

Java 8

Licence d'informatique 3^{ème} année

Java 8

Lambda

Sélectionner des pommes

```
public static List<Pomme> filtrerPommeVertes(List<Pomme> pommes) {  
  
    List<Pomme> resultat = new ArrayList<>();  
    for(Pomme pomme: pommes){  
        if( "vert".equals(pomme.getCouleur()) ) {  
            resultat.add(pomme);  
        }  
    }  
    return resultat;  
}
```

Que faire pour sélectionner les pommes rouges ?

Sélectionner des pommes

```
public static List<Pomme> filtrerPommesParCouleur(List<Pomme> pommes
                                                    , String couleur) {
    List<Pomme> resultat = new ArrayList<>();
    for(Pomme pomme: pommes){
        if( couleur.equals(pomme.getCouleur()) ) {
            resultat.add(pomme);
        }
    }
    return resultat;
}
```

Que faire pour obtenir les pommes de plus de 150 grammes ?

Sélectionner des pommes

```
public static List<Pomme> filtrerPommesParPoids(List<Pomme> pomme
                                                , int poids) {
    List<Pomme> resultat = new ArrayList<>();
    for(Pomme pomme: pommes){
        if( pomme.getPoids() > poids) {
            resultat.add(pomme);
        }
    }
    return resultat;
}
```

On ne dupliquerait pas du code ?

Sélectionner des pommes

```
public interface CriterePomme {  
    boolean test(Pomme p);  
}
```

```
public class CriterePommeVerte implements CriterePomme {  
    public boolean test(Pomme p) {  
        return p.getCouleur().equals("vert");  
    }  
}
```

```
public static List<Pomme> filtrerPomme(List<Pomme> lp, CriterePomme c){  
  
    List<Pomme> resultat = new ArrayList<Pomme>();  
    for(Pomme p : lp){  
        if(c.test(p))  
            resultat.add(p);  
    }  
    return resultat;  
}
```

Et si on veut sélectionner des oranges ?

Sélectionner des pommes

```
public interface Critere<A> {  
    boolean test(A a);  
}
```

```
public class CriterePommeVerte implements Critere<Pomme> {  
    public boolean test(Pomme p) {  
        return p.getCouleur().equals("vert");  
    }  
}
```

```
public static <A> List<A> filtrer(List<A> lp, Critere<A> c) {  
  
    List<A> resultat = new ArrayList<A>();  
    for(A p : lp) {  
        if(c.test(p))  
            resultat.add(p);  
    }  
    return resultat;  
}
```

Mais alors il faut créer une classe pour chaque critère ?

Sélectionner des pommes

```
public static <A> List<A> filtrer(List<A> lp, Critere<A> c) {  
  
    List<A> resultat = new ArrayList<A>();  
    for(A p : lp){  
        if(c.test(p))  
            resultat.add(p);  
    }  
    return resultat;  
}
```

```
Liste<Pomme> grossesPommes = Utils.filtrer(pommes, new Critere<Pomme>() {  
    public boolean test(Pomme p) {  
        return p.getPoids() > 100;  
    }  
});
```

C'est encore un peu verbeux !!!

Sélectionner des pommes

```
public static <A> List<A> filtrer(List<A> lp, Critere<A> c) {  
  
    List<A> resultat = new ArrayList<A>();  
    for(A p : lp) {  
        if(c.test(p))  
            resultat.add(p);  
    }  
    return resultat;  
}
```

```
Liste<Pomme> grossesPommes = Utils.filtrer(pommes,  
                                           (Pomme p) -> p.getPois() > 100);  
  
Liste<Pomme> pommesVertes = Utils.filtrer(pommes,  
                                           (Pomme p) -> p.getColor().equals("vert"));
```

Trier des pommes

```
public interface Comparator<T> {  
    public int compare(T o1, T o2);  
}
```

Classe Anonyme

```
pommes.sort(new Comparator<Pomme>() {  
    public int compare(Pomme p1, Pomme p2){  
        return p1.getPoids().compareTo(p2.getPoids());  
    }  
});
```

Lambda

```
pommes.sort( (Pomme p1, Pomme p2) ->  
    p1.getPoids().compareTo(p2.getPoids()) );
```

Lambda expression

- Une lambda expression peut être considérée comme une représentation concise d'une fonction anonyme qui peut être passée comme paramètre :
 - ❑ Anonyme : pas de nom
 - ❑ Fonction : une lambda n'est pas associée à une classe
 - ❑ Passée comme paramètre : une lambda peut être passée en paramètre, retournée par une méthode ou stockée dans une variable
 - ❑ Concise : moins verbeuse que l'utilisation d'une classe anonyme.

Lambda expression

Syntaxe

```
(paramètres) -> expression  
ou  
(paramètres) -> { instructions; }
```

Exemples

```
(String s) -> s.length()
```

```
(Pomme p) -> a.getPoids() > 150
```

```
(int x, int y) -> {  
    System.out.println(" Result:  ");  
    System.out.println(x+y);  
}
```

```
() -> 42
```

```
(Pomme p1, Pomme p2) -> p1.getPoids().compareTo(p2.getPoids())
```

Lambda expression : Quiz

- `() -> {}`
- `() -> "Ray"`
- `() -> {return "Connor";}`
- `(Integer i) -> return "Abby" + i;`
- `(String s) -> {"Bridget";}`

Lambda expression : Quiz

- ✓ `() -> {}`
- ✓ `() -> "Ray"`
- ✓ `() -> {return "Connor";}`
- `(Integer i) -> return "Abby" + i;`
- `(String s) -> {"Bridget";}`

Exemples de lambdas

Cas d'usage	Exemples de lambdas
Expression booléenne	<code>(List<String> list) -> list.isEmpty()</code>
Créer un objet	<code>() -> new Pomme(10)</code>
Consommer un objet	<code>(Pomme p)-> { System.out.println(p.getPoids()); }</code>
Sélectionner/Extraire depuis un objet	<code>(Pomme p) -> p.getPoids()</code>
Combiner deux valeurs	<code>(int a, int b) -> a * b</code>
Comparer deux valeurs	<code>(Pomme p1, Pomme p2) -> p1.getPoids().compareTo(p2.getPoids())</code>

Où est-il possible d'utiliser des lambdas ?

En paramètre d'une méthode

```
Utils.filtrer(pommes, (Pomme p) -> p.getPois() > 100);
```

```
pommes.sort( (Pomme a1, Pomme a2) ->  
             p1.getPoids().compareTo(p2.getPoids()) );
```

Assignée à une variable

```
Critere<Pomme> crPomme = (Pomme p) -> p.getPois() > 100;
```

```
Comparator<Pomme> comp = (Pomme a1, Pomme a2) ->  
                          p1.getPoids().compareTo(p2.getPoids());
```

Critère et Comparator sont des interfaces fonctionnelles

Interface fonctionnelle

- Une interface fonctionnelle est une interface qui déclare une et une seule méthode abstraite.
- Il est possible d'utiliser des lambdas partout où sont attendues des interfaces fonctionnelles.

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

```
public interface Critere<T> {  
    boolean test(T t);  
}
```

```
public interface Predicate<T> {  
    boolean test(T t);  
}
```

Interface fonctionnelle : Quiz

- ```
public interface Adder {
 int add(int a, int b);
}
```
- ```
public interface SmartAdder extends Adder  
{  
    int add(double a, double b);  
}
```
- ```
public interface Nothing{
}
```

# Interface fonctionnelle : Quiz

- ✓ 

```
public interface Adder {
 int add(int a, int b);
}
```
- ```
public interface SmartAdder extends Adder  
{  
    int add(double a, double b);  
}
```
- ```
public interface Nothing{
}
```

---

# Descripteur de fonction

- La signature de la méthode abstraite de l'interface fonctionnelle décrit la signature de la lambda expression.
- Cette méthode abstraite est appelée descripteur de fonction.
- Une notation spéciale est utilisée pour définir la signature de la lambda.
  - `(Pomme, Pomme) -> int` dénote une fonction qui prend 2 pommes en paramètre et retourne un entier
  - `() -> void` dénote une fonction qui ne prend pas de paramètre et ne retourne rien.

---

# Descripteur de fonction

- Une lambda expression peut être assignée à une variable ou passée à une méthode attendant une interface fonctionnelle comme paramètre, si et seulement si la lambda expression a la même signature que la méthode abstraite de l'interface fonctionnelle.
- Dans l'API Java, les interfaces fonctionnelles (comme Predicate ou Comparator) sont annotées avec `@FunctionalInterface`). Si l'interface annotée n'est pas une interface fonctionnelle le compilateur produira un message d'erreur :
  - "Multiple overriding abstract methods found in interface"

---

# Interface fonctionnelle

- Afin de pouvoir utiliser différentes lambda expressions, l'API Java fournit un ensemble d'interfaces fonctionnelles décrivant les descripteurs de fonction les plus courants.
  - `package java.util.function`
  - `ex : Predicate, Consumer, Function, etc.`

# Predicate

- `java.util.function.Predicate<T>`
- Définit une méthode abstraite `test` qui accepte un objet du type générique `T` et retourne un booléen

```
@FunctionalInterface
public interface Predicate<T> {
 boolean test(T t);
}

public static <T> List<T> filtrer(List<T> list, Predicate<T> p) {
 List<T> results = new ArrayList<>();
 for(T s: list){
 if(p.test(s)){
 results.add(s);
 }
 }
 return results;
}
```

```
Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();
List<String> nonEmpty = filtrer(listOfStrings, nonEmptyStringPredicate);
```

# Consumer

- `java.util.function.Consumer<T>`
- Définit une méthode abstraite `accept` qui prend un objet du type générique `T` et ne retourne rien (`void`)

```
@FunctionalInterface
public interface Consumer<T>{
 void accept(T t);
}

public static <T> void forEach(List<T> list, Consumer<T> c){
 for(T i: list){
 c.accept(i);
 }
}
```

```
forEach(Arrays.asList(1,2,3,4,5),
 (Integer i) -> System.out.println(i));
```



# Function

- `java.util.function.Function<T, R>`
- Définit une méthode abstraite `apply` qui prend un objet du type générique `T` et retourne un objet du type générique `R`.

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

public static <T, R> List<R> map(List<T> list, Function<T, R> f{
 List<R> result = new ArrayList<>();
 for(T s: list){
 result.add(f.apply(s));
 }
 return result;
}
```

```
List<Integer> l = map(
 Arrays.asList("després", "est", "génial"), (String s) -> s.length());
```

# Spécialisation des primitives

- Java 8 propose des interfaces fonctionnelles spécialisées pour les types primitifs afin d'éviter les opérations d'autoboxing qui ont un coût.

```
public interface IntPredicate{
 boolean test(int t);
}
```

```
IntPredicate evenNumbers = (int i) -> i % 2 == 0;
evenNumbers.test(1000);
```

```
Predicate<Integer> oddNumbers = (Integer i) -> i % 2 == 1;
oddNumbers.test(1000);
```

# Interfaces fonctionnelles de l'API Java 8

| Interface Fonctionnelle | Descripteur de fonction | Primitives spécialisées                                                                                                                                                                       |
|-------------------------|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Predicate<T>            | T -> boolean            | IntPredicate, LongPredicate, DoublePredicate                                                                                                                                                  |
| Consumer<T>             | T -> void               | IntConsumer, LongConsumer, DoubleConsumer                                                                                                                                                     |
| Function<T, R>          | T -> R                  | IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T> |
| Supplier<T>             | () -> T                 | BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier                                                                                                                                    |
| UnaryOperator<T>        | T -> T                  | IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator                                                                                                                                      |

# Interfaces fonctionnelles de l'API Java 8

| Interface Fonctionnelle                  | Descripteur de fonction | Primitives spécialisées                                                 |
|------------------------------------------|-------------------------|-------------------------------------------------------------------------|
| BinaryOperator<T>                        | (T, T) -> T             | IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator             |
| BiPredicate<L, R>                        | (L, R) -> boolean       |                                                                         |
| BiConsumer<T, U>                         | (T, U) -> void          | ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>             |
| BiFunction<T, U, R>                      | (T, U) -> R             | ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U> |
| Christophe Després - Université du Maine |                         |                                                                         |

# Interfaces fonctionnelles : Quiz

■  $T \rightarrow R$

Function<T, R>

■  $(\text{int}, \text{int}) \rightarrow \text{int}$

IntBinaryOperator

■  $T \rightarrow \text{void}$

Consumer<T>

■  $() \rightarrow T$

Supplier<T>

■  $(T, U) \rightarrow R$

BiFunction<T, U, R>

# Exemples de lambdas

| Cas d'usage                                  | Exemples de lambdas                                                                                                                               | Interface Fonctionnelle                                                                                                                                   |
|----------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| Expression booléenne                         | <code>(List&lt;String&gt; list) -&gt; list.isEmpty()</code>                                                                                       | <code>Predicate&lt;List&lt;String&gt;&gt;</code>                                                                                                          |
| Créer un objet                               | <code>() -&gt; new Pomme(10)</code>                                                                                                               | <code>Supplier&lt;Pomme&gt;</code>                                                                                                                        |
| Consommer un objet                           | <code>(Pomme p)-&gt; {<br/>    System.out.println(p.getPoids());<br/>}</code>                                                                     | <code>Consumer&lt;Pomme&gt;</code>                                                                                                                        |
| Sélectionner/<br>Extraire depuis<br>un objet | <code>(String s) -&gt; s.length()</code>                                                                                                          | <code>Function&lt;String, Integer&gt;</code><br><code>ToIntFunction&lt;String&gt;</code>                                                                  |
| Combiner deux<br>valeurs                     | <code>(int a, int b) -&gt; a * b</code>                                                                                                           | <code>IntBinaryOperator</code>                                                                                                                            |
| Comparer deux<br>valeurs                     | <code>(Pomme p1, Pomme p2) -&gt;<br/>p1.getPoids().compareTo(p2.getPoids())</code><br><br><small>Christophe Després - Université du Maine</small> | <code>Comparator&lt;Pomme&gt;</code><br><code>BiFunction &lt;Pomme,<br/>Pomme, Integer&gt;</code><br><code>ToIntBiFunction&lt;Pomme,<br/>Pomme&gt;</code> |

# Références de méthodes

- Les références de méthodes permettent de réutiliser des méthodes définies en lieu et place de lambdas.
- Cela permet d'obtenir un code plus lisible et concis.

```
filtrer(pommes, (Pomme p1) -> p1.estMure());
```

```
filtrer(pommes, (Pomme::estMure));
```

- Quand vous utilisez une référence de méthode, la référence cible (la classe) est placée avant le délimiteur :: et le nom de la méthode est fourni après.

# Références de méthodes

- Il y a 3 types de références de méthodes :
  - Une référence à une méthode statique (ex: `parseInt` de la classe `Integer`)

```
(String s) -> Integer.parseInt(s)
```

```
Integer::parseInt
```

- Une référence à une méthode d'instance d'une classe (ex: `getPoids` de la classe `Pomme`)

```
(Pomme p) -> p.getPoids()
```

```
Pomme::getPoids
```

- Une référence à une méthode d'instance d'un objet (ex: `getPoids` de la pomme `p1`)

```
() -> p1.getPoids()
```

```
p1::getPoids
```



# Références de constructeurs

```
public class Pomme {
 private int poids;
 private String color;

 public Pomme() {
 this(0, null);
 }

 public Pomme(int poids) {
 this(poids, null);
 }

 public Pomme(String couleur) {
 this(0, couleur);
 }

 public Pomme(int poids, String color) {
 super();
 this.poids = poids;
 this.color = color;
 }
}
```

# Références de constructeurs

## Constructeur sans argument

```
public static void main(String[] args) {
 Supplier<Pomme> cp = Pomme::new;
 Pomme p01 = cp.get();
 System.out.println(p01.toString());
}
```



Pomme [poids=0, color=null]

## Constructeur à un argument

```
public static void main(String[] args) {
 Function<Integer, Pomme> cpp = Pomme::new;
 Pomme p02 = cpp.apply(130);
 System.out.println(p02.toString());
}
```




Pomme [poids=130, color=null]

# Références de constructeurs

## Constructeur à un argument


```
public static void main(String[] args) {
 Function<String, Pomme> ccp = Pomme::new;
 Pomme p03 = ccp.apply("rouge");
 System.out.println(p03.toString());
}
```



Pomme [poids=0, color=rouge]

## Constructeur à deux arguments

```
public static void main(String[] args) {
 BiFunction<Integer, String, Pomme> cpcp = Pomme::new;
 Pomme p04 = cpcp.apply(90, "verte");
 System.out.println(p04.toString());
}
```



Pomme [poids=90, color=verte]

# Références de constructeurs

## Constructeur sans argument

```
public static void main(String[] args) {
 Supplier<Pomme> cp = () -> new Pomme();
 Pomme p01 = cp.get();
 System.out.println(p01.toString());
}
```

## Constructeur à un argument

```
public static void main(String[] args) {
 Function<String, Pomme> ccp =
 (couleur) -> new Pomme(couleur);
 Pomme p03 = ccp.apply("rouge");
 System.out.println(p03.toString());
}
```

## Constructeur à deux arguments

```
public static void main(String[] args) {
 BiFunction<Integer, String, Pomme> cpcp =
 (poids, couleur) -> new Pomme(poids, couleur);
 Pomme p04 = cpcp.apply(90, "verte");
 System.out.println(p04.toString());
}
```

# Construction de pommes à la chaîne

```
public static List<Pomme> map(List<Integer> list,
 Function<Integer, Pomme> f){
 List<Pomme> result = new ArrayList<>();
 for(Integer e: list){
 result.add(f.apply(e));
 }
 return result;
}

public static void main(String[] args) {
 List<Integer> poids= Arrays.asList(7, 3, 4, 10);
 List<Pomme> pommes= map(poids, Pomme::new);
}
```

# Lambdas et références de méthodes

- L'API Java 8 fournit une méthode de tri aux listes.

```
public void sort(Comparator<? super E> c);
```

- Il faut lui fournir une stratégie de comparaison des éléments deux à deux.

```
public Class PommeComparator implements Comparator<Pomme>{
 public int compare(Pomme p1, Pomme p2){
 return p1.getPoids().compareTo(p2.getPoids());
 }
}
```

# Lambdas et références de méthodes

- Pour ne pas avoir à créer de classe pour chaque type de comparaison, on peut utiliser une classe anonyme

```
pommes.sort(new Comparator<Pomme>() {
 public int compare(Pomme p1, Pomme p2){
 return p1.getPoids().compareTo(p2.getPoids());
 }
});
```

- Ou mieux, une lambda expression

```
pommes.sort((Pomme p1, Pomme p2) ->
 p1.getPoids().compareTo(p2.getPoids()));
```

# Lambdas et références de méthodes

- L'interface `Comparator` dispose d'une méthode statique qui prend une fonction d'extraction d'un élément `Comparable` et retourne un objet `Comparator`

```
Comparator<Pomme> c = Comparator.comparing((Pomme p) -> p.getPoids());
```

- Et grâce à l'import statique...

```
pommes.sort(comparing((Pomme p)->p.getPoids()));
```

- Cerise sur le gâteau : référence de méthode

```
pommes.sort(comparing(Pomme::getPoids));
```



# Combiner les lambdas

- Les interfaces fonctionnelles comme `Comparator`, `Function` ou `Predicate` dispose de méthodes permettant de combiner des lambdas.

## Exemples pour Predicate

```
Predicate<Pomme> pommeRouge =
 (Pomme p) -> p.getCouleur().equals("rouge");

Predicate<Pomme> pommePasRouge = pommeRouge.negate();

Predicate<Pomme> grossePommeRouge =
 pommeRouge.and((Pomme p) -> p.getPoids()>150);

Predicate<Pomme> rougeOuVerte =
 pommeRouge.or((Pomme p) -> p.getCouleur().equals("vert"));
```

# Combiner les lambdas

## Exemples pour Comparator

```
pommes.sort (comparing (Pomme::getPoids) .reversed()) ;
```

```
Pommes.sort (comparing (Pomme::getPoids)
 .reversed()
 .thenComparing (Apple::getCouleur)) ;
```

## Exemples pour Function

```
Function<Integer, Integer> f = x -> x + 1;
```

```
Function<Integer, Integer> g = x -> x * 2;
```

```
Function<Integer, Integer> h = f.andThen(g);
```

```
int resultat = h.apply(1); // resultat = 4
```

```
h = f.compose(g);
```

```
resultat = h.apply(1); // resultat = 3
```

# Méthodes par défaut

- Depuis java 8 les interfaces peuvent comporter des implémentations. Il s'agit d'implémentations par défaut, qui seront utilisées si la méthode n'est pas redéfinie.

```
public interface Foo {
 public default void foo() {
 System.out.println("Default implementation of foo()");
 }
}
```

# Méthodes par défaut

```
public interface Itf {

 /** Pas d'implémentation - comme en Java 7 et antérieur */
 public void foo();

 // Implémentation par défaut, qu'on surchargera dans la classe fille
 public default void bar() {
 System.out.println("Itf -> bar() [default]");
 }


 // Implémentation par défaut, non surchargée dans la classe fille
 public default void baz() {
 System.out.println("Itf -> baz() [default]");
 }

}
```

# Méthodes par défaut

```
public class Cls implements Itf {
 @Override
 public void foo() {
 System.out.println("Cls -> foo()");
 }
 @Override
 public void bar() {
 System.out.println("Cls -> bar()");
 }
}
```

```
public class Test {
 public static void main(String[] args) {
 Cls cls = new Cls();
 cls.foo();
 cls.bar();
 cls.baz();
 }
}
```



Cls -> foo()  
Cls -> bar()  
Itf -> baz() [default]

# Exemple : Comparable et Orderable

```
public interface Orderable<T> extends Comparable<T> {

 // La méthode compareTo() est définie
 // dans la super-interface Comparable

 public default boolean isAfter(T other) {
 return compareTo(other) > 0;
 }

 public default boolean isBefore(T other) {
 return compareTo(other) < 0;
 }

 public default boolean isSameAs(T other) {
 return compareTo(other) == 0;
 }
}
```

# Héritage multiple

```
public interface InterfaceA {
 public default void foo() {
 System.out.println("A -> foo()");
 }
}

public interface InterfaceB {
 public default void foo() {
 System.out.println("B -> foo()");
 }
}

private class Test implements InterfaceA, InterfaceB {
 // Erreur de compilation : "class Test inherits unrelated defaults
 // for foo() from types InterfaceA and InterfaceB"
}
```

- Une erreur de compilation nous indique que la classe Test comporte deux implémentations de la même méthode.

# Héritage multiple

- Pour résoudre le conflit, une seule solution : redéfinir la méthode au niveau de la classe

```
public class Test implements InterfaceA, InterfaceB {
 public void foo() {
 System.out.println("Test -> foo()");
 }
}
```

- Il est possible, dans la redéfinition au niveau de la classe, d'invoquer une des implémentations des interfaces avec la syntaxe :

<Interface>.super.<méthode>

```
public class Test implements InterfaceA, InterfaceB {
 public void foo() {
 InterfaceB.super.foo();
 }
}
```



# Java 8

## *Stream*

# Régime basse calorie

```
List<Menu> menusBasseCalorie = new ArrayList<>();
for(Menu m: menus){
 if(m.getCalories() < 400){
 menusBasseCalorie.add(m);
 }
}

Collections.sort(menusBasseCalorie , new Comparator<Menu>() {
 public int compare(Menu m1, Menu m2){
 return Integer.compare(m1.getCalories(), m2.getCalories());
 }
});

List<String> nomsBasseCalorie= new ArrayList<>();
for(Menu m: menusBasseCalorie){
 nomsBasseCalorie.add(m.getNom());
}
```

**menusBasseCalorie est une variable intermédiaire**

# Régime basse calorie

```
import static java.util.Comparator.comparing;
import static java.util.stream.Collectors.toList;

List<String> nomsBasseCalorie =
 menus.stream()
 .filter(m -> m.getCalories() < 400)
 .sorted(comparing(Menu::getCalories()))
 .map(Menu::getNom)
 .collect(toList());
```

Et pour exploiter au mieux votre architecture multi-cœurs...

```
import static java.util.Comparator.comparing;
import static java.util.stream.Collectors.toList;

List<String> nomsBasseCalorie =
 menus.parallelStream()
 .filter(m -> m.getCalories() < 400)
 .sorted(comparing(Menu::getCalories()))
 .map(Menu::getNom)
 .collect(toList());
```

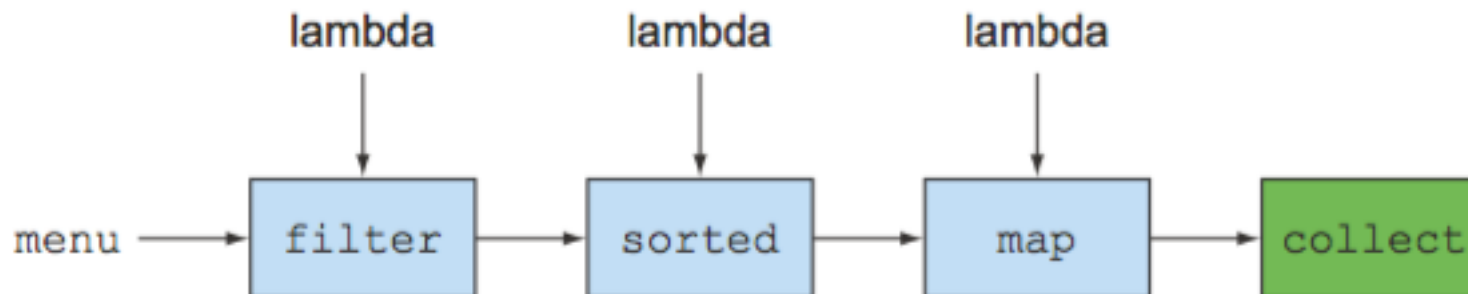
---

# Quelles différences ?

- Le code est écrit de façon déclarative
  - Vous spécifiez ce que vous voulez faire
    - Filtrer les menus peu caloriques
    - Les trier selon le nombre de calories
    - Extraire leur nom
    - Empaqueter le tout dans une liste
  - Vous ne spécifiez pas comment le faire
    - Pas de structures de contrôles
  - Le comportement est paramétré par des lambdas
    - Il est simple d'obtenir les menus fortement caloriques sans dupliquer de code

# Quelles différences ?

- Les traitements sont chaînés
  - ❑ Le résultat de filter est l'entrée de sorted
  - ❑ Le résultat de sorted est l'entrée de map
  - ❑ Le résultat de map est l'entrée de collect



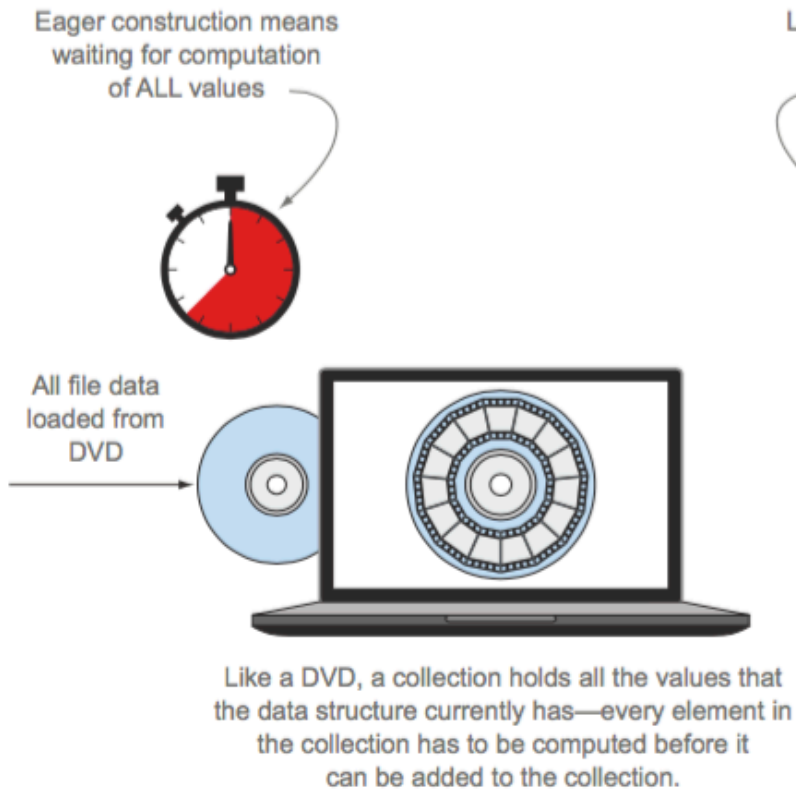
---

# Stream

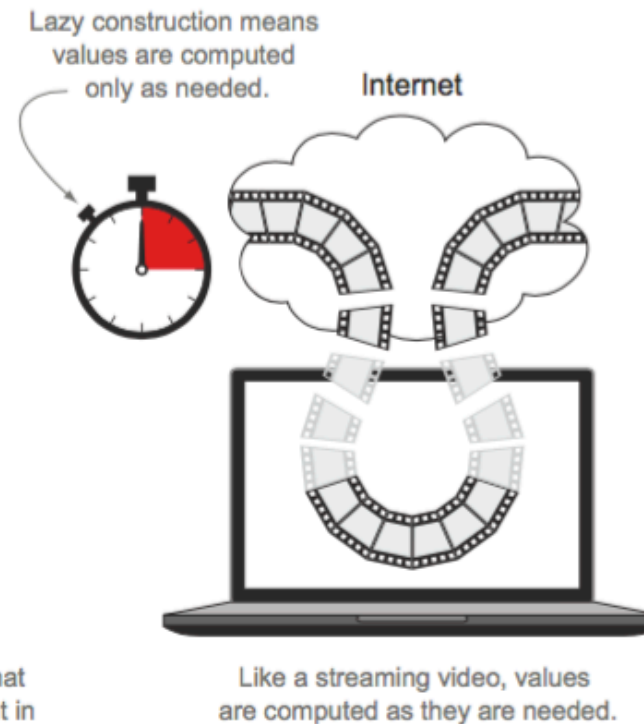
- Une séquence d'éléments
  - Les streams sont des séquences de valeurs (objets ou types primitifs) d'un certain type.
  - Contrairement aux collections, il ne stockent pas leurs éléments
- Source
  - Les streams consomment les éléments issues d'une source (collections, tableaux, I/O) en conservant l'ordre
- Opérations de traitement des données
  - Les streams supportent les opérations de type requête de base de données ET les opérations typiques des langages de programmation fonctionnelle : filter, map, reduce, find, match, sort, etc.
  - Ces opérations peuvent être exécutées séquentiellement ou en parallèle.

# Streams vs Collections

A collection in Java 8 is like  
a movie stored on DVD



A stream in Java 8 is like a movie  
streamed over the internet.



- Avec le streaming il n'est pas nécessaire d'avoir téléchargé la dernière image du film pour pouvoir commencer à le regarder.
- Pas de stockage ou très peu (buffer).

# Stream

- Comme les itérateurs, les streams ne peuvent être *traversés* qu'une seule fois.
- Une fois *traversé*, on dit que le stream a été *consommé*.

```
List<String> mots = Arrays.asList("Després", "est", "génial");
Stream<String> s = mots.stream();
s.forEach(System.out::println);
s.forEach(System.out::println);
```



java.lang.IllegalStateException: stream has already been operated upon or closed.



# Itérateur interne vs externe

## Itérateur externe

Moi : Lucie, est-ce qu'il y a un jouet qui traine par terre ?

Lucie : Oui, le ballon.

Moi : Ok, range le ballon dans la malle. Y-a-t-il autre chose ?

Lucie : Oui, mon livre.

Moi : Ok, range le livre dans la malle. Y-a-t-il autre chose ?

Lucie : Oui, mon doudou.

Moi : Ok, range le doudou dans la malle. Y-a-t-il autre chose ?

Lucie : Non.

Moi : Très bien tu as terminé.

## Itérateur interne

Moi : Lucie, range tous les jouets qui sont par terre dans la malle.

- Lucie peut choisir de ranger les jouets deux par deux en prenant un dans chaque main.
- Elle peut décider de commencer par les jouets les plus proches de la malle.
- Etc.

# Itérateur interne vs externe

## Itérateur interne

```
List<String> noms = menus.stream()
 .filter(m -> m.getCalories() > 300)
 .map(Menu::getNom)
 .limit(3)
 .collect(toList());
```

## Itérateur externe

```
List<String> noms;
List<String> temp = new ArrayList<String>();
for(Menu m:menus){
 if(m.getCalories() > 300)
 temp.add(m.getNom())
}
noms = temp.subList(0,2);
```

# Itérateur interne vs externe

## Itérateur interne

```
List<String> noms = menus.stream()
 .filter(m -> m.getCalories() > 300)
 .map(Menu::getNom)
 .limit(3)
 .collect(toList());
```

## Itérateur externe

```
List<String> noms;
List<String> temp = new ArrayList<String>();
for(Menu m:menus){
 if(m.getCalories() > 300)
 temp.add(m.getNom())
}
noms = temp.subList(0,2);
```

---

# Opérations sur les streams

- L'interface Stream définit un grand nombre d'opérations que l'on peut classer en deux grandes catégories :
  - Les opérations intermédiaires qui, à partir d'un Stream, produisent un Stream. Elles peuvent donc être chaînées :
    - filter, map, limit, etc.
  - Les opérations terminales qui provoquent l'exécution des opérations en chaîne et ferme le Stream.
    - collect


---

# Opérations intermédiaires

- Ces opérations retournent un autre stream ce qui leur permet d'être connectées pour former une requête plus complexe.
- Ces opérations ne réalisent aucun traitement tant qu'une opération terminale n'est pas invoquée sur le stream. Elles sont dites paresseuses (*lazy evaluation*).
- Les opérations intermédiaires peuvent ainsi être fusionnées et réalisées en une seule passe par l'opération terminale.

# Opérations intermédiaires

```
List<String> noms = menu.stream()
 .filter(m -> { System.out.println("Filtrage" + m.getNom());
 return m.getCalories() > 300; })
 .map(m -> { System.out.println("Mapping" + m.getNom());
 return m.getNom(); })
 .limit(3)
 .collect(toList());
System.out.println(noms);
```



```
Filtrage porc
Mapping porc
Filtrage bœuf
Mapping bœuf
Filtrage poulet
Mapping poulet
[porc, bœuf, poulet]
```

---

# Opérations terminales

- Une opération terminale produit un résultat à partir d'un `Stream`.
- Le résultat est toute valeur qui n'est pas un `Stream` comme une `List`, un `Integer` ou même `void`.
- Exemple :  

```
menus.stream().forEach(System.out::println);
```

  - `forEach` est une opération terminale qui retourne `void`.

# Opérations terminales

```
long nbDiffMenus = menu.stream()
 .filter(m -> m.getCalories() > 300)
 .distinct()
 .count();

System.out.println(nbDiffMenus);
```

- `count` est une opération terminale qui retourne un `long` représentant le nombre d'éléments dans le `Stream`
- `filter` et `distinct` sont des opérations intermédiaires (elles produisent un `Stream`)



---

# Programmer avec les Stream

- Pour résumer, programmer avec des Stream, implique 3 éléments :
  - ❑ Une source de données (Collection, Array, I/O) sur laquelle réaliser une requête.
  - ❑ Une chaine d'opérations intermédiaires.
  - ❑ Une opération terminale qui exécute les opérations intermédiaires et produit un résultat.

# Opérations déjà vues dans le cours

## Opérations intermédiaires

| Opération | Type de retour | Argument de l'opération | Descripteur de fonction |
|-----------|----------------|-------------------------|-------------------------|
| filter    | Stream<T>      | Predicate<T>            | T -> boolean            |
| map       | Stream<R>      | Function<T, R>          | T -> R                  |
| limit     | Stream<T>      | int                     |                         |
| sorted    | Stream<T>      | Comparator<T>           | (T, T) -> int           |
| distinct  | Stream<T>      |                         |                         |

## Opérations terminales

| Opération | Objectif                                                          |
|-----------|-------------------------------------------------------------------|
| forEach   | Consomme chaque élément en appliquant la lambda à l'élément.      |
| count     | Retourne le nombre d'éléments dans le Stream.                     |
| collect   | Réduit le Stream en une collection ou une valeur (Integer, etc.). |

---

# Aplatir un Stream

- On souhaite obtenir la liste des plats de tous les menus.

```
menus.stream()
 .map(Menu::getPlats)
 .distinct()
 .collect(toList());
```

- Problème : on n'obtient pas une `List<Plat>` mais une `List<List<Plat>>`

# Aplatir un Stream

- On va devoir utiliser la fonction `flatMap` de la classe `Stream` :

```
<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>>)
```

- `menus.stream().map(Menu::getPlats)` est du type `Stream<List<Plat>>` donc `T` est du type `List<Plat>`
- `R` est du type `Plat`
- Il faut donc fournir à `flatMap` une fonction qui prend une `List<Plat>` et renvoie un `Stream<Plat>`. Cette fonction existe. C'est `stream()`.

```
List<Plat> plats = menus.stream()
 .map(Menu::getPlats)
 .flatMap(List::stream)
 .distinct()
 .collect(toList());
```

---

# Tester

## anyMatch

```
if(menu.getPlats().stream().anyMatch(Plat::estVegetarien))
 System.out.println("Il y a au moins un plat végétarien");
```

## allMatch

```
if(menu.getPlats().stream().allMatch(Plat::estVegetarien))
 System.out.println("Le menu est 100% végétarien");
```

## noneMatch

```
if(menu.getPlats().stream().noneMatch(Plat::estVegetarien))
 System.out.println("Le menu n'a rien de végétarien");
```

# Trouver

## findAny

```
Optional<Plat> plat = menu.getPlats().stream()
 .filter(Plat::estVegetarien)
 .findAny();
```

## findFirst

```
List<Integer> nombres= Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> premierCarreDivisibleParTrois =
 nombres.stream()
 .map(x -> x * x)
 .filter(x -> x % 3 == 0)
 .findFirst();
```

- Vous vous demandez ce qu'est Optional ?
- Demandez-vous plutôt ce qui se passe quand on ne trouve pas.

# Optional

- La classe `Optional<T>` est un conteneur permettant de représenter l'existence ou l'absence d'une valeur.
- Bye bye les `NullPointerException`
- `isPresent()` retourne `true` si l'objet `Optional` contient une valeur.
- `ifPresent(Consumer<T> block)` exécute `block` si une valeur est présente.
- `T get()` retourne la valeur contenue dans l'objet `Optional`. S'il n'y a pas de valeur, une `NoSuchElementException` est lancée.
- `T orElse(T other)` retourne la valeur si elle est présente, `other` sinon.

# Optional

## isPresent

```
List<Integer> nombres= Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> premierCarreDivisibleParTrois =
 nombres.stream()
 .map(x -> x * x)
 .filter(x -> x % 3 == 0)
 .findFirst();
if(premierCarreDivisibleParTrois.isPresent())
 System.out.println(premierCarreDivisibleParTrois.get());
```

## ifPresent

```
List<Integer> nombres= Arrays.asList(1, 2, 2, 4, 5);
nombres.stream()
 .map(x -> x * x)
 .filter(x -> x % 3 == 0)
 .findFirst()
 .ifPresent(System.out::println);
```



# Optional

## orElse

```
List<Integer> nombres= Arrays.asList(1, 2, 3, 4, 5);
Integer premierCarreDivisibleParTrois =
 nombres.stream()
 .map(x -> x * x)
 .filter(x -> x % 3 == 0)
 .findFirst()
 .orElse(0);
System.out.println(premierCarreDivisibleParTrois);
```

- `premierCarreDivisibleParTrois` n'est plus une instance d'`Optional` mais d'`Integer`.

# Réduire

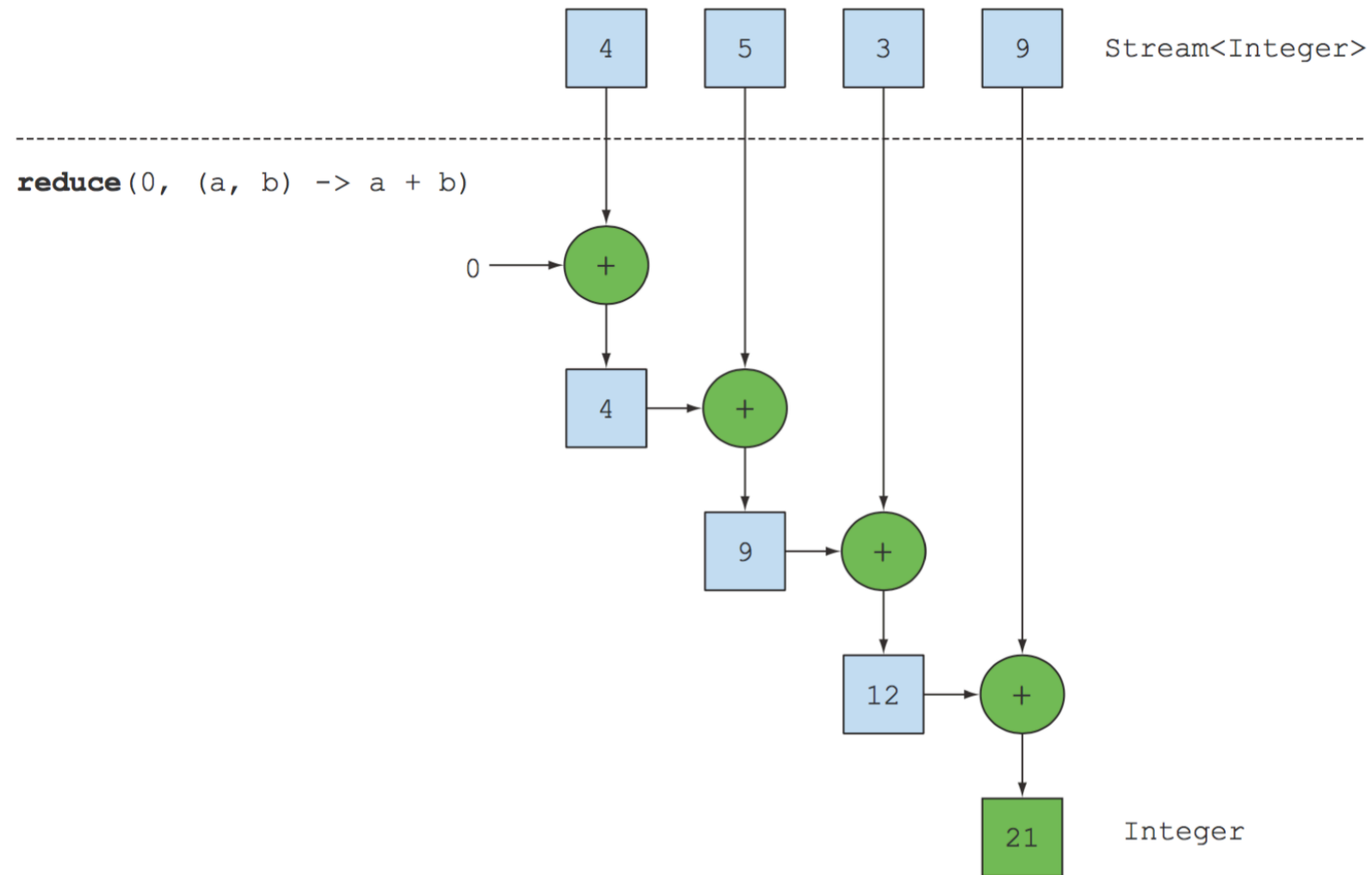
- Les opérations de réduction consistent à combiner les éléments d'un `Stream` pour produire une seule valeur (par ex. un `Integer`)

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

```
List<Integer> nombres= Arrays.asList(4, 5, 3, 9);

int somme = nombres.stream().reduce(0, (a,b) -> a + b);
int produit= nombres.stream().reduce(1, (a,b) -> a * b);
```

# Réduire



# Réduire

```
List<Integer> nombres= Arrays.asList(4, 5, 3, 9);

int sommel = nombres.stream().reduce(0, Integer::sum);
Optional<Integer> somme2 = nombres.stream().reduce((a,b)->a+b);

System.out.println(sommel);
System.out.println(somme2.orElse(0));
```

```
List<Integer> nombres= Arrays.asList(4, 5, 3, 9);

int max1 = nombres.stream().reduce(Integer.MIN_VALUE, (a,b)->a>b?a:b);
Optional<Integer> max2 = nombres.stream().reduce((a,b)->a>b?a:b);
Optional<Integer> max3 = nombres.stream().reduce(Integer::max);

System.out.println(max1);
System.out.println(max2.orElse(Integer.MIN_VALUE));
System.out.println(max3.orElse(Integer.MIN_VALUE));
```

# Stream numériques

- Il existe des `Stream` pour les types numériques primitifs `int`, `double` et `long` :
  - `IntStream`
  - `DoubleStream`
  - `LongStream`
- Ces `Stream` permettent :
  - de limiter le coût des opérations d'auto-boxing (unboxing)
  - de profiter d'opérations terminales dédiées : `sum`, `max`, `min`, `average`, etc.

# Stream numériques

## Mapping et somme

```
int calories = menu.getPlats().stream()
 .mapToInt(Plat::getCalories)
 .sum();
```

## Conversion

```
IntStream intStream = menu.getPlats().stream()
 .mapToInt(Plat::getCalories);

Stream<Integer> stream = intStream.boxed();
```

# Optional pour les numériques

- De la même façon que pour les `Stream`, il existe des `Optional` pour les types numériques primitifs : `OptionalInt`, `OptionalDouble`, `OptionalLong`

```
OptionalInt maxCalories = menu.getPlats().stream()
 .mapToInt(Plat::getCalories)
 .max();

int max = maxCalories.orElse(0);
```

# Construire des Stream

- Les `IntStream` et `LongStream` offrent la possibilité de générer les nombres dans un intervalle grâce à deux méthodes de classe : `range` et `rangeClosed`.

```
IntStream nombresPairs = IntStream.rangeClosed(1, 100) //1..100
 .filter(n -> n % 2 == 0);
System.out.println(nombresPairs.count()); // 50

nombresPairs = IntStream.range(1, 100) //1..99
 .filter(n -> n % 2 == 0);
System.out.println(nombresPairs.count()); // 49
```



# Construire des Stream

## A partir de valeurs

```
Stream<String> stream = Stream.of("Després", "est", "génial");

stream.map(String::toUpperCase).forEach(System.out::println);
```

## A partir d'un tableau

```
int[] numbers = {2, 3, 5, 7, 11, 13};

int sum = Arrays.stream(numbers).sum();
```

## A partir d'un fichier

```
long uniqueWords = 0;
try(Stream<String> lines = Files.lines(Paths.get("data.txt"))){
 uniqueWords = lines.flatMap(line ->
 Arrays.stream(line.split(" "))
 .distinct()
 .count());
} catch(IOException e){ }
```

# Construire des Stream

## ■ Construire des Stream infinis définis par une fonction :

### □ iterate

- `static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)`
- Stream produit par application itérative d'une fonction `f` sur l'élément initial `seed`.
- Le Stream produit `seed`, `f(seed)`, `f(f(seed))`, etc.


### □ generate

- `static <T> Stream<T> generate(Supplier<T> s)`
- Chaque élément du Stream est produit par le Supplier
- Cela est généralement utilisé pour produire des Stream constant ou des Stream d'éléments aléatoires.

# Construire des Stream

## Iterate


```
Stream.iterate(0, n -> n + 2)
 .limit(10)
 .forEach(x -> System.out.print(x + " "));
```



0 2 4 6 8 10 12 14 16 18

## Generate

```
Stream.generate(Math::random)
 .limit(5)
 .forEach(System.out::println);
```



0.05670151338118268  
0.9022216378234214  
0.8355136188928481  
0.9058196933812706  
0.0579429672078996

# Construire des Stream

## Fibo avec generate

```
Supplier<BigInteger> fib = new Supplier<BigInteger>() {
 private BigInteger UN= new BigInteger("0");
 private BigInteger UN1= new BigInteger("1");
 public BigInteger get() {
 BigInteger ancienUN= this.UN;
 BigInteger prochainUN1= this.UN.add(this.UN1);
 this.UN = this.UN1;
 this.UN1 = prochainUN1;
 return ancienUN;
 }
};
```

```
Stream.generate(fib).limit(100).forEach(System.out::println);
```



```
0
1
1
2
3
5
8
```

# Construire des Stream

## Fibo avec iterate

Stream

```
.iterate(new BigInteger[]{new BigInteger("0"), new BigInteger("1")}
 , t -> new BigInteger[]{t[1], t[0].add(t[1])})
.limit(200)
.forEach(t -> System.out.println("(" + t[0] + "," + t[1] + ")"));
```



(0,1)(1,1)(1,2)(2,3)(3,5)(5,8)(8,13)(13,21)(21,34)(34,55)(55,89)(89,144)(144,233)(233,377)