         System.out.println("=== Sorted Desc SurName ===");
42         Collections.sort(personList, SortSurNameDesc);
43
44         for(Person p:personList){
45             p.printName();
46         }
47     }
48 }
49 }

```

## Lambda Lambda

```

19 public class ListenerTest {
20     public static void main(String[] args) {
21
22         JButton testButton = new JButton("Test Button");
23         testButton.addActionListener(new ActionListener() {
24             public void actionPerformed(ActionEvent ae) {
25                 System.out.println("Click Detected by Anon Class");
26             }
27         });
28
29         testButton.addActionListener(e -> {

```

```
30     System.out.println("Click Detected by Lambda Listner");
31 });
32
33 // Swing stuff
34 JFrame frame = new JFrame("Listener Test");
35 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
36 frame.add(testButton, BorderLayout.CENTER);
37 frame.pack();
38 frame.setVisible(true);
39
40 }
41 }
```

### III. Les streams (interface java.util.stream.Stream)

#### a) Généralités

Un des cas typiques d'utilisation des expressions lambda introduits par Java 8 est la manipulation de Stream. Il s'agit d'objets (qui n'ont strictement rien à voir avec InputStream ou OutputStream), qui ont été créés pour permettre d'appliquer des traitements sur des collections sans passer par les méthodes habituelles (les méthodes get, les itérateurs etc ...) en introduisant notamment l'idée de parallélisme.

Il est possible de les considérer comme des wrappers (des classes enveloppes) permettant de travailler sur des sources de données comme des tableaux ou des collections, proposant toute une série de traitements (forEach, map, filter, reduce, findFirst ...) à appliquer à ces données de manière séquentielle ou parallèle.

Exemple :

```
Integer[] array = { 10, 12, 14, 15, -5 };  
Stream.of(array).filter(i -> i > 12).map(i -> i * i)  
    .forEach(i -> System.out.println(i));
```

Les principales caractéristiques sont les suivantes :

- Un stream se construit à partir d'une source de données (une collection, un tableau ou des sources I/O par exemple),
- Un stream ne stocke pas de données, contrairement à une collection. Il se contente de les transférer d'une source vers une suite d'opérations.
- Un stream ne permet pas de modifier les données de sa source; si l'on souhaite des données modifiées, il faut construire un nouveau stream à partir de la source de données initiales
- Les streams sont des objets paresseux et ont un comportement lazy : il essaient de traiter le moins de données possibles pour arriver au résultat qui intéresse le développeur. Les opérations qui devraient traverser un stream de multiples fois ne le font en fait qu'une seule fois. Par exemple : `streama.map(someOp).filter(someTest).findFirst().get()` effectue le mapping et les opérations de filtre un élément à la fois, et continue jusqu'au premier résultat trouvé par le filtre.
- Un stream peut être infini (peut travailler sur un ensemble de données de taille illimité), contrairement aux collections. Il faudra cependant veiller à ce que les opérations se terminent en un temps fini – par exemple avec des méthodes comme `limit(n)` ou `findFirst()`.

- Enfin, un stream est utilisable une seule fois : une fois parcouru et traité, pour réutiliser les données, il faudra construire un nouvel objet stream sur la même collection ou source de données.

Noter que deux types d'opérations existent sur les stream :

- les opérations intermédiaires : ce sont des opérations qui créent de nouveaux stream, de manière lazy, que l'on peut empiler les uns sur les autres (map et les opérations associées comme mapToInt, flatMap, etc., filter, distinct, sorted, peek, limit, skip, parallel, sequential, unordered). Tant qu'aucune opération terminale n'aura été appelée sur un stream pipeline, les opérations intermédiaires ne seront pas réellement effectuées
- les opérations terminales : ce sont des opérations qui vont déclencher l'intégralité du traitement des opérations intermédiaires et calculer un résultat. Une fois cette opération exécutée, les streams sont détruits et ne pourront plus être utilisés. C'est la Javadoc qui indique quelle opération est terminale (forEach, forEachOrdered, toArray, reduce, collect, min, max, count, anyMatch, allMatch, noneMatch, findFirst, findAny, iterator) ou quelle opération est intermédiaire.

En fait, on peut rajouter une troisième catégorie d'opérations : les opérations short-circuit. Il s'agit d'opérations demandant aux opérations intermédiaires de continuer jusqu'au moment où l'opération short-circuit trouve un résultat (anyMatch, allMatch, noneMatch, findFirst, findAny, limit et skip)

## b) Construction d'un stream

Plusieurs techniques existent :

- à partir d'une collection : la méthode stream (ou parallelStream) est définie dans Interface Collection.

```
List<String> words = ...;
words.stream().filter(...).map(...).somethingElse(...);
List<Employee> workers = ...;
workers.stream().map(...).filter(...).other(...);
```

- à partir d'un tableau d'objets : la méthode statique of de la classe Stream est applicable à un tableau d'objets

```
Employee[] workers = ...;
Stream.of(workers).map(...).filter(...).other(...);
```

- à partir d'un tableau de valeurs primitives : la méthode statique stream de la classe Arrays est applicable à un tableau de valeurs primitives

```
int[] arrayInt = { 10, 12, 14, 15, -5 };
Arrays.stream(arrayInt).filter(i -> i > 12).map(i -> i * i)
    .forEach(i -> System.out.println(i));
```

- à partir d'éléments individuels : la méthode statique of de la classe Stream est applicable à des objets pris en tant qu'éléments

```
Employee e1 = ...;
– Employee e2 = ...;
– Stream.of(e1,e2).map(...).filter(...).other(...);
```

- à partir d'expressions lambda : certaines méthodes permettent de générer des valeurs, comme Stream.generate ou Stream.iterate  

```
Stream.iterate(1, x -> x*2)
```

renverra un stream infini d'entiers contenant la suite des puissances de 2. Le premier argument contient la valeur initiale du stream, et le deuxième la fonction permettant de passer de l'élément n à l'élément n+1 dans le stream.
- à partir d'autres stream (filter, map etc...)
- à partir de méthodes Java : par exemple, la méthode chars() de la classe String, qui renvoie un IntStream construit sur les différents caractères de la chaîne de caractères, ou encore la méthode lines() de la classe BufferedReader qui crée un stream de chaînes de caractères à partir des lignes du fichier ouvert. À la classe Random s'ajoute aussi une méthode intéressante, ints(), qui renvoie un stream d'entiers pseudo aléatoires.

Il peut être intéressant de noter qu'un stream peut produire des structures de données classiques :

- **des tableaux**
  - strm.toArray(EntryType[]::new)
    - E.g., employeeStream.toArray(Employee[]::new)
The argument to toArray is normally EntryType[]::new, but in general is a Supplier that takes an int (size) as an argument and returns an empty array that can be filled in. There is also a zero-argument toArray() method, but this returns Object[], not Blah[], and it is illegal to cast Object[] to Blah[], even when all entries in the array are Blahs.
- des listes
  - strm.collect(Collectors.toList())
    - Common to do “import static java.util.stream.Collectors.\*;”
    - then to do strm.collect(toList())
- d'autres collections
  - strm.collect(Collectors.toSet())
  - strm.collect(Collectors.groupingBy(...)), etc.
- des chaînes (String)
  - strm.collect(Collectors.toStringJoiner(delim)).toString()





### c) Quelques opérations intéressantes

Fonctions	Rôles et propriétés	Exemples
forEach	<p>forEach permet de boucler sur tous les éléments d'un stream; elle attend une expression lambda et cette lambda est appliquée à chacun des éléments du stream (en réalité, le développeur doit fournir un Consumer à forEach, chaque élément du stream est transmis à la méthode accept du Consumer)</p> <p>forEach est une opération terminale</p> <p>Il existe aussi une méthode peek, fonctionnant de la même façon mis retournant le stream original en fin de traitement</p> <p>Avantages par rapport à une boucle for :</p> <ul style="list-style-type: none"> <li>• conçue pour les lambdas et donc court</li> <li>• réutilisable</li> <li>• peut être rendue // sans effort</li> </ul> <p>someStream.<b>parallel()</b>.forEach(someLambda);</p>	<p>Affichage des éléments d'un stream</p> <pre>Stream.of(tableau).forEach(System.out::println);</pre> <p>Vider les JTextfield d'un écran</p> <pre>fieldList.stream().forEach(field -&gt; field.setText(""));</pre> <p>Réutilisation</p> <pre>Consumer&lt;Employee&gt; giveRaise = e -&gt; {     System.out.printf("%s earned \$%,d before raise.%n",         e.getFullName(), e.getSalary());     e.setSalary(e.getSalary() * 11/10);     System.out.printf("%s will earn \$%,d after raise.%n",         e.getFullName(), e.getSalary()); }; googlers().forEach(giveRaise); sampleEmployees().forEach(giveRaise);</pre>
map	<p>map permet de fabriquer un nouveau stream en appliquant une fonction à chacun des éléments d'un stream original</p>	<p>Fabrication de tableau contenant des carrés de nombres réels</p> <pre>Double[] nums = { 1.0, 2.0, 3.0, 4.0, 5.0 }; Double[] squares =     Stream.of(nums).map(n -&gt; n * n).toArray(Double[]::new);</pre> <p>Liste d'employés de codes donnés</p> <pre>Integer[] ids = { 1, 2, 4, 8 }; List&lt;Employee&gt; matchingEmployees =     Stream.of(ids).map(EmployeeUtils::findById)         .collect(Collectors.toList());</pre>
filter	<p>filter permet de fabriquer un nouveau stream contenant les éléments vérifiant un prédicat</p>	<p>Tableau de nombres entiers pairs</p> <pre>Integer[] nums = { 1, 2, 3, 4, 5, 6 }; Integer[] evens =     Stream.of(nums).filter(n -&gt; n%2 == 0).toArray(Integer[]::new);</pre>

		<p>Tableaux de nombres entiers plus grands que 3</p> <pre>Integer[] evens = Stream.of(nums).filter(n -&gt; n%2 == 0) .filter(n -&gt; n&gt;3)  .toArray(Integer[]::new);</pre>
findFirst	<p>findFirst renvoie le premier élément d'un stream (operation de type short-circuit) sous a forme d'un objet Optional (stocke un objet de type T ou bien rien); cette opération est très rapide avec map ou filter car elle stoppe dès qu'un élément est trouvé</p> <p>NB :</p> <ul style="list-style-type: none"> <li>• si l'on sait qu'il y a au moins une entrée : stream.map(...).findFirst().get()</li> <li>• si l'on ne sait pas si il y a au moins une entrée stream.filter(...).findFirst().orElse(otherValue)</li> </ul> <p><b>Syntaxe des Optional</b></p> <p><b>Fabrication</b></p> <pre>Optional&lt;Type&gt; value = Optional.of(valeurType); Optional&lt;Type&gt; value = Optional.empty(); // pas de valeur</pre> <p><b>Fonctions courantes</b></p> <ul style="list-style-type: none"> <li>• value.get() : renvoie la valeur si présente ou émet une exception</li> <li>• value.orElse(other) : renvoie la valeur su présente ou other</li> <li>• value.ifPresent(Consumer) : exécute la lambda si val existe</li> <li>• value.isPresent(): renoie true si val existe</li> </ul>	
reduce	<p>reduce est une opération permettant de combiner une première valeur (seed) avec le premier élément d'un stream, d'en prendre le résultat et de le combiner avec la deuxième valeur du même stream et ainsi de suite.</p>	<p>Maximum d'une série de nombres réels</p> <pre>nums.stream().reduce(Double.MIN_VALUE, Double::max)</pre> <p>Produit de nombres entiers</p> <pre>nums.stream().reduce(1, (n1, n2) -&gt; n1 * n2)</pre> <p>Concaténation</p> <pre>List&lt;String&gt; letters = Arrays.asList("a", "b", "c", "d"); String concat = letters.stream().reduce("", String::concat);</pre>

limit, size	<p>Ces deux opérations ont vocation à limiter la taille d'un stream.</p> <p>limit(n) renvoie un stream formé des n premiers éléments d'un stream  skip(n) ren  Les deux sont des opérations short-circuit (par exemple, si on dispose d'un flux de 1000 éléments, l'instruction suivante applique fn1 10 fois, évalue pred 10 fois, et applique fn2 au plus 10 fois.  <b>strm.map(fn1).filter(pred).map(fn2).limit(10)</b></p>	<p>Extraction des 10 premiers éléments  stream.limit(10)</p> <p>Extraction des 15 deniers éléments  twentyElementStream.skip(5)</p> <p>Conversion d'un stream en liste  List&lt;String&gt; emps = googlers().map(Person::getFirstName)  .limit(8)  .skip(2)  .collect(Collectors.toList());  System.out.printf("Names of 6 Googlers: %s.%n", emps);  Noter qu'ici la fonction getFirstName n'est appelée que 6 fois même si le flux possède 1000 éléments</p>
sorted, min, max, distinct	<p>Il s'agit d'opérations basées sur des comparaisons.</p> <p>sorted : trie un stream de manière intéressante car on peut lui appliquer map, filter, limit auparavant (mais limit et skip ne sont plus ici des opérations short-circuit) et utilise un Comparable (si non défini dans la classe)</p> <p>min et max : renvoie le plus petit ou le plus grand des éléments d'un flux en utilisant un Comparable obligatoire et renvoie un Optional</p> <p>distinct : supprime les éléments identiques d'un stream</p>	<p>Tri par salaire  empStream.map(...).filter(...).limit(...)  .sorted((e1, e2) -&gt; e1.getSalary() - e2.getSalary())</p> <p>Meilleur salaire  empStream.max((e1, e2) -&gt; e1.getSalary() - e2.getSalary())  .get();</p> <p>Doublons de mots supprimés  stringStream.distinct()</p> <p>Tri d'un stream d'employés  List&lt;Employee&gt; emps3 =  sampleEmployees().sorted(Person::firstNameComparer)  .limit(2)  .collect(Collectors.toList());  System.out.printf("Employees sorted by first name: %s.%n", emps3);</p> <p>Noter que limit(2) ne réduit pas le nombre de fois où</p>

		firstNameComparer est appelée
allMatch, noneMatch, anyMatch, count	<p>Les trois premières opérations vérifient un prédicat, et renvoient un booléen en s'arrêtant dès qu'un résultat peut être fourni. Par exemple, allMatch s'arrête si le premier élément ne vérifie pas le prédicat imposé.</p> <p>Count renvoie simplement un nombre d'éléments (est une opération terminale)</p>	<p>Existe-t-il un employé gagnant plus de 500000 euros employeeStream.anyMatch(e -&gt; e.getSalary() &gt; 500000)</p> <p>Combien d'employés vérifient un critère ? employeeStream.filter(somePredicate).count()</p>
generate, iterate	<p>Ces deux opérations génèrent des ensembles de données accessibles au travers de flux :</p> <p>Stream.generate(valueGenerator) : cette opération attend un Supplier, appelé à chaque fois que le système a besoin d'un élément. Elle ne fonctionne pas en //</p> <p>Stream.iterate(initialValue, valueTransformer) : cette opération attend un opérateur unaire f. La graine (initialValue) devient le premier élément du flux, f(initialValue) devient le second etc ...</p> <p>Noter que les valeurs ne sont pas calculées avant que le besoin s'en fasse sentir, et que pour éviter des générations infinies, il est nécessaire d'utiliser limit ou findFirst</p>	<p>Génération d'une liste d'employés List&lt;Employee&gt; emps = Stream.generate(() -&gt; randomEmployee()) .limit(...) .collect(Collectors.toList());</p> <p>Génération de nombres Supplier&lt;Double&gt; random = Math::random; System.out.println("2 Random numbers:"); Stream.generate(random).limit(2).forEach(System.out::println); System.out.println("4 Random numbers:"); Stream.generate(random).limit(4).forEach(System.out::println);</p> <p>Génération d'une liste de puissances de deux List&lt;Integer&gt; powersOfTwo = Stream.iterate(1, n -&gt; n * 2) .limit(...) .collect(Collectors.toList());</p>

#### **d) Parallélisation**

L'un des points forts de cette nouvelle API est de permettre de paralléliser des traitements de façon particulièrement aisée. En effet, n'importe quel stream peut être parallélisé en appelant sa méthode `parallel()` héritée de l'interface `BaseStream` – de la même façon, un stream peut être rendu séquentiel en invoquant la méthode `sequential()`. On peut également construire un stream parallèle sur une collection directement en appelant la méthode `parallelStream()` sur cette collection.

Ces méthodes nous permettent de masquer la répartition du travail, mais ne doivent pas être prises à la légère : en essayant de gagner en performance en parallélisant n'importe quel traitement, on prend le risque de produire l'effet inverse (nous y reviendrons plus tard).

## Why Java needs Lambda Expressions?

Since its beginning, the Java language hasn't evolved much if you ignore some of the features like Annotations, Generics etc. Mostly during its life Java always remained Object first language. After working with functional language like JavaScript, it becomes clear to one how Java enforce its strict object-oriented nature and strict typed on the source code. You see Functions are not important for Java. On their own they cannot live in Java world.



Functions are first class citizens in a functional programming language. They exist on their own. You can assign them to a variable and pass them as arguments to other functions. JavaScript is one of the best examples of an FP language. There are some good articles [here](#) and [here](#) that clearly describe the benefits of JavaScript as a functional language. A functional language provides a very powerful feature called Closure that has quite a few advantages over the traditional way of writing applications. A closure is a function or reference to a function together with a referencing environment — a table storing a reference to each of the non-local variables of that function. The closest thing that Java can provide to Closure is Lambda expressions. There is a significant difference between a Closure and Lambda expression, but at least Lambda expression provides a good alternative to Closure.

In his quite sarcastic and funny [blog post](#), Steve Yegge describes how the Java world is strictly about Nouns. If you haven't read his blog, go first read it. It's funny, it's interesting and it describes the exact reason why Java had to add Lambda expressions.

Lambda expression adds that missing link of functional programming to Java. Lambda expression lets us have functions as first class citizens. Although this is not 100% correct, we will shortly see how Lambda expressions are not closures but they are as much close as we can get to closures. In languages that support first class functions, the type of the lambda expression would be a function; but in Java, the lambda expressions are represented as objects, and so they must be bound to a particular object type known as a functional interface. We will see in detail what Functional interface are.

Here is a [nicely written article](#) by Mario Fusco on Why we need Lambda Expression in Java. He explains why a modern programming language must have features like closures.

# Why We Need Lambda Expressions in Java - Part 1

Lambda expressions are coming to Java 8 and together with Raoul-Gabriel Urma and Alan Mycroft I started writing a [book](#) on this topic. Anyway apparently they are still encountering some resistance and not all Java developers are convinced of their usefulness. In particular they say that it could be a mistake to try to add some functional features to Java, because they fear that this could compromise its strong object oriented and imperative nature. The purpose of this article is to hopefully remove any further doubt and show clearly, with practical and straightforward examples, why it is not possible for a modern programming language to not support lambda expressions.

## External vs. internal iteration

Let's start with something very simple, a list of integers:

[view source](#)

[print?](#)

```
1.List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
```

and a for cycle that iterates all the items in the list and prints them:

[view source](#)

[print?](#)

```
1.for (int number : numbers) {  
2.System.out.println(number);  
3.}
```

Straightforward as much as common: I don't remember a single day of my more than decennial working life as Java developer when I haven't write at least one cycle like this. Simple as much as ... completely wrong. It reminds me a lot how my 2 years old daughter Sofia puts away her toys after having played with them. It goes on more or less in this way:

Me: "Sofia, let's put the toys away. Is there a toy on the ground"

Sofia: "Yes, the ball"

Me: "Ok, put the ball in the box. Is there something else?"

Sofia: "Yes, there is my doll"

Me: "Ok, put the doll in the box. Is there something else?"

Sofia: "Yes, there is my book"

Me: "Ok, put the book in the box. Is there something else?"

Sofia: "No, nothing else"

Me: "Fine, we are done"

This is exactly what we do everyday with our Java collections, but unfortunately the biggest

part of us is not 2 years old. We iterate the collection externally, explicitly pulling out and processing the items one by one. It would be far better for me if I could tell to Sofia just: "put inside the box all the toys that are on the ground". There are two other reasons, why an internal iteration is preferable: first Sofia could choose to take at the same time the doll with one hand and the ball with the other and second she could decide to take the objects closest to the box first and then the others. In the same way using an internal iteration the JIT compiler could optimize it processing the items in parallel or in a different order. These optimizations are impossible if we iterate the collection externally as we are used to do in Java and more in general with the imperative programming.

So, why don't we iterate internally? I think this is only a bad mental habit caused by the lack of support of this pattern in the Java Collection Framework that in turn has been caused by the verbosity (creation of an anonymous inner class) that this implies in pre-8 Java. Something like this:

[view source](#)  
[print?](#)

```
1.numbers.forEach(new Consumer<Integer>() {  
2.public void accept(Integer value) {  
3.System.out.println(value);  
4.}  
5.});
```

Actually both the `forEach` method and the `Consumer` interface have been added in Java 8, but you can already do something very similar in Java 5+ using libraries like [guava](#) or [lambdaj](#). However Java 8 lambda expressions allow to achieve the same result in a less verbose and more readable way:

[view source](#)  
[print?](#)

```
1.numbers.forEach((Integer value) -> System.out.println(value));
```

The lambda expression is made of two parts the one on the left of the arrow symbol (`->`) listing its parameters and the one on the right containing its body. In this case the compiler automatically figures out that the lambda expression has the same signature of the only non implemented method of the `Consumer` interface (that for this reason is called a functional interface) and treat the first as it was an instance of the second, even if the generated bytecode could potentially be different. The declaration of the types of the lambda expression arguments can be, in the biggest part of cases, inferred by the compiler and then omitted as it follows:

[view source](#)  
[print?](#)

```
1.numbers.forEach(value -> System.out.println(value));
```

But we can rewrite this last statement even more concisely using a method reference, another feature introduced in Java 8. More in details in Java 8 it is possible to reference both a static and an instance a method using the new `::` operator as in:

[view source](#)  
[print?](#)

```
1.numbers.forEach(System.out::println);
```

In this way, with a process that in functional programming is known as eta expansion, the name of the method is "expanded" by the compiler in the method itself that, as we have already seen, has the same signature of the only abstract method of the `Consumer` functional interface and then can be in turn converted in an instance of it.

## Passing behaviors, not only values



What we have seen in the former example is the main and possibly the only reason why lambda expressions are so useful. Passing a lambda expression to another function allow us to pass not only values but also behaviors and this enable to dramatically raise the level of our abstraction and then project more generic, flexible and reusable API. Let's reenforce this with a further example: starting with the usual list of Integer

[view source](#)  
[print?](#)

```
1.List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
```

we are requested to write a method that sums all the Integers in the list as for instance:

[view source](#)  
[print?](#)

```
1.public int sumAll(List<Integer> numbers) {  
2.int total = 0;  
3.for (int number : numbers) {  
4.total += number;  
5.}  
6.return total;  
7.}
```

The day after a manager comes to our cubicle and tells you that the business also requires to have a function that sums only the even number in the list. So what is the quickest thing we could do? Easy. Just copy and paste the former method and add to it the required filtering condition:

[view source](#)  
[print?](#)

```
01.public int sumAllEven(List<Integer> numbers) {  
02.int total = 0;  
03.for (int number : numbers) {  
04.if (number % 2 == 0) {  
05.total += number;  
06.}  
07.}  
08.return total;  
09.}
```

Another day, another requirement: this time they need to sum the numbers in the list again but only if they are greater than 3. So what could we do? Well, we could again copy and paste the former method and just change that boolean condition ... but it feels so dirty, isn't it? Now, following the "[First Write, Second Copy, Third Refactor](#)" principle it is time to wonder if there is a smarter and more generic way to do this. In this case implementing an higher-order function accepting together with the list also a Predicate (another functional interface added in Java 8) that defines how to filter the numbers in the list itself before to sum them up.

[view source](#)  
[print?](#)

```
01.public int sumAll(List<Integer> numbers, Predicate<Integer> p) {  
02.int total = 0;  
03.for (int number : numbers) {  
04.if (p.test(number)) {  
05.total += number;  
06.}  
07.}  
08.return total;  
09.}
```

In other words we are passing to the method not only the data (the list of numbers) but also a behavior (the Predicate) defining how to use them. In this way we can satisfy all the 3 requirements with a single more generic and then more reusable method:

[view source](#)

[print?](#)

```
1.sumAll(numbers, n -> true);  
2.sumAll(numbers, n -> n % 2 == 0);  
3.sumAll(numbers, n -> n > 3);
```

In the [second part of this article](#) I will show other examples to demonstrate how lambda expressions can make our Java code more readable and concise.

## Why We Need Lambda Expressions in Java - Part 2

04.01.2013

| 16641 views |

[inShare](#)3

[inShare](#)

[+ reddit this!](#)

In the [first part of this article](#) I started explaining, with hopefully straightforward examples, how the introduction of lambda expressions can make Java a more concise, readable, powerful and, in a word, modern language. I also announced that together with Raoul-Gabriel Urma and Alan Mycroft I started writing a [book](#) on this topic. In this second part I'll try to reenforce this idea with 2 more equally practical examples, hoping to help to convince all the Java developers, especially the most experienced ones, that we can no longer wait before to have a more functional oriented Java.

### Efficiency through laziness

Another advantage of internal iteration of collections and more in general of functional programming is the lazy evaluation that it allows. Let me demonstrate this starting again from the same List of Integer I already used in the first part:

[view source](#)

[print?](#)

```
1.List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
```

This time let's suppose we want to achieve something just a bit more convoluted like to take only the even numbers in the list, double them and finally print the first one bigger than 5:

[view source](#)

[print?](#)

```

01.for (int number : numbers) {
02.if (number % 2 == 0) {
03.int n2 = number * 2;
04.if (n2 > 5) {
05.System.out.println(n2);
06.break;
07.}
08.}
09.}

```

What is wrong with this solution should be clear: it does too much in a single, deeply nested and then difficult to read and maintain block of code. To fix this problem we can split the 3 operations performed by that code in 3 one-line methods each having a single well defined responsibility:

[view source](#)

[print?](#)

```

01.public boolean isEven(int number) {
02.return number % 2 == 0;
03.}
04.
05.public int doubleIt(int number) {
06.return number * 2;
07.}
08.
09.public boolean isGreaterThan5(int number) {
10.return number > 5;
11.}

```

and rewrite the former code using them:

[view source](#)

[print?](#)

```

01.List<Integer> l1 = new ArrayList<Integer>();
02.for (int n : numbers) {
03.if (isEven(n)) l1.add(n);
04.}
05.
06.List<Integer> l2 = new ArrayList<Integer>();
07.for (int n : l1) {
08.l2.add(doubleIt(n));
09.}
10.
11.List<Integer> l3 = new ArrayList<Integer>();
12.for (int n : l2) {
13.if (isGreaterThan5(n)) l3.add(n);
14.}
15.
16.System.out.println(l3.get(0));

```

In practice I created a pipeline of list transformations where each stage uses one of the 3 methods operating on the single numbers. But is this code better than the former one?

Probably no, and for two distinct reasons: the first and more obvious one is its verbosity, while the second is that it is computationally inefficient because it does more work than what is strictly necessary. To make more evident this second issue we can add a println in each of the 3 isEven, doubleIt and isGreaterThan5 methods, obtaining the following output:

[view source](#)

[print?](#)

```

01.isEven: 1
02.isEven: 2
03.isEven: 3
04.isEven: 4
05.isEven: 5

```

```
06.isEven: 6
07.doubleIt: 2
08.doubleIt: 4
09.doubleIt: 6
10.isGreaterThan5: 4
11.isGreaterThan5: 8
12.isGreaterThan5: 12
13.8
```

Here we can note that all the 6 numbers in the list get evaluated by our pipeline while the former nested for cycle evaluated only the first 4, being the 4th number the first that satisfies all the requested conditions.

Streams can help a lot to fix both this problems. You can create a Stream from any Collection by invoking the new stream() method on it. However Streams differ from Collections in several ways:

- No storage: Streams don't have storage for values; they carry values from a data structure through a pipeline of operations.
- Functional in nature: an operation on a stream produces a result, but does not modify its underlying data source. A Collection can be used as a source for a stream.
- Laziness-seeking: many stream operations, such as filtering, mapping, sorting, or duplicate removal can be implemented lazily, meaning we only need to examine as many elements from the stream as we need to find the desired answer.
- Bounds optional: there are many problems that are sensible to express as infinite streams, letting clients consume values until they are satisfied. Collections don't let you do this, but streams do.

Using a Stream we can solve the former problem with a single fluent statement:

[view source](#)

[print?](#)

```
1.System.out.println(
2.numbers.stream()
3..filter(Lazy::isEven)
4..map(Lazy::doubleIt)
5..filter(Lazy::isGreaterThan5)
6..findFirst()
7.);
```

that produces the following output.

[view source](#)

[print?](#)

```
01.isEven: 1
02.isEven: 2
03.doubleIt: 2
04.isGreaterThan5: 4
05.isEven: 3
06.isEven: 4
07.doubleIt: 4
08.isGreaterThan5: 8
09.IntOptional[8]
```

Here we can note 2 things. First of all the laziness of the Stream allow us to don't waste CPU cycles and evaluate only the first 4 numbers in the List. In fact until, the second filter() invocation and before having called findFirst(), the Stream performs absolutely no operations because we haven't still asked a result to it. Second the findFirst() method doesn't return 8 but IntOptional[8]. An Optional is a wrapper of a value that can potentially doesn't exist at all. In other words in functional programming is common to use an Optional (the equivalent of

Option in Scala or Maybe in Haskell) to model a value that couldn't be there instead of returning a null reference as we are traditionally used to do in Object Oriented Java. I already wrote [another article](#) where I explained in more details how Optional works and pointed out some limitations of the current Java 8 implementation suggesting an alternative one. In this particular case, since we are sure that there is a result, we can safely unwrap the Optional by invoking `getAsInt()` on it and obtain the result 8 as expected.

## The loan pattern

To give one last example of how we can leverage functional programming in Java and in particular to show how we can achieve better encapsulation and avoid repetitions, let's suppose we have a Resource:

[view source](#)

[print?](#)

```
01.public class Resource {
02.
03.public Resource() {
04.System.out.println("Opening resource");
05.}
06.
07.public void operate() {
08.System.out.println("Operating on resource");
09.}
10.
11.public void dispose() {
12.System.out.println("Disposing resource");
13.}
14.}
```

that we can create, do something on it and, after having used it, we have to dispose in order to avoid leaks (of memory, file descriptors, etc.)

[view source](#)

[print?](#)

```
1.Resource resource = new Resource();
2.resource.operate();
3.resource.dispose();
```

What's wrong with this? Of course the fact that while operating on the resource we could get a `RuntimeException` so, in order to be sure that the `dispose()` method will be actually called, we have to put it in a finally block:

[view source](#)

[print?](#)

```
1.Resource resource = new Resource();
2.try {
3.resource.operate();
4.} finally {
5.resource.dispose();
6.}
```

The problem here is that we will have to repeat this try/finally block again and again every time we use the Resource (something that clearly violates the DRY principle) and in case we will forget it in some point of our code we will run the risk of having a leak. To fix this issue I suggest to encapsulate this try/finally block in a static method of the Resource class and possibly to make its constructor private in order to oblige the clients of that class to use it only through that method:

[view source](#)

[print?](#)

```
1. public static void withResource(Consumer<Resource> consumer) {  
2. Resource resource = new Resource();  
3. try {  
4. consumer.accept(resource);  
5. } finally {  
6. resource.dispose();  
7. }  
8. }
```

The argument passed to this method is an instance of the Consumer functional interface that allows to consume the resource or, in other words, to perform some actions on it. In this way we can operate on the resource by just passing a lambda expression to that method:

[view source](#)

[print?](#)

```
1. withResource(resource -> resource.operate());
```

It guarantees that the resource will be always disposed correctly avoiding any repetition. Also it should be now clearer where the name of this pattern comes from: the "lender" (the code holding the resource) manages the resources once the "lendee" (the lambda expression accessing it) has finished to use it.

[Javarevisited](#)

Blog about Java programming language, FIX Protocol, Tibco Rendezvous and related Java technology stack.

Wednesday, February 26, 2014

## 10 Example of Lambda Expressions and Streams in Java 8

Java 8 release is just a couple of weeks away, scheduled at 18th March 2014, and there is lot of buzz and excitement about this path breaking release in Java community. One of feature, which is synonymous to this release is lambda expressions, which will provide ability to pass behaviours to methods. Prior to Java 8, if you want to pass behaviour to a method, then your only option was Anonymous class, which will take 6 lines of code and most important line, which defines the behaviour is lost in between. *Lambda expression replaces anonymous classes* and removes all boiler plate, enabling you to write code in functional style, which is some time more readable and expression. This mix of bit of functional and full of object oriented capability is very exciting development in Java eco-system, which will further enable development and growth of parallel third party libraries to take advantage of multi-processor CPUs. Though industry will take its time to adopt Java 8, I don't think any serious Java developer can overlook key features of Java 8 release e.g. lambda expressions, functional interface, stream API, default methods and new Date and Time API. As a developer, I have found that best way to learn and master lambda expression is to try it out, do as many examples of lambda expressions as possible. Since biggest impact of Java 8 release will be on Java Collections framework its best to try examples of Stream API and lambda expression to extract, filter and sort data from Lists and Collections. I have been writing about Java 8 and have shared some

[useful resources to master Java 8](#) in past. In this post, I am going to share you 10 most useful ways to use lambda expressions in your code, these examples are simple, short and clear, which will help you to pick lambda expressions quickly.

## **Java 8 Lambda Expressions Examples**

I am personally very excited about Java 8, particularly lambda expression and stream API. More and more I look them, it makes me enable to write more clean code in Java. Though it was not like this always; when I first saw a Java code written using lambda expression, I was very disappointed with cryptic syntax and thinking they are making Java unreadable now, but I was wrong. After spending just a day and doing *couple of examples of lambda expression and stream API*, I was happy to see more cleaner Java code then before. It's like the [Generics](#), when I first saw I hated it. I even continued using old Java 1.4 way of dealing with Collection for few month, until one of my friend explained me benefits of using Generics. Bottom line is, don't afraid with initial cryptic impression of lambda expressions and method reference, you will love it once you do couple of examples of extracting and filtering data from Collection classes. So let's start this wonderful journey of learning lambda expressions in Java 8 by simple examples.

### **Example 1 - implementing Runnable using Lambda expression**

One of the first thing, I did with Java 8 was trying to replace anonymous class with lambda expressions, and what could have been best example of anonymous class then implementing Runnable interface. Look at the code of implementing runnable prior to Java 8, it's taking four lines, but with lambda expressions, it's just taking one line. What we did here? the whole [anonymous class](#) is replaced by `() -> {}` code block.

```
//Before Java 8:
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Before Java8, too much code for too little to
do");
    }
}).start();

//Java 8 way:
new Thread( () -> System.out.println("In Java8, Lambda expression rocks
!!") ).start();
```

#### **Output:**

```
too much code, for too little to do
Lambda expression rocks !!
```

This example brings us syntax of lambda expression in Java 8. You can write following kind of code using lambdas :

```
(params) -> expression
(params) -> statement
(params) -> { statements }
```

for example, if your method don't change/write a parameter and just print something on console, you can write it like this :

```
() -> System.out.println("Hello Lambda Expressions");
```

If your method accept two parameters then you can write them like below :

```
(int even, int odd) -> even + odd
```

By the way, it's general practice to **keep variable name short inside lambda expressions**. This makes your code shorter, allowing it to fit in one line. So in above code, choice of variable names as a,b or x, y is better than even and odd.

### Example 2 - Event handling using Java 8 Lambda expressions

If you have ever done coding in Swing API, you will never forget writing event listener code. This is another classic use case of plain old Anonymous class, but no more. You can write better event listener code using lambda expressions as shown below.

```
// Before Java 8:
JButton show = new JButton("Show");
show.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Event handling without lambda expression is boring");
    }
});

// Java 8 way:
show.addActionListener((e) -> {
    System.out.println("Light, Camera, Action !! Lambda expressions Rocks");
});
```

Another place where Java developers frequently use anonymous class is for providing [custom Comparator](#) to `Collections.sort()` method. In Java 8, you can replace your ugly anonymous class with more readable lambda expression. I leave that to you for exercise, should be easy if you follow the pattern, I have shown during implementing [Runnable](#) and `ActionListener` using lambda expression.

### Example 3 - Iterating over List using Lambda expressions

If you are doing Java for few years, you know that most common operation with Collection classes are iterating over them and applying business logic on each elements, for example processing a list of orders, trades and events. Since Java is an imperative language, all code looping code written prior to Java 8 was sequential i.e. their is on simple way to do parallel processing of list items. If you want to do parallel filtering, you need to write your own code, which is not as easy as it looks. Introduction of lambda expression and default methods has separated what to do from how to do, which means now Java Collection knows how to iterate, and they can now provide parallel processing of Collection



elements at API level. In below example, I have shown you [how to iterate over List](#) using with and without lambda expressions, you can see that now List has a `forEach()` method, which can iterate through all objects and can apply whatever you ask using lambda code.

```
//Prior Java 8 :
List features = Arrays.asList("Lambdas", "Default Method", "Stream API",
    "Date and Time API");
for (String feature : features) {
    System.out.println(feature);
}

//In Java 8:
List features = Arrays.asList("Lambdas", "Default Method", "Stream API",
    "Date and Time API");
features.forEach(n -> System.out.println(n));

// Even better use Method reference feature of Java 8
// method reference is denoted by :: (double colon) operator
// looks similar to scope resolution operator of C++
features.forEach(System.out::println);
```

**Output:**  
Lambdas  
Default Method  
Stream API  
Date and Time API

The last example of [looping over List](#) shows *how to use method reference in Java 8*. You see the double colon, scope resolution operator from C++, it is now used for method reference in Java 8.

#### Example 4 - Using Lambda expression and Functional interface Predicate

Apart from providing support for functional programming idioms at language level, Java 8 has also added a new package called `java.util.function`, which contains lot of classes to enable functional programming in Java. One of them is `Predicate`, By using `java.util.function.Predicate` functional interface and lambda expressions, you can provide logic to API methods to add lot of dynamic behaviour in less code. Following *examples of Predicate in Java 8* shows lot of common ways to [filter Collection data in Java code](#). `Predicate` interface is great for filtering.

```
public static void main(args[]){
    List languages = Arrays.asList("Java", "Scala", "C++", "Haskell",
    "Lisp");

    System.out.println("Languages which starts with J :");
    filter(languages, (str)->str.startsWith("J"));

    System.out.println("Languages which ends with a ");
    filter(languages, (str)->str.endsWith("a"));

    System.out.println("Print all languages :");
    filter(languages, (str)->true);

    System.out.println("Print no language : ");
    filter(languages, (str)->false);

    System.out.println("Print language whose length greater than 4:");
```

```

        filter(languages, (str)->str.length() > 4);
    }

    public static void filter(List names, Predicate condition) {
        for(String name: names) {
            if(condition.test(name)) {
                System.out.println(name + " ");
            }
        }
    }
}

```

#### Output:

```

Languages which starts with J :
Java
Languages which ends with a
Java
Scala
Print all languages :
Java
Scala
C++
Haskell
Lisp
Print no language :
Print language whose length greater than 4:
Scala
Haskell

```

```

//Even better
public static void filter(List names, Predicate condition) {
    names.stream().filter((name) -> (condition.test(name))).forEach((name)
-> {
        System.out.println(name + " ");
    });
}

```

You can see that filter method from Stream API also accept a Predicate, which means you can actually replace our custom filter() method with the in-line code written inside it, that's the power of lambda expression. By the way, Predicate interface also allows you test for multiple condition, which we will see in our next example.

#### Example 5 : How to combine Predicate in Lambda Expressions

As I said in previous example, java.util.function.Predicate allows you to combine two or more Predicate into one. It provides methods similar to logical operator AND and OR named as and(), or() and xor(), which can be used to combine the condition you are passing to filter() method. For example, In order to get all languages, which starts with J and are four character long, you can define two separate Predicate instance covering each condition and then combine them using Predicate.and() method, as shown in below example :

```

// We can even combine Predicate using and(), or() And xor() logical
functions
// for example to find names, which starts with J and four letters long,
you
// can pass combination of two Predicate

```

```

Predicate<String> startsWithJ = (n) -> n.startsWith("J");
Predicate<String> fourLetterLong = (n) -> n.length() == 4;

names.stream()
    .filter(startsWithJ.and(fourLetterLong))
    .forEach((n) -> System.out.print("\nName, which starts with 'J' and
four letter long is : " + n));

```

Similarly you can also use `or()` and `xor()` method. This example also highlight important fact about using Predicate as individual condition and then combining them as per your need. In short, you can use Predicate interface as traditional Java imperative way, or you can take advantage of lambda expressions to write less and do more.

### Example 6 : Map and Reduce example in Java 8 using lambda expressions

This example is about one of the popular functional programming concept called map. It allows you to transform your object. Like in this example we are transforming each element of `costBeforeTeax` list to including Value added Test. We passed a lambda expression `x -> x*x` to `map()` method which applies this to all elements of the stream. After that we use `forEach()` to print the all elements of list. You can actually get a List of all cost with tax by using Stream API's Collectors class. It has methods like `toList()` which will combine result of map or any other operation. Since Collector perform terminal operator on Stream, you can't re-use Stream after that. You can even use `reduce()` method from Stream API to reduce all numbers into one, which we will see in next example

```

// applying 12% VAT on each purchase
// Without lambda expressions:
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
for (Integer cost : costBeforeTax) {
    double price = cost + .12*cost;
    System.out.println(price);
}

// With Lambda expression:
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
costBeforeTax.stream().map((cost) -> cost +
.12*cost).forEach(System.out::println);

```

#### Output

```

112.0
224.0
336.0
448.0
560.0
112.0
224.0
336.0
448.0
560.0

```

### Example 6.2 - Map Reduce example using Lambda Expressions in Java 8

In previous example, we have seen how map can transform each element of a Collection class e.g.

List. There is another function called `reduce()` which can combine all values into one. Map and Reduce operations are core of functional programming, reduce is also known as fold operation because of its folding nature. By the way reduce is not a new operation, you might have been already using it. If you can recall SQL aggregate functions like `sum()`, `avg()` or `count()`, they are actually reduce operation because they accept multiple values and return a single value. Stream API defines `reduce()` function which can accept a lambda expression, and combine all values. Stream classes like `IntStream` has built-in methods like `average()`, `count()`, `sum()` to perform reduce operations and `mapToLong()`, `mapToDouble()` methods for transformations. It doesn't limit you, you can either use built-in reduce function or can define yours. In this Java 8 Map Reduce example, we are first applying 12% VAT on all prices and then calculating total of that by using `reduce()` method.

```
// Applying 12% VAT on each purchase
// Old way:
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
double total = 0;
for (Integer cost : costBeforeTax) {
    double price = cost + .12*cost;
    total = total + price;
}
System.out.println("Total : " + total);

// New way:
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
double bill = costBeforeTax.stream().map((cost) -> cost +
    .12*cost).reduce((sum, cost) -> sum + cost).get();
System.out.println("Total : " + bill);
```

#### Output

```
Total : 1680.0
Total : 1680.0
```

### Example 7: Creating a List of String by filtering

Filtering is one of the common operation Java developers perform with large collections, and you will be surprise how much easy it is now to filter bulk data/large collection using lambda expression and stream API. Stream provides a `filter()` method, which accepts a Predicate object, means you can pass lambda expression to this method as filtering logic. Following examples of filtering collection in Java with lambda expression will make it easy to understand.

```
// Create a List with String more than 2 characters
List<String> filtered = strList.stream().filter(x -> x.length() >
2).collect(Collectors.toList());
System.out.printf("Original List : %s, filtered list : %s %n", strList,
filtered);
```

#### Output :

```
Original List : [abc, , bcd, , defg, jk], filtered list : [abc, bcd, defg]
```

By the way, there is a common confusion regarding `filter()` method. In real world, when we

filter, we left with something which is not filtered, but in case of using `filter()` method, we get a new list which is actually filtered by satisfying filtering criterion.

### Example 8: Applying function on Each element of List

We often need to apply certain function to each element of List e.g. multiplying each element by certain number or dividing it, or doing anything with that. Those operations are perfectly suited for `map()` method, you can supply your transformation logic to `map()` method as lambda expression and it will transform each element of that collection, as shown in below example.

```
// Convert String to Uppercase and join them using coma
List<String> G7 = Arrays.asList("USA", "Japan", "France", "Germany",
    "Italy", "U.K.", "Canada");
String G7Countries = G7.stream().map(x ->
    x.toUpperCase()).collect(Collectors.joining(", "));
System.out.println(G7Countries);
```

**Output :**

USA, JAPAN, FRANCE, GERMANY, ITALY, U.K., CANADA

### Example 9: Creating a Sub List by Copying distinct values

This example shows how you can take advantage of `distinct()` method of Stream class to filter duplicates in Collection.

```
// Create List of square of all distinct numbers
List<Integer> numbers = Arrays.asList(9, 10, 3, 4, 7, 3, 4);
List<Integer> distinct = numbers.stream().map(i ->
    i*i).distinct().collect(Collectors.toList());
System.out.printf("Original List : %s, Square Without duplicates : %s %n",
    numbers, distinct);
```

**Output :**

Original List : [9, 10, 3, 4, 7, 3, 4], Square Without duplicates : [81, 100, 9, 16, 49]

### Example 10 : Calculating Maximum, Minimum, Sum and Average of List elements

There is a very useful method called `summaryStatistics()` in stream classes like `IntStream`, `LongStream` and `DoubleStream`. Which returns an `IntSummaryStatistics`, `LongSummaryStatistics` or `DoubleSummaryStatistics` describing various summary data about the elements of this stream. In following example, we have used this method to calculate maximum and minimum number in a List. It also has `getSum()` and `getAverage()` which can give sum and average of all numbers from List.

```
//Get count, min, max, sum, and average for numbers
List<Integer> primes = Arrays.asList(2, 3, 5, 7, 11, 13, 17, 19, 23, 29);
IntSummaryStatistics stats = primes.stream().mapToInt((x) ->
    x).summaryStatistics();
System.out.println("Highest prime number in List : " + stats.getMax());
System.out.println("Lowest prime number in List : " + stats.getMin());
System.out.println("Sum of all prime numbers : " + stats.getSum());
System.out.println("Average of all prime numbers : " + stats.getAverage());
```

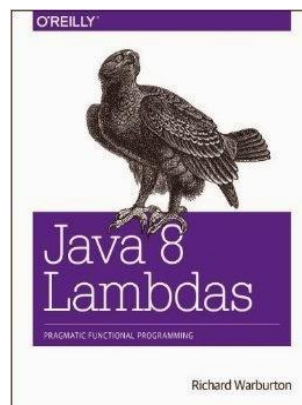
Output :

```
Highest prime number in List : 29
Lowest prime number in List : 2
Sum of all prime numbers : 129
Average of all prime numbers : 12.9
```

## Lambda Expression vs Anonymous class

Since lambda expression is effectively going to replace Anonymous inner class in new Java code, its important to do a comparative analysis of both of them. One key difference between using Anonymous class and Lambda expression is the use of [this keyword](#). For anonymous class 'this' keyword resolves to *anonymous class*, whereas for lambda expression 'this' keyword resolves to *enclosing class* where lambda is written. Another difference between lambda expression and anonymous class is in the way these two are compiled. Java compiler compiles lambda expressions and convert them into [private method](#) of the class. It uses `invokedynamic` byte code instruction from Java 7 to bind this method dynamically.

## Things to remember about Lambdas in Java 8



So far we have see 10 examples of lambda expression in Java 8, this is actually a good dose of lambdas for beginners, you will probably need to run examples by your own to get most of it. Try changing requirement, and create your own examples to learn quickly. I would also like to suggest using Netbeans IDE for practising lambda expression, it has got really good Java 8 support. Netbeans shows hint for converting your code into functional style as and when it sees opportunity. It's extremely easy to *convert an [Anonymous class](#) to lambda expression*, by simply following Netbeans hints. By the way, If you love to read books then don't forget to check Java 8 Lambdas, pragmatic functional programming by Richard Warburton, or you can also see Manning's Java 8 in Action, which is not yet published but I guess have free PDF of first chapter available online. But before you busy with other things, let's revise some of the important things about Lambda expressions, default methods and functional interfaces of Java 8.

1) Only predefined Functional interface using `@Functional` annotation or method with one abstract method or SAM (Single Abstract Method) type can be used inside lambda expressions. These are actually known as target type of lambda expression and can be used as return type, or parameter of lambda targeted code. For example, if a method accepts a `Runnable`, [Comparable](#) or

Callable interface, all has single abstract method, you can pass lambda expression to them. Similarly if a method accept interface declared in `java.util.function` package e.g. Predicate, Function, Consumer or Supplier, you can pass lambda expression to them.

2) You can use method reference inside lambda expression if method is not modifying the parameter supplied by lambda expression. For example following lambda expression can be replaced with a method reference since it is just a single method call with the same parameter:

```
list.forEach(n -> System.out.println(n));
```

```
list.forEach(System.out::println); // using method reference
```

However, if there's any transformations going on with the argument, we can't use method references and have to type the full lambda expression out, as shown in below code :

```
list.forEach((String s) -> System.out.println("*" + s + "*"));
```

You can actually omit the type declaration of lambda parameter here, compiler is capable to infer it from generic type of List.

3) You can use both [static](#), non static and [local variable](#) inside lambda, this is called **capturing variables inside lambda**.

4) Lambda expressions are also known as **closure** or **anonymous function** in Java, so don't be surprise if your colleague calling closure to lambda expression.

5) Lambda methods are internally translated into private methods and `invokedynamic` byte code instruction is now issued to dispatch the call:. You can use [javap tool](#) to decompile class files, it comes with JDK. Use command `javap -p` or `javap -c -v` to take a look at byte code generated by lambda expressions. It would be something like this

```
private static java.lang.Object lambda$0(java.lang.String);
```

6) One restriction with lambda expression is that, you can only reference either final or effectively final local variables, which means you cannot modified a variable declared in the outer scope inside a lambda.

```
List<Integer> primes = Arrays.asList(new Integer[]{2, 3, 5, 7});
int factor = 2;
primes.forEach(element -> { factor++; });
```

Compile time error : *"local variables referenced from a lambda expression must be final or effectively final"*

By the way, simply accessing them, without modifying is Ok, as shown below :

```
List<Integer> primes = Arrays.asList(new Integer[]{2, 3, 5, 7});
int factor = 2;
```

```
primes.forEach(element -> { System.out.println(factor*element); });
```

#### Output

4  
6  
10  
14

So its more like a closure with immutable capture, similar to Python.


That's all in this **10 examples of lambda expressions in Java 8**. This is going to be one of the biggest change in Java's history and will have a huge impact on how Java developer use Collections framework going forward. Anything I can think of similar scale was Java 5 release, which brings lots of goodies to improve code quality in Java, e.g. Generics, Enum, Autoboxing, Static imports, Concurrent API and variable arguments. Just like above all feature helped to write clean code in Java, I am sure lambda expression will take it to next level. One of the thing I am expecting is development of parallel third-party libraries, which will make writing high performance application slightly easier than today.

[Share on email](#) [Share on delicious](#) [Share on linkedin](#) [Share on stumbleupon](#) [Share on reddit](#) [Share on digg](#) [Share on dzone](#) [Share on hackernews](#) [More Sharing Services](#)

You might like:

- [What is Race Condition in multithreading – 2 Examples in Java](#)
- [How to reverse String in Java using Iteration and Recursion](#)
- [REST Web services Framework interview questions answers](#)
- [15 Java NIO, Socket, and Networking Interview Questions Answers](#)

Recommended by

Posted by Javin Paul at [6:22 AM](#) 

[Email This Blog](#)[This!](#) [Share to Twitter](#) [Share to Facebook](#)

Labels: [core java](#), [Java 8](#), [Lambda expression](#), [programming](#)

#### 13 comments :

[Sushil nagpal](#) said...

I loved it.. Awesome Post buddy. Thanks :)

[February 26, 2014 at 10:33 PM](#)

[Blog4U](#) said...

Awesome revolutions in JAVA 8

[February 26, 2014 at 10:40 PM](#)

Anonymous said...



Thanks for this wonderful article, Most important thing, which I know I come to know is that if a method accept type of an interface, which has just one abstract method e.g. Comparator or Comparable, we can use a lambda expression instead of defining and then creating a new instance of a class that implements Comparator, and if that lambda expression does nothing but call an existing method, we can use method reference. This really helped me to understand where can we use lambda expression, and answered my long had doubt about why everyone talk about Comparable and Runnable when they talk Java 8 and lambdas.

[February 26, 2014 at 11:19 PM](#)

[Demelor](#) said...

Great article!

[February 27, 2014 at 9:13 PM](#)

[ella asley](#) said...

Thanks for sharing this information. Java program examples are quite useful for me. In General [Java Lambda Expressions and Streams](#) are one of the great feature added in Java 8. Lambda Expressions are used to represent functions and streams are nothing but wrappers, arrays and collections.

[February 27, 2014 at 10:50 PM](#)

[Vasi](#) said...

Oh no, this will make debugging a nightmare. Java was awesome because of it's simplicity; to bad it's loosing that...

[March 3, 2014 at 1:16 AM](#)

Anonymous said...

I have a question.

Suppose that an interface has 2 methods with the same signature but different names:

```
public void actionPerformed(ActionEvent e);
```

```
public void actionWhatever(ActionEvent e);
```

If you use:

```
show.addActionListener((e) -> { System.out.println("Light, Camera, Action !!
```

```
Lambda expressions Rocks"); });
```

How will it know which one of the two methods to replace with it?

[March 15, 2014 at 2:03 PM](#)

Anonymous said...

Nice article. clearly explained about lambda expressions. Thanks for posting. Great Job.

[March 17, 2014 at 12:03 PM](#)

[Brian van vlymen](#) said...

awesome article!!! I will be very exciting to learn something new!

[March 17, 2014 at 6:10 PM](#)

Anonymous said...

If an interface has more then one method, then that will not be a functional interface, and therefore will not be supportes by lambda expressions...

[March 19, 2014 at 1:57 PM](#)

[Amit Jain](#) said...

wonderful, perfect way to understand and learn. Thanks

[April 14, 2014 at 7:41 AM](#)

Anonymous said...

Better way to filter non null values in a List in Java 8 :

```
stockList.stream().filter(Objects::nonNull).forEach(o -> {...});
```

better use of static method reference instead of lambdas and functional interface.

[April 14, 2014 at 8:33 AM](#)

[rajesh](#) said...

Explained very well....

[May 8, 2014 at 9:56 PM](#)

## Post a Comment

[Newer Post](#) [Older Post](#) [Home](#)

Subscribe to: [Post Comments \( Atom \)](#)


## Best of Javarevisited

- [How Android works, Introduction for Java Programmers](#)
- [Difference between Java and Scala Programming](#)
- [Top 5 Java Programming Books for Developers](#)
- [Top 10 JDBC Best Practices for Java programmers](#)
- [Tips and Best practices to avoid NullPointerException in Java](#)

- [10 Object Oriented Design Principles Java Programmer Should Know](#)
- [10 HotSpot JVM Options, Every Java Programmer Should Know](#)

## Subscribe To This Blog Free

▼  Posts

▼  Comments

## Follow Us

## Followers

**Subscribe by email:**

[By Javin Paul](#)

## Blog Archive

- [▼ 2014](#) ( 88 )
  - [▶ October](#) ( 1 )
  - [▶ September](#) ( 9 )
  - [▶ August](#) ( 9 )
  - [▶ July](#) ( 8 )
  - [▶ June](#) ( 8 )
  - [▶ May](#) ( 11 )
  - [▶ April](#) ( 10 )
  - [▶ March](#) ( 11 )
  - [▼ February](#) ( 11 )
    - [Why Catching Throwable or Error is bad?](#)
    - [10 Example of Lambda Expressions and Streams in Ja...](#)
    - [How to Format and Display Number to Currency in Ja...](#)
    - [Fixing java.net.BindException: Cannot assign reque...](#)
    - [Top 30 Java Phone Interview Questions Answers for ...](#)
    - [How to Create Tabs UI using HTML, CSS, jQuery, JSP...](#)
    - [Why Static Code Analysis is Important?](#)
    - [Display tag Pagination, Sorting Example in JSP and...](#)
    - [Java Comparable Example for Natural Order Sorting](#)
    - [Difference between Association, Composition and Ag...](#)
    - [StringTokenizer Example in Java with Multiple Deli...](#)
  - [▶ January](#) ( 10 )
- [▶ 2013](#) ( 136 )
- [▶ 2012](#) ( 217 )

- [▶ 2011](#) ( 145 )
- [▶ 2010](#) ( 33 )

## References

- [Java API documentation JDK 6](#)
- [Spring framework doc](#)
- [Struts](#)
- [JDK 7 API](#)
- [MySQL](#)
- [Linux](#)
- [Eclipse](#)
- [jQuery](#)

Copyright by Javin Paul 2012. Powered by [Blogger](#).

- [About Me](#)
- [Privacy Policy](#)

## Searching for archives?

[archive](#)

Cliquez Ici Maintenant: [archive](#)

[ppcs.tlbsearch.com](http://ppcs.tlbsearch.com)

Read more: <http://javarevisited.blogspot.com/2014/02/10-example-of-lambda-expressions-in-java8.html#ixzz3FMd2a9ME>

E

## Exercices

### Exercice 13.1

=====

```
/**
 * Rôle : compose n fois la lambda f : f o f ... o f
 */
static <T> Function<T,T> composeNFoisR(Function<T,T> f, int n) {
    if (n==1) return f;
    return (x) -> f.apply(composeNFoisR(f,n-1).apply(x));
}
```

### Exercice 13.2

=====

```
interface Fonction2<T,R> {
    R apply(T x, T y);
}

static <T,R> R appliquer(Fonction2 <T,R> f, T ... args) {
    R r = f.apply(args[0], args[1]);
    for (int i=2; i<args.length; i++)
        r = f.apply(r, args[i]);
    return r;
}
```

### Exercice 13.3

=====

```
public static <T,R> Liste<R> map(Function<T,R> f, T ... args) {
    Liste<R> r = new ListeTableau<R>();
    int i=1;
    for (T x : args)
        r.ajouter(i++,f.apply(x));
    return r;
}

public static void main(String [] args) {
    System.out.println(map((x) -> x*x*x, 1, 2, 3, 4, 5 ));
}
```

### Exercice 13.5

=====

```
interface Fonc<R> {
    R apply(int x, int y);
}

public static <R> R fibCont(int n, Fonc<R> g) {
    if (n==2 || n==1)
        return g.apply(1,1);
    //
    return fibCont(n-1, (u, v) -> g.apply(v, v+u));
}

public static void main(String [] args) {
    for (int i=1; i<Integer.valueOf(args[0]); i++)
```

```

        System.out.print(fibCont(i, (u,v) -> v)+ " ");
        System.out.println();
    }

```

#### Exercice 13.7

=====

```

interface Func<R,T> {
    R eval(T ... x);
}

static <T> T [] append(T[] args, T ... x) {
    List<T> aux = new ArrayList<T>(args.length+x.length);
    for (T e : args) aux.add(e);
    for (T e : x) aux.add(e);
    return aux.toArray(args);
}

static <R,T> Func<R,T> curry_n(Func<R,T> f, T ... args) {
    return (T ... x) -> f.eval(append(args, x));
}

```

#### Exercice 13.8

=====

```

import java.util.function.*;
import java.io.*;

@FunctionalInterface
interface Fonc<T> {
    T apply();
}

class Générateur {

    public static void main(String [] args) throws IOException {

        Function<Integer,Fonc<Integer>> créerGénérateur = (x) -> {
            int [] cpt = { x };
            return () -> cpt[0]++;
        };

        Fonc<Integer> générateur = créerGénérateur.apply(3);

        for (int i=0; i<10; i++)
            System.out.print(générateur.apply() + " ");
        System.out.println();
    }
}

```

## [Java 8 : petit exercice pour s'échauffer le neurone à lambda](#)

[Java / JEE](#) › [Articles](#) | Tags : [fp](#), [java](#) Par [Olivier Croisier](#)

A une semaine de Devovx France 2014 qui risque d'être riche en sessions sur Java 8 et la programmation fonctionnelle, je vous propose un petit exercice pour vous dérouter le neurone à lambdas.

Le but du jeu est d'écrire une fonction permettant de concaténer un certain nombre de listes, passées en paramètre sous forme de *var-arg* :

```
public <T> List<T> concatLists(List<T>... lists);
```

Evidemment, on essaiera d'utiliser le plus possible de nouvelles fonctionnalités de Java 8 - le but est de s'amuser et de tordre un peu Java, et pas forcément de respecter les bonnes pratiques industrielles.

Allez hop, en route !

## Version 1 : foreach

Tout d'abord, comment écrirait-on cette méthode en Java "traditionnel" (Java <=7) ? A vue de nez, on bouclerait sur les listes à concaténer, et on déverserait le contenu de chaque liste dans une liste résultat initialement vide :

```
public <T> List<T> concatLists1(List<T>... lists) {
    ArrayList<T> result = new ArrayList<>();
    for (List<T> list : lists) {
        result.addAll(list);
    }
    return result;
}
```

Facile.

Mais, dites-moi, ce pattern "je boucle sur une collection, et j'applique une fonction à chaque boucle", ça ne vous rappelle rien ? Si, c'est un [Fold](#) ! Et en Java 8, cette opération est disponible sur les Streams, sous le nom de [reduce](#).

## Version 2 : Fold à la rescousse

Recette du Fold de Papy Lambda, pour 4 personnes :

- Un ensemble d'éléments sur lesquels itérer : le *var-arg* en entrée de la méthode
- Une valeur initiale pour l'accumulateur (identité) : une liste vide
- Une opération à appliquer à chaque étape

Pour ce dernier point, `reduce()` attend en paramètre un [java.util.function.BinaryOperator<T>](#). Cette drôle d'interface n'est en réalité qu'un synonyme pour [java.util.function.BiFunction<T, T, T>](#), qui représente une fonction acceptant deux paramètres de type `T` (l'accumulateur et l'élément en cours d'itération), et renvoyant un résultat du même type `T` (la nouvelle valeur de l'accumulateur).

C'est sous cette forme que nous devons exprimer notre opération de réduction, basée sur `List.addAll()` comme précédemment.



Voyons comment ça se présente :

```
public <T> List<T> concatLists2(List<T>... lists) {  
    return Stream.of(lists).reduce(new ArrayList<>(), new BinaryOperator<List<T>>() {  
        @Override  
        public List<T> apply(List<T> list1, List<T> list2) {  
            list1.addAll(list2);  
            return list1;  
        }  
    });  
}
```

C'est beau, non ?

Bon d'accord, ça fonctionne, c'est plus "fonctionnel", mais c'est tout de même moins lisible que la simple boucle - la faute au `BinaryOperator` implémenté sous forme de classe anonyme, et aux Generics qui s'invitent à tous les étages.

Peut-on faire mieux ?

### Version 3 : Viva la lambda-da

`BinaryOperator` étant une interface fonctionnelle ([@FunctionalInterface](#)), il est possible de l'exprimer sous la forme d'une expression lambda.

Adieu donc toute la tuyauterie des classes anonymes ! Seule la substantifique moelle de l'algorithme est conservée :

```
public <T> List<T> concatLists3(List<T>... lists) {  
    return Stream.of(lists).reduce(new ArrayList<>(), (list1, list2) -> {  
        list1.addAll(list2);  
        return list1;  
    });  
}
```

Le code ainsi obtenu est même plus court que notre boucle for-each initiale !

Mais on a dit qu'on était là pour utiliser un maximum de nouveautés de Java 8 sur ce petit exercice. Que peut-on rajouter ? Les références de méthodes ?

Challenge accepted.

### Version 4 : Pour bien faire la référence

Java 8 propose maintenant une forme de "duck typing"<sup>[1]</sup> : partout où une interface fonctionnelle est attendue en paramètre d'une méthode, il est possible de fournir une référence vers une méthode existante proposant une signature compatible.

Par exemple, la méthode [Iterable.forEach\(\)](#) prend en paramètre un [java.util.function.Consumer<T>](#), qui définit une méthode acceptant un unique paramètre de type `T` et ne renvoyant rien.

La méthode `System.out.println()` correspond justement à cette définition ; on peut donc

l'assimiler à un `Consumer`, et donc l'utiliser comme ceci - notez l'utilisation du double deux-points (`::`) :

```
Arrays.asList(1,2,3).forEach(System.out::println);
```

Dans notre cas, `reduce()` attend en paramètre un `BinaryOperator`, qui définit une méthode acceptant deux paramètres de même type, et renvoyant un résultat également du même type.

Malheureusement, la méthode `List.addAll()` ne correspond pas à cette définition, car elle renvoie un booléen. Mais nous pouvons écrire une méthode statique possédant la bonne signature, encapsulant l'appel à `List.addAll()` :

```
public static <E> List<E> concatLists(List<E> list1, List<E> list2) {  
    list1.addAll(list2); // Ignore return value  
    return list1;  
}
```

Et l'utiliser ainsi dans le fold :

```
public List<Integer> concatLists4(List<Integer>... lists) {  
    return Stream.of(lists).reduce(new ArrayList<>(), MyClass::concatLists);  
}
```

Victoire ! Un one-liner !

Enfin presque - il faut tout de même écrire une méthode qui se conforme à l'interface `BinaryOperator`, ce qui ne fait que déplacer le problème.  
Au final, cette solution semble moins satisfaisante qu'une simple lambda...

Mais... c'est sans compter sur le **Fluentizer**©® 3000 !

## Version 5 : Fluentizer 3000 fait étinceler vos interfaces

En fait, ce qui nous empêche de passer directement une référence vers `List.addAll()`, c'est que cette méthode renvoie un booléen au lieu de renvoyer une autre liste. Vilaine, vilaine méthode. Et c'est d'autant plus ballot que de toute façon on ignore le booléen qu'elle renvoie !

Pourrait-on imaginer un système qui transformerait une méthode de signature  $(T, T) \rightarrow R$  (autrement dit, une `BiFunction<T, T, R>`) en méthode compatible avec l'interface fonctionnelle `BinaryOperator (T, T) -> T` ?

Allez hop, sous vos yeux ébahis :

```
public static <T> BinaryOperator<T> fluentize(BiFunction<T, T, ?> op) {  
    return (t1, t2) -> {  
        op.apply(t1, t2); // drop result  
        return t1;  
    };  
}
```

Oui, c'est moche, mais ça permet maintenant d'écrire ceci :

```
public List<T> concatLists5(List<T>... lists) {  
    return Stream.of(lists).reduce(new ArrayList<>(), fluentize(List::addAll));  
}
```

N'est-ce pas le bonheur suprême ?

## Exercice 1 - Le compte est bon

Le but de cet exercice est de découvrir comment utiliser des lambdas et des streams.

1. On cherche à compter le nombre d'occurrence d'un mot dans une liste.
2. 

```
List<String> list = Arrays.asList("hello", "world", "hello", "lambda");
```
3. 

```
System.out.println(count(list, "hello"));
```

Ecrire le code de la méthode `count` sachant que le compteur est un entier long.

4. On cherche à écrire une méthode `count2` sémantiquement équivalente (qui fait la même chose) que `count` mais en utilisant l'API des [Stream](#).  
Comment obtenir un `Stream` à partir d'un objet de type `List` ?  
L'idée, ici, est de filtrer le stream pour ne garder que les mots égaux au mot passé en paramètre puis de compter ceux-ci.  
Quelle est le nom des méthodes permettant respectivement de filtrer un stream et de compter le nombre d'élément ?  
La méthode qui permet de filtrer prend un objet de type `Predicate<T>` en paramètre. Dans notre cas, quelle est le type correspondant à `T` ?  
Indiquer le code permettant de créer une lambda filtrant sur le mot passé en paramètre que l'on peut déclarer en tant que `Predicate`  
Ecrire le code de la méthode `count2`.
5. Il est possible d'utiliser la syntaxe des méthodes références au lieu de celle des lambdas pour implanter le filtre.  
Modifier le code pour utiliser la syntaxe des méthodes références.

## Exercice 2 - En majuscule

Le but de cet exercice est de découvrir comment utiliser en plus des lambdas et des streams, des méthodes références.

1. On cherche écrire une méthode prenant en paramètre une liste de chaîne de caractères et renvoyant une nouvelle liste contenant les chaînes de caractère en majuscule.
2. 

```
List<String> list = Arrays.asList("hello", "world", "hello", "lambda");
```
3. 

```
System.out.println(upperCase(list));
```

Ecrire la méthode `upperCase` dans un premier temps sans utiliser l'API des `Stream`.

4. On cherche maintenant à écrire une méthode `upperCase2` faisant la même chose.  
Comment peut-on utiliser la méthode `Stream.map` ici ?  
Pour stocker le résultat dans une nouvelle liste, l'idée est de créer la liste pour d'ajouter chaque mot dans la liste.
5. 

```
public static List<String> upperCase2(List<String> words) {
```
6. 

```
    ArrayList<String> uppercases = new ArrayList<>();
```
7. 

```
    ...
```

pour demander l'ajout, on utilisera sur le stream la méthode `forEach`.

Ecrire le code de la méthode `upperCase2` en utilisant des lambdas.

8. En fait, au lieu d'utiliser des lambdas, il est possible dans cet exemple d'utiliser la syntaxe des références de méthodes avec l'opérateur `::` (coloncolon).  
Ecrire une méthode `upperCase3` qui utilise la syntaxe des références de méthodes.
9. En fait, au lieu d'utiliser `forEach`, il est plus pratique d'utiliser la méthode `collect` avec comme [Collector](#) celui renvoyé par la méthode [Collectors.toList\(\)](#).  
Ecrire une méthode `upperCase4` en utilisant le collector `Collectors.toList()`.

### Exercice 3 - Comptons sur une réduction

Le but de cet exercice est de découvrir comment utiliser effectuer une réduction sur un stream.  
Lors du premier exercice, nous avons utilisé la méthode `count` qui retourne un entier long, on souhaite maintenant écrire une nouvelle méthode `count3` qui renvoie un entier sur 32 bits.  
Pour cela, une fois les mots filtrés, nous allons transformer (avec `map`) chaque mot en 1 (le nombre) puis nous allons avec la méthode `reduce` faire l'aggrégation des valeurs.

1. Expliquer pourquoi, nous n'allons pas utiliser la méthode `map` mais la méthode `mapToInt` ?
2. Ecrire le code de la méthode `count3`.

Si vous vous sentez perdu dans cet exercice, vous pouvez aller regarder comment la méthode `Stream.count()` que nous avons vu au premier exercice est implantée (un Ctrl+click sur une méthode dans Eclipse charge son code !).

### Exercice 4 - Evaluation de vitesse

On cherche à savoir parmi les 3 façons d'écrire `count` quelle est la plus rapide.  
Nous allons pour cela utiliser une liste définie par le code suivant

```
List<String> list2 =  
    new Random(0)  
        .ints(1_000_000, 0, 100)  
        .mapToObj(Integer::toString)  
        .collect(Collectors.toList());
```

1. Expliquer ce que contient la variable locale `list2`.
2. On cherche à écrire une méthode `printAndTime` permettant de calculer le temps d'exécution de l'appel à la méthode `count` sur la `list2`.  
Pour calculer le temps d'exécution, on demande le temps avant puis le temps après et si l'on soustrait les deux temps, on trouve le temps d'exécution.
3. `long start = System.nanoTime();`
4. `... // faire le calcul`
5. `long end = System.nanoTime();`
6. `System.out.println("result " + result);`
7. `System.out.println(" elapsed time " + (end - start));`

Tester ce code en appelant la méthode `count()`.

8. On cherche maintenant A NE PAS dupliquer le code ci-dessus dans le cas où l'on appelle `count2` ou `count3` mais à utiliser une méthode `printAndTime` et de passer en paramètre la méthode `count` qui doit être appelée.  
Quelle doit être la signature de la méthode de l'interface fonctionnel tel que l'on puisse appeler la méthode `printAndTime` comme ceci:
9. `printAndTime() -> count(list2, "33");`
10. `printAndTime() -> count2(list2, "33");`
11. `printAndTime() -> count3(list2, "33");`

Déclarer l'interface et écrivez le nouveau code de `printAndTime`.

12. En fait, il existe déjà une interface [LongSupplier](#), modifier le code de `printAndTime` pour utiliser cette interface.
13. Expliquer les résultats obtenus.  
Que se passe-t'il si on permute les appels à `printAndTime` ?

A screenshot of a Windows 8 taskbar. The taskbar is dark grey and contains several application icons: the Start button (Windows logo), a folder icon, a game controller icon, a colorful square icon, a Firefox icon, a Chrome icon, a document icon, a calendar icon, and a purple icon with a crown. On the right side of the taskbar, there is a search icon (magnifying glass), a help icon (question mark), and a system tray area showing the date and time as 19:54 on 25/10/2014.