



Notre expertise est votre avenir



Nouveautés Java 8

Les expressions lambda	p.9
Les expressions lambda : pourquoi un tel engouement?	p.11
Cheminement vers les expressions lambda : étape 1	p.13
Cas commun d'utilisation d'une expression lambda	p.22
Synthèse des expressions lambda	p.28
Règles des expressions lambda	p.30
 Les interfaces fonctionnelles	 p.31
Qu'est-ce qu'une interface fonctionnelle?	p.32
Relation entre expression lambda et interface fonctionnelle	p.34
Interfaces fonctionnelles existantes	p.39
Autre interface fonctionnelle	p.46
 Les références de méthodes	 p.53
Références de méthodes : modalités	p.55
Les types de références de méthodes	p.56
Exemple de références de méthodes (lambda)	p.57
 Les méthodes par défaut	 p.63
Exemple d'utilisation	p.66
Nouvelles méthodes dans les Collections	p.68
Exemple d'utilisation des nouvelles méthodes	p.69
 Les Streams	 p.73
Point essentiel des Streams	p.78
Créer des Streams	p.79
Les opérations intermédiaires et terminales	p.81
Opérations terminales	p.92
MapReduce	p.104
Autre exemple sur les Streams	p.109
Opérations sur les Streams	p.113
 Les dates	 p.121
L'API DateTime	p.117
Classes importantes de DateTime	p.118
Utilisation	p.120


La programmation fonctionnelle

- Le programmation fonctionnelle est un style de programmation qui met l'accent sur l'évaluation d'expressions plutôt que sur l'exécution de commandes.

Les programmes impératifs (pas fonctionnels)

- Sont des séquences d'instruction.
- Exemple :

Temps



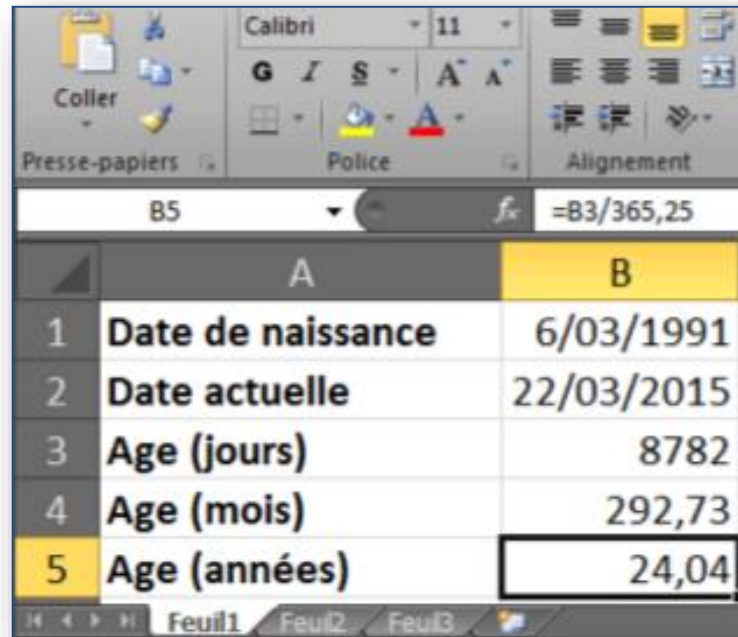
```
void estPair(int x)
{
    if (x % 2 == 0)
        printf("x est pair !\n");
    else
        printf("x est impair !\n");
}
```

Fais ça
puis ça
puis ça
...

- La programmation fonctionnelle retire cette notion de temps.

Excel : un excellent exemple de langage fonctionnel

- Exemple :



The screenshot shows the Excel ribbon with the 'Formules' (Formulas) tab active. The formula bar displays the formula `=B3/365,25` in cell B5. The worksheet contains a table with the following data:

	A	B
1	Date de naissance	6/03/1991
2	Date actuelle	22/03/2015
3	Age (jours)	8782
4	Age (mois)	292,73
5	Age (années)	24,04

The formula in cell B5, `=B3/365,25`, calculates the age in years by dividing the number of days (B3) by the average number of days in a year (365,25). The result, 24,04, is displayed in cell B5.

La programmation fonctionnelle

- Une fonction ou une expression accepte des données en entrée et en retourne de nouvelles.
- MAIS NE MODIFIE AUCUN ETAT GLOBAL, NI AUCUN DE SES ARGUMENTS
 - Il y a donc copie des données

La programmation fonctionnelle : comment ?

- Pas de modification de données
 - Pour modifier une donnée, on en crée une nouvelle copie
- Pas d'effet de bord
 - Pas de modification de variable globale
 - Pas d'entrée / sortie

Java 8 et fonctions

- Dans Java 8, les fonctions deviennent des entités de « première classe », comme en JavaScript.
- Désormais dans Java 8, les fonctions peuvent se désolidariser des objets pour acquérir le statut d'**entités opératoires autonomes** et manipulables en tant que telles.
- Nous allons pouvoir :
 - Passer des fonctions en tant que paramètre de méthode !

LES EXPRESSIONS LAMBDA

Les expressions lambda (EL)

- C'est une fonction anonyme, donc une fonction qui n'a pas de nom, ni d'identificateur.
 - Plus de classe ni d'instanciation
- Une expression lambda est écrite à l'endroit où elle sera utilisée, donc une fois.
 - Ainsi une même expression lambda s'exécutera différemment selon le contexte, selon l'endroit où elle est utilisée.
- Une expression lambda s'exécute dans le contexte où elle apparaît.

Les expressions lambda : pourquoi un tel engouement ?

- La plupart des langages actuels sont objets. La classe est un élément de fondation.
- Exemple en Java, aucune fonction ne peut être écrite en dehors d'une classe; une fonction seule n'a pas de sens en Java.
 - Ainsi, si l'on veut faire quelque chose de très simple, on est obligé de passer par une classe, une méthode, etc.
 - C'est ce que l'on veut éviter de faire avec les lambda
- En programmation fonctionnelle, vous pouvez définir des variables qui référencent des fonctions, et par exemple les passer en tant qu'argument (paramètre) de méthode.
 - JavaScript est un bon exemple
 - Java va permettre à peu près la même chose

Exemple de syntaxe d'expression lambda

- Exemple simple :
 - $(x,y) \rightarrow x+y$
 - C'est une fonction qui prend 2 paramètres et retourne leur somme.
- Comment en est-on arrivé à cette syntaxe depuis Java et ses classes ?
 - Le cheminement, page suivante.

Cheminement vers les expressions lambda : étape 1

- Partons de cet exemple bien connu

```
interface Vehicule{
    void rouler();
}
class Voiture implements Vehicule{
    public void rouler()
    {
        System.out.println("la voiture roule");
    }
}
public class L1 {
    public static void main(String[] args) {
        Vehicule v1;
        v1 = new Voiture();
        v1.rouler();
    }
}
```

- Il est nécessaire de coder Voiture pour utiliser Vehicule:

Cheminement vers les expressions lambda : étape 2

- Si l'on tente d'instancier l'interface pour l'utiliser, problème de compilation :

```
interface Vehicule{
void rouler();
}
class Voiture implements Vehicule{
public void rouler(){
System.out.println("la voiture roule");
}
}
public class L1 {
public static void main(String[] args) {
Vehicule v1;
v1 = new Vehicule();
v1.rouler();
}
}
```

Cannot instantiate
Vehicule !!!!

Cheminement vers les expressions lambda : étape 3

- Nous pouvons résoudre comme ceci :

```
interface Vehicule{
void rouler();
}
public class L1 {
public static void main(String[] args) {
Vehicule v1;
    v1 = new Vehicule(){
        @Override
        public void rouler() {
            System.out.println("la voiture roule");
        }
    };
    v1.rouler();
}
```

Toute la notion
des lambda est
ici, ainsi que
celles des
interfaces
fonctionnelles



Ce qui est important

- L'interface Vehicule ne possède qu'une méthode abstraite (rouler).
 - Obligation pour les « interfaces fonctionnelles ».
- Il est donc facile de déduire que seule la méthode rouler est utilisable dès que l'on traite de Vehicule.
- L'unique méthode est bien signée. Ici, elle n'attend rien et renvoie void.
 - Les types sont connus, il n'est donc pas nécessaire de les rappeler.
- Voir page suivante pour le passage à l'écriture lambda

Cheminement vers les expressions lambda : étape 4

- Finalement, le code précédent peut être remplacé par l'opérateur lambda pour faire la même chose :

```
interface Vehicule{
    void rouler();
}
public class L1 {
    public static void main(String[] args) {
        Vehicule v1;
```

```
        v1 = ()->{System.out.println("la voiture roule");};
```

```
    }
}
```

Valeur des paramètres qu'attend rouler(), ici, aucun

Méthode non nommée car, déduisant le type de l'interface dans laquelle la lambda expression est affectée, on connaît la méthode (rouler de Vehicule)

Retour de la lambda Expression, ici void, car println() retourne void

Affectation et exécution

- Dans le code précédent, il y a bien eu :
 - Affectation, qui ne produit rien
 - `v1 = ()->{System.out.println("la voiture roule");};`
 - Exécution
 - `v1.rouler();`

Cheminement vers les expressions lambda : étape 5

- Si par exemple, on change la signature de rouler() en ajoutant des paramètres, alors on écrit :

```
interface Vehicule{
void rouler(String type);
}
public class L1 {
    public static void main(String[] args) {
        Vehicule v1;
v1 = (String letype)->{System.out.println(letype);};
        v1.rouler("la voiture roule");
    }
}
```

Cheminement vers les expressions lambda : étape 5

- On peut même encore simplifier, en déduisant le type :

```
interface Vehicule{
void rouler(String type);
}
public class L1 {
public static void main(String[] args) {
Vehicule v1;
v1 = (letype)->{System.out.println(letype);};
v1.rouler("la voiture roule");
}
}
```

Cheminement vers les expressions lambda : étape 6

- Encore de la simplification :

```
interface Vehicule{
void rouler(String type);
}
public class L1 {
    public static void main(String[] args) {
        Vehicule v1;
        v1 = letype->{System.out.println("c'est certain " + letype + " sur la route");};
        v1.rouler("la voiture roule");
    }
}
```

Cas commun d'utilisation d'une expression lambda

- La gestion des événements, comme dans swing, se prête parfaitement aux lambda
- En effet, la gestion d'un événement est localisée, en général non réutilisée.
 - Examinons cet exemple:

```

12 public class Affiche extends JFrame
13 {
14     private JButton btnClickMe = new JButton("Ici !");
15     public void start() {
16         this.setVisible(true);
17         this.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
18         this.setSize(600,400);
19         this.setLocationRelativeTo(null);
20         JPanel content =(JPanel) this.getContentPane();
21         content.setLayout(new FlowLayout());
22         content.add(btnClickMe);
23         btnClickMe.addActionListener(new ActionListener() {
24             @Override
25             public void actionPerformed(ActionEvent e) {
26                 System.out.println("Bien");
27             }
28         });
29     }
30 }

```

Seule
méthode

Cas commun d'utilisation d'une expression lambda

- Dans l'exemple précédent, les lignes 23 à 28 se prêtent bien à l'utilisation d'une lambda.
- En effet, `addActionListener` attend en paramètre une interface nommée `ActionListener`, qui, elle-même, ne possède qu'une méthode...
- On peut donc remplacer les lignes 23 à 28 par :

35

```
btnClickMe.addActionListener((e) ->System.out.println("Bien")) ;
```



La lambda est compatible avec
l'interface fonctionnelle attendue
par `addActionListener`

Cas commun d'utilisation d'une expression lambda

- Cette ligne de la page précédente :

```
35 btnClickMe.addActionListener((e) ->System.out.println("Bien")) ;
```

- Peut aussi, et sans problème être remplacée par :

```
38 btnClickMe.addActionListener(e ->System.out.println("bien")) ;
```

- Ou aussi par :

```
40 btnClickMe.addActionListener((ActionEvent e) ->System.out.println("bien")) ;
```

- On peut bien sûr exploiter les paramètres :

```
44 btnClickMe.addActionListener(e ->System.out.println("ok lambda 4"+e.getSource())) ;
```


Cas commun d'utilisation d'une expression lambda

- Dès lors qu'il y a plus d'une ligne de code dans l'expression lambda, il est nécessaire d'utiliser le « ; », mais aussi d'utiliser les accolades. Le « return » devra être en accord avec le type de retour de la méthode virtuelle de l'interface fonctionnelle :

```
54 btnClickMe.addActionListener(  
55     e ->{  
56         System.out.println("Bien" +e.getSource());  
57         return;  
58     }  
59 );
```

Autres cas commun d'utilisation des lambda

- Supposons que l'on veuille afficher une collection , sans utiliser de lambda

```
Collection<String> list = Arrays.asList("Pierre","Roger")  
    for (int i = 0; i < list.size(); i++) {  
        System.out.println(list.get(i));  
    }
```

- Grâce au fait que l'interface Collection se soit vue complétée en Java 8 de la méthode default `forEach`, on pourra faire la même chose avec les lambda de deux façons :

```
Collection<String> list = Arrays.asList("Pierre","Roger");  
list.forEach(System.out::println); → on passe ici une référence de méthode en paramètre  
list.forEach((e)->{System.out.println(e);}); → Lambda expression
```


Cas commun d'utilisation d'une expression lambda

- Enfin, il est possible de référencer une méthode tout à fait normale depuis une expression lambda. Il n'est nullement nécessaire de passer les paramètres, le système s'en chargera :

```

66         btnClickMe.addActionListener(this::clickListener);
67
68     }
69     private void clickListener(ActionEvent ae) {
70         System.out.println("bien");
71     }

```



Synthèse des expressions lambda

- Une expression lambda ne s'utilise jamais seule.
 - Exemple :
 - `() -> « salut »`; ne sert à rien.
- Une expression lambda doit être « affectée » à une interface, que l'on appelle interface fonctionnelle.
 - Exemple :
 - `String exemple = () -> « salut »`; ne fonctionne pas car `String` n'est pas une interface fonctionnelle.
- La signature de l'expression lambda (ci-dessus `String`) doit correspondre à la signature **de l'interface fonctionnelle** (unique méthode virtuelle)
 - Exemple :
 - `Runnable exemple = () -> « salut »`; ne fonctionne pas car « salut » est `String` et la méthode par défaut de `Runnable` (`run`) est `void`.
 - Ainsi, il faudrait écrire:
 - Affectation :
 - `Runnable exemple = () -> {System.out.println(« salut »);}`;
 - Exécution :
 - `exemple.run();`
- Ou bien, en utilisant l'interface fonctionnelle `Supplier` (ne prend pas d'argument mais retourne une `String`)
 - `Supplier<String> exemple = () -> « salut »`;
 - `String salut = exemple.get();`

Bases des expressions lambda

- Possibilités d'écriture d'expressions lambda :
 - (parametres) -> expression
 - (parametres)->{instruction;}
 - ()->expression
- Exemples :
 - (int a, int b) -> a*b; → 2 entiers et retour du produit
 - (a,b) -> a-b; → 2 nombres et différence
 - (String a) -> System.out.println(a) → prend un String, affiche sa valeur et retourne void.
 - c -> { /* */ } → Execute la série d'instructions, avec ; et {}

Règles des expressions lambda

- Une expressions lambda peut avoir de 0 à n paramètres.
- Le type de ces paramètres peut être déclaré de façon explicite **ou déduit du contexte**.
- S'il y a plusieurs paramètres, alors ils sont séparés par des virgules à l'intérieur de parenthèses.
- Des parenthèses vides représentent une absence de paramètres.
- Lorsqu'il n'y a qu'un seul paramètre, si on a choisi un type déduit, alors les parenthèses sont facultatives.
 - `A->return A*A`
- Le corps d'une expression lambda peut contenir de une à n instructions.

LES INTERFACES FONCTIONNELLES

Qu'est ce qu'une interface fonctionnelle ?

- C'est tout simplement une interface contenant une seule méthode abstraite.
 - Ce qu'on appelle une SAM (Single Abstract Method Interface).
 - Ce n'est pas nouveau, exemple : Runnable, issu de JDK 1!
 - Java 8 ajoute l'annotation `@FunctionalInterface` pour qu'une SAM devienne interface fonctionnelle, mais ce n'est pas obligatoire.
 - Exemple en Java 8 :

```
@FunctionalInterface
    public interface Runnable {
        public abstract void run();
    }
```

- Si vous tentez d'ajouter une méthode à une interface `@FunctionalInterface`, il y a erreur de compilation.
 - Avantage d'ajouter `@FunctionalInterface`

Qu'est ce qu'une interface fonctionnelle ?

- Une interface fonctionnelle, outre le fait qu'elle ne doit contenir qu'une seule méthode abstraite, peut contenir plusieurs méthodes implémentées, à condition qu'elles soient signées « default ».
 - Mécanisme utilisé par exemple dans l'interface Collection de Java 8, où foreach a été ajouté en default, ce qui a permis d'ajouter des fonctionnalités à cette interface sans casser la compatibilité ascendante du code.

Relation entre expression lambda et interface fonctionnelle (1)

- Les expressions lambda sont des fonctions anonymes *et sont passées la plupart du temps à d'autres fonctions en tant que paramètre.*
- En Java, les paramètres de méthode ont toujours un type, et ce type est exploité afin de déterminer quelle méthode appeler dans le cas de surcharge par exemple.
- De même, chaque expression lambda doit être convertie vers un type acceptable en tant que paramètre de méthode.
 - Le type dans lequel l'expression lambda est convertie EST TOUJOURS une interface fonctionnelle.

Relation entre expression lambda et interface fonctionnelle (2)

- Exemple : Nous devons écrire un thread qui affiche « ok » la console. Le code le plus simple est :

```
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println(" ok");
    }
}).start();
```

- Avec une expression lambda :

```
new Thread(
    () -> {
        System.out.println("ok");
    }
).start();
```

Relation entre expression lambda et interface fonctionnelle (3)

- Éléments sur la page précédente :
 - Runnable est une interface fonctionnelle avec une seule méthode, run().
 - Ainsi, lorsque l'on passe une expression lambda au constructeur de la classe Thread, le compilateur essaie de convertir cette expression en équivalent Runnable, tel que présenté au premier exemple page précédente.
 - Si le compilateur réussit, tout fonctionne, sinon, une erreur survient.
 - Ainsi, ici, l'expression lambda est convertie en Runnable, qui est une interface fonctionnelle.

En résumé

- *Lambda expression* correspond à « instance » en objet
- *Interface fonctionnelle* correspond à « type » en objet
- Une expression lambda est une sorte d'instance d'interface fonctionnelle. Mais l'expression lambda ne contient aucune information concernant l'interface fonctionnelle qu'elle implémente, car ces informations sont déduites du contexte dans lequel l'expression lambda est utilisée.

Les interfaces fonctionnelles

- Chacun peut créer son interface fonctionnelle. Mais pour faciliter, JDK8 propose dans le package `java.util.function`, une série d'interfaces fonctionnelles prêtes à être utilisées.

Interfaces fonctionnelles existantes

- Les interfaces définies avec des types génériques sont :
 - `Consumer<T>` : opération qui accepte un unique argument (type T) et ne retourne pas de résultat.
 - `void accept(T);`
 - `Function<T,R>` : opération qui accepte un argument (type T) et retourne un résultat (type R).
 - `R apply(T);`
 - `Supplier<T>` : opération qui ne prend pas d'argument et qui retourne un résultat (type T).
 - `T get();`
 - L'interface `Predicate` qui est une spécialisation de `Function` visant à tester une valeur et retourner un booléen.
 - `boolean test(T);`

Exemple d'utilisation de Consumer et Supplier

- Ici, on comprend que la notion de « pointeur sur fonction » commence à exister en Java :

```

65 faire(
66     () -> 42,          ← On fournit une valeur
67     System.out::println, ← On consomme une valeur
68     ex -> System.err.println("Error: " + ex.getMessage())
69 );

```

```

71 }
72 public <T> void faire(Supplier<T> function, Consumer<T> onSuccess, Consumer<Exception> onError) {
73     try {
74         T res = function.get();
75         onSuccess.accept(res);
76     } catch (Exception ex) {
77         onError.accept(ex);
78     }
79     return;
80 }

```


Exemple d'utilisation de Predicate

- Exemple simple

```
72 Predicate<String> i = (s) -> s.length() > 5;  
73 |  
74 System.out.println(i.test("formation java"));
```

Quelques interfaces fonctionnelles existantes

- `Function<T,R>` : un param d'entrée (T), un return (R)

```
public class TestInterfacesFonctionnelles {
    public static void main (String[] arg){
        Function<String, String> function = x -> x.toUpperCase();
        Function<String, String> function2 = x -> x.toLowerCase();
        convertString(function); // affiche STRANGE
        convertString(function2); // affiche strange
    }
    public static void convertString(Function<String, String> function){
        System.out.println(function.apply("StRaNgE")); → Ici on passe le string, on récupère le résultat
    }
}
```

- Dès que `apply()` est exécuté, l'expression est évaluée. Notez que le type du paramètre d'entrée ainsi que le return n'est pas nécessaire du fait qu'ils sont inférés à partir de la signature de la méthode `apply()`.

Encore un exemple de Supplier

- Supplier <T> : un return , pas de paramètre, c'est un fournisseur de résultat

```
Supplier<String> supplier1 = () -> "String1";
```

```
Supplier<String> supplier2 = () -> "String2";
```

```
printSuppliedString(supplier1);
```

```
printSuppliedString(supplier2);
```

```
public static void printSuppliedString(Supplier<String> supplier){
```

```
    System.out.println(supplier.get());
```

```
}
```

- L'appel de get() permet l'évaluation de l'expression lambda.

Encore un exemple de Consumer

- Consumer <T> : un paramètre , sans résultat

```
Consumer<String> function3 = x -> System.out.println(x);  
Consumer<String> function4 = x -> System.out.println(x.toLowerCase());  
consumeString(function3, "StringA");  
consumeString(function4, "StringA");  
public static void consumeString(Consumer<String> consumer, String x) {  
    consumer.accept(x);  
}
```

- L'appel de accept() permet l'évaluation de l'expression lambda.

Encore un exemple de Predicate

- Predicate <T> : un paramètre, résultat de type booléen

```
Predicate<Double> function5 = x -> x > 10;
```

```
Predicate<Double> function6 = x -> x < -10;
```

```
System.out.println(function5.test(new Double(9)));// affiche false
```

```
System.out.println(function6.test(new Double(-20)));// affiche true
```

```
public static void testValue(Predicate<Double> predicate, Double d){
    predicate.test(d);
}
```

- L'appel de test() permet l'évaluation de l'expression lambda.

Autre interface fonctionnelle

- BiFunction <T,U,R> : deux paramètres, un résultat

```
BiFunction <String,String,String> replace3 =  
(param,param1) -> {String a=param.replace(param1,"b");  
return a;};  
System.out.println(replace3.apply("abcde","c"));
```

- L'appel de apply() permet l'évaluation de l'expression lambda.

Utilisation d'une expression lambda

- Une expression lambda ne peut être utilisée que dans un contexte où le compilateur peut identifier l'utilisation de son type cible (target type) qui doit être une interface fonctionnelle :
 - Déclaration d'une variable
 - Assignation d'une variable
 - Valeur de retour avec l'instruction return
 - Initialisation d'un tableau
 - Paramètre d'une méthode ou d'un constructeur
 - Corps d'une expression lambda
 - Opérateur ternaire ?:
 - Cast

Utilisation d'expression lambda

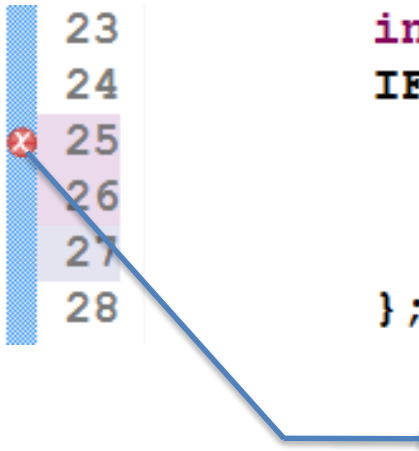
- Le compilateur associe une expression lambda à une interface fonctionnelle et comme une interface fonctionnelle ne peut avoir qu'une seule méthode abstraite :
 - Les types des paramètres doivent correspondre à ceux des paramètres de la méthode
 - Le type de retour du corps de l'expression doit correspondre à celui de la méthode
 - Toutes les exceptions levées dans le corps de l'expression doivent être compatibles avec les exceptions déclarées dans la clause throws de la méthode

Exemple de l'explication précédente

```
package lambdadefense;
public class Calculatrice {
    @FunctionalInterface
    interface OperationEntiere {
        long effectuer(int a, int b);
    }
    public long calculer(int a, int b, OperationEntiere operation) {
        return operation.effectuer(a, b);
    }
    public static void main(String[] args) {
        Calculatrice calc = new Calculatrice();
        OperationEntiere addition = (a, b) -> a + b;
        OperationEntiere soustraction = (a, b) -> a - b;
        System.out.println(calc.calculer(10, 5, addition));
        System.out.println(calc.calculer(10, 5, soustraction));
    }
}
```

Expression lambda et visibilité des variables (1)

- Le corps de la lambda ne définit pas un nouveau scope, il est le même que le scope englobant.
- Par exemple, ici « i » est défini à l'extérieur et à l'intérieur de l'expression lambda, et il y a erreur :



```

23      int i;
24      IFonctionnel ifonct1 = (num1,num2) ->{
25          int i;
26          i=num1*num2;
27          return i;
28      };
    
```

Error : Lambda expression's local variable i cannot redeclare another local variable defined in an enclosing

Expression lambda et visibilité des variables (2)

- Quand une lambda utilise une variable déclarée en dehors d'elle et assignée depuis son espace (entre {}), une restriction importante s'applique :
 - Elle DOIT ETRE déclarée final, ou faire en sorte de l'être, ainsi :
 - Les expressions lambda capturent les valeurs des variables, et non les variables. Les variables locales que l'expression lambda utilise sont nommées « effectively final ».
 - Une variable « effectively final » NE CHANGE PAS.
 - Il n'est par contre pas nécessaire (mais possible) de les déclarer final, à partir du moment où elles ne sont pas réutilisées avant l'expression lambda.
- Exemple :

```

23      int i=8;|
24      IFonctionnel ifonct1 = (num1,num2)->{
25          i=num1*num2;
26          return i;
27      };
28
29      int a = ifonct1.func(10,20);
    
```

Error : Local variable i defined in an enclosing scope must be final or effectively final

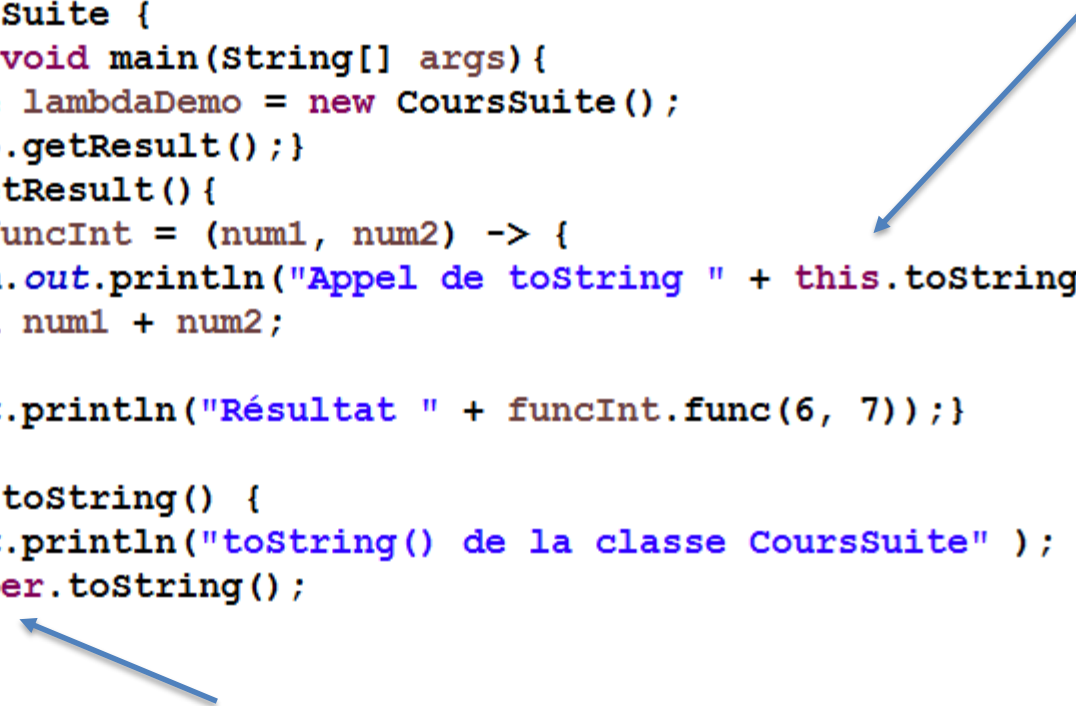
Expression lambda et visibilité des variables (3)

- Les utilisations de `this` et de `super` dans une expression sont égales à celles utilisées dans leur contexte englobant.

```

2  @FunctionalInterface
3  interface IFuncInt {
4      int func(int num1, int num2);
5      public String toString();
6  }
7  public class CoursSuite {
8      public static void main(String[] args){
9          CoursSuite lambdaDemo = new CoursSuite();
10         lambdaDemo.getResult();
11     public void getResult(){
12         IFuncInt funcInt = (num1, num2) -> {
13             System.out.println("Appel de toString " + this.toString());
14             return num1 + num2;
15         };
16         System.out.println("Résultat " + funcInt.func(6, 7));
17     @Override
18     public String toString() {
19         System.out.println("toString() de la classe CoursSuite" );
20         return super.toString();
21     }
22 }

```



LES RÉFÉRENCES DE MÉTHODES

Les références de méthodes

- Une fonction pure peut se déclarer de manière autonome en utilisant une lambda.
 - Il est aussi possible de transformer une méthode en fonction en utilisant un mécanisme de référence.
 - Il existe plusieurs modalités de référencement des méthodes.

Références de méthodes : modalités

- Statique :
 - Permet de référencer une méthode statique d'une classe.
- Instance :
 - Permet de référencer une méthode d'une instance.
- Super :
 - Permet de référencer une méthode d'une instance mais qui est définie dans une classe parente.
- Constructeur :
 - Permet de référencer le constructeur d'une classe.

Les types de références de méthodes

- Il existe différents types de références de méthodes :
 - Référence à une méthode statique
 - `nomClasse::nomMethodeStatique`
 - `String::valueOf`
 - Référence à une méthode sur une instance
 - `objet::nomMethode`
 - `personne::toString`
 - Référence à un constructeur
 - `nomClasse::new`
 - `Personne::new`

Exemple de références de méthodes (lambda)

- Partant de cette classe qui servira d'exemple :

```
public class Example {
    public int add(int a, int b) {
        return a + b;}
    public static int mul(int a, int b) {
        return a * b;}
    public String lower(String a) {
        return a.toLowerCase();}
    public void printDate(Date date) {
        System.out.println(date);}
    public void oper(IntBinaryOperator operator, int a, int b) {
        System.out.println(operator.applyAsInt(a, b));}
    public void operS(Function<String, String> stringOperator, String a) {
        System.out.println(stringOperator.apply(a));}
    public GregorianCalendar operC(Supplier<GregorianCalendar> supplier) {
        return supplier.get();}
```

Exemple de références de méthodes (lambda)

- Exemple de référencement de méthode statique en utilisant le nom de la classe :
 - Les deux écritures sont égales en résultat

Exemple `ex = new Example();`

//Forme lambda

`ex.oper((a, b) -> Example.mul(a, b), 4, 2);`

//référence de méthode

`ex.oper(Example::mul, 4, 2);`

Exemple de références de méthodes (lambda)

- Exemple de référencement de méthode en utilisant une instance d'objet :
 - Les deux écritures sont égales en résultat

```
Example ex = new Example();
```

```
//Forme lambda
```

```
ex.oper((a, b) -> ex.add(a, b), 1, 2);
```

```
//Référence de méthode
```

```
ex.oper(ex::add, 1, 2);
```

Exemple de références de méthodes (lambda)

- Exemple de référencement de méthode en utilisant un constructeur :
 - Les deux écritures sont égales en résultat

```
Exemple ex = new Example();
```

```
//Forme lambda
```

```
ex.operC()->{ return new GregorianCalendar();});
```

```
// référence de méthode
```

```
ex.operC(GregorianCalendar::new);
```

Expressions lambda : composition

- La composition permet d'appliquer des expressions lambda les unes après les autres.
- Deux méthodes sont possibles :
 - Function compose (Function before)
 - La fonction before est appliquée en premier, puis la fonction appelante.
 - Function andThen (Function after)
 - La fonction after est appliquée après la fonction appelante

Exemple de composition

```
public class ExampleCompose {
    public static void main(String[] args) {
        ExampleCompose ex = new ExampleCompose();
        Function<Double , Double> sin = d -> ex.sinus(d);
        Function<Double , Double> log = d -> ex.logarithme(d);
        Function<Double , Double> exp = d -> ex.exposant(d);
        ExampleCompose compose = new ExampleCompose();
        System.out.println(compose.calculate(sin.compose(log), 0.8)); // dans l'ordre : log puis sin
        System.out.println(compose.calculate(sin.andThen(log), 0.8)); // dans l'ordre : sin puis log
        System.out.println(compose.calculate(sin.compose(log).andThen(exp), 0.8)); // dans l'ordre : log puis sin puis exp
        System.out.println(compose.calculate(sin.compose(log).compose(exp), 0.8)); // dans l'ordre : exp puis log puis sin
        System.out.println(compose.calculate(sin.andThen(log).compose(exp), 0.8)); // dans l'ordre : exp puis sin puis log
        System.out.println(compose.calculate(sin.andThen(log).andThen(exp), 0.8)); // dans l'ordre : sin puis log puis exp
    }
    public Double calculate(Function<Double , Double> operator, Double d) {
        return operator.apply(d);}
    public Double sinus(Double d) {
        System.out.print("sin:");
        return Math.sin(d);}
    public Double logarithme(Double d) {
        System.out.print("log:");
        return Math.log(d);}
    public Double exposant(Double d) {
        System.out.print("exp:");
        return Math.exp(d);}}
```

LES METHODES PAR DEFAUT

Méthode par défaut

- On connaît la notion d'interface en Java.
 - Obligation d'implémenter l'ensemble des méthodes, ce qui peut être lourd.
 - Ainsi, après publication de cette interface, on ne peut plus ajouter de nouvelles méthodes abstraites, à moins de casser la compatibilité source et binaire.
- Java 8 apporte la possibilité de faire évoluer l'interface en supportant les méthodes par défaut.
- Une méthode par défaut est une méthode définie dans l'interface dont la signature commence avec le mot clé "default".
 - Les classes filles sont donc libérées de l'implémentation de cette méthode.
- Il fournit aussi un corps avec du code.
- Chaque classe qui implémente l'interface hérite des méthodes par défaut et peut donc les surcharger.

Exemple de méthode par défaut : l'interface

```
public interface Addressable
{
    String getStreet();
    String getCity();
    default String getFullAddress()
    {
        return getStreet()+" "+getCity();
    }
}
```

Exemple d'utilisation

```
public class Letter implements Addressable // (voir page précédente)
{
    private String street;
    private String city;
    public Letter(String street, String city)
    {
        this.street = street;
        this.city = city;
    }
    @Override
    public String getCity()
    {
        return city;
    }
    @Override
    public String getStreet()
    {
        return street;
    }
    public static void main(String[] args)
    {
        Letter l = new Letter("Rue poulet langlet", "Neuilly Plaisance");
        System.out.println(l.getFullAddress());
    }
}
```

Cas d'utilisation des méthodes par défaut

- Par exemple, pour implémenter la nouvelle API Streams, il a fallu faire évoluer l'interface de Collections `java.util.Collection` en ajoutant les méthodes par défaut
 - `Stream<E> stream()`
 - Et
 - `Stream<E> parallelStream()`.
- Sans ces méthodes par défaut, les Collections d'implémentation telles que `java.util.ArrayList` auraient été obligées d'implémenter ces nouvelles méthodes ou bien de casser la compatibilité...

Nouvelles méthodes dans les Collections

- De nouvelles méthodes par défaut ont été ajoutées dans la hiérarchie des Collections.
- La plupart de ces méthodes utilisent les expressions lambda, simplifiant leur utilisation.
- Le développeur est libéré de la réalisation de l'itération, et peut se concentrer sur ce que fait effectivement l'itération.
- Les avantages sont :
 - Facilité de lecture du code
 - Rapidité de développement

Exemples d'utilisation des nouvelles méthodes

- `void forEach(Consumer<? super T> action)`
 - Itère chaque élément de la liste et appelle l'expression lambda spécifiée par 'action'.
 - Exemple :

```
List<Double> temperature = new ArrayList<Double>(Arrays.asList(new Double[] { 20.0,  
22.0, 22.5 }));
```

```
temperature.forEach(s -> System.out.println(s));
```

// ou bien

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

Exemples d'utilisation des nouvelles méthodes

- boolean `removeIf(Predicate<? super E> filter)`

- Itère à travers la collection et retire les éléments qui correspondent au filtre
- Exemple :

```
List<Double> temperature = new ArrayList<Double>(Arrays.asList(new Double[] {  
    20.0, 22.0, 22.5 }));
```

```
temperature.removeIf(s -> s > 22);
```

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

Exemples d'utilisation des nouvelles méthodes

■ boolean replaceAll(UnaryOperator<E> operator)

- Très utile. Remplace les éléments de la liste par le résultat d'application de l'opérateur (méthode apply)
- Exemple :

```
List<Double> temperature = new ArrayList<Double>(Arrays.asList(new Double[] { 20.0, 22.0, 22.5 }));
```

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

```
temperature.replaceAll(s->Math.pow(s, 0.5));
```

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

○ **Résultat :**

20.0

22.0

22.5

4.47213595499958

4.69041575982343

4.743416490252569

Exemples d'utilisation des nouvelles méthodes

- `V getOrDefault(Object key, V defaultValue)`
 - Retourne la valeur mappée par la clé, ou s'il n'est pas présente, la valeur par défaut fournie
 - Exemple :
`authorBooks.getOrDefault("Nouvel auteur", 0)`
`//Le nouvel auteur, si inexistant, a écrit zero livre`

LES STREAMS



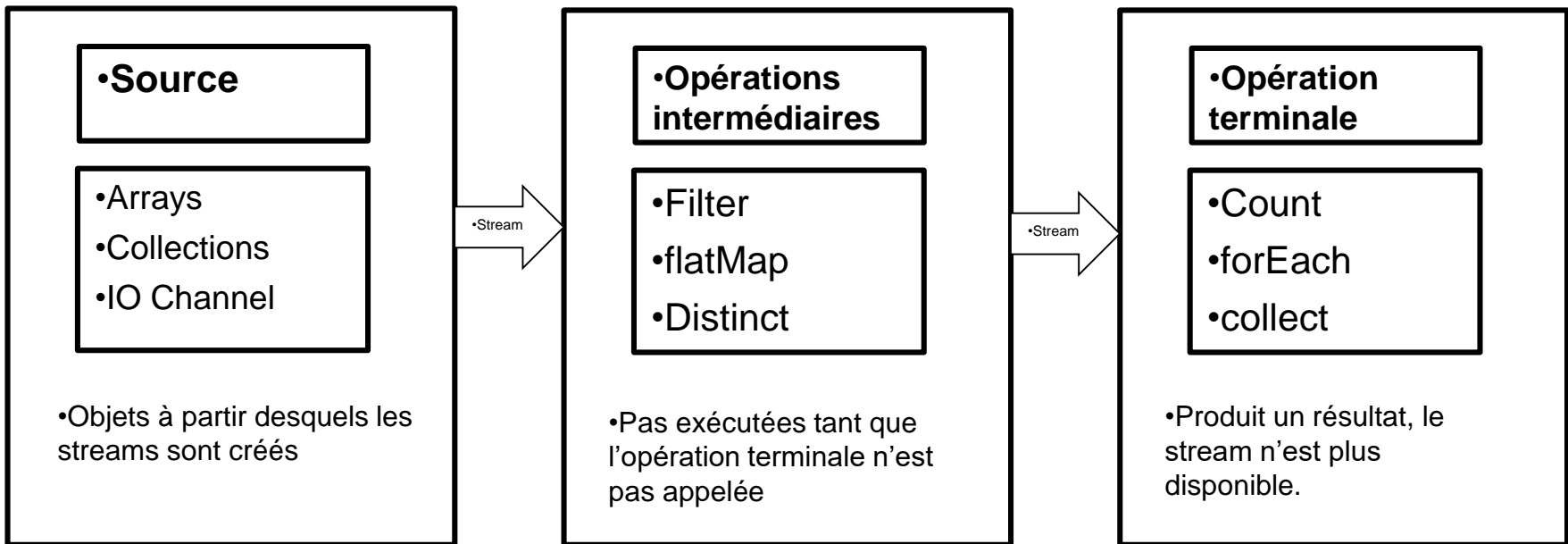
Les Streams

- Les Streams permettent de travailler sur les Collections d'une façon très simple, et permettent de travailler en programmation parallèle d'une façon également simple.
- Voici un exemple simple.
 - Ici on crée une liste de types de musiques, ensuite, on identifie le nombre de musiques commençant par un "r".
 - La 2eme ligne ci-dessous fait cela :

```
List<String> genre = new ArrayList<String>(Arrays.asList("rock", "pop", "jazz", "reggae"));
```
 - ```
long a = genre.stream().filter(s -> s.startsWith("r")).count();
```
  - ```
System.out.println(a);
```

Les Streams

- Un Stream est une séquence d'objets or de types primitifs. Les operations peuvent être réalisées en séquence ou en **parallèle** :



Point important sur l'opération intermédiaire parallèle

- Les opérations intermédiaires peuvent être stateless (sans état).
 - Elles ne gèrent aucun état pendant le traitement du flux, ce qui permet de paralléliser les traitements sans trop de surcoûts. Ces opérations se basent uniquement sur l'élément courant du flux, indépendamment des autres éléments.
- Les opérations intermédiaires peuvent être stateful (avec état).
 - Elles doivent gérer un état pour effectuer correctement leur traitement (par exemple `limit()`, `distinct()`, `sorted()`). Ces opérations peuvent devenir très coûteuses si on manipule un flux de données à la fois ordonné et en parallèle. Dans ce cas-là, il peut être préférable d'utiliser un traitement séquentiel (via l'opération `sequential()`) ou d'ignorer l'ordre des éléments (via l'opération `unordered()`).

Les Streams

- Un Stream est créé à partir d'une source, qui peut être un tableau, une collection, un IO channel, etc. Une fois le Stream créé, vous pouvez lui appliquer des opérations.
- Dans le dernier exemple, nous avons appliqué une opération de filtre sur le Stream.
- La dernière étape du cycle est appelée operation terminale. **Cette operation produit un résultat. Rien n'est démarré tant que cette dernière étape n'est pas décrite.**
- Le Stream disparaît après l'opération terminale. Cette opération peut être une opération de comptage par exemple.

Point essentiel des Streams

- Les Streams ont un cycle de vie qui consiste en :
 - Operation de creation
 - Operation intermédiaire
 - Operation terminale.

Créer des Streams

- Les Streams peuvent être créés à partir de Collections en utilisant la méthode par défaut
 - `Stream<E> stream()` de l'interface `Collection`.
- Pour les Arrays, utiliser la méthode `Arrays.stream(T[] array)`.
- Exemple (mais insuffisant !!)

```
44 List<String> strings = Arrays.asList("mercure", "vénus", "terre", "mars", "jupiter");  
45 strings.stream()
```

Autres possibilités de créer des Streams

- Créer un Stream vide

```
Stream.empty();
```

- Stream généré à partir d'une liste de valeurs :

```
Stream.of("a", "b", "c");
```

- Stream généré à partir des valeurs de 0 à 100 par exemple

```
IntStream.rangeClosed(0, 100);  
IntStream.range(0, 100);
```

- Stream généré à partir du contenu d'un tableau :

```
String[] array = { "a", "b", "c" };  
Arrays.stream(array);
```

- Stream généré à partir d'une collection (voir page précédente) :

```
Collection<String> collection = Arrays.asList("a", "b", "c");  
collection.stream();  
// Même chose, mais avec un Stream parallèle :  
collection.parallelStream();
```


Les opérations intermédiaires et terminales

- Un stream est un « pipeline d'opérations »
- 0 à N opération(s) intermédiaire(s)
 - Retourne toujours un Stream (chaînage possible)
 - Déclaratif : **leur traitement n'est réalisé que lors de l'appel de l'opération terminale**
 - Stateful ou staless
 - Exemple : filter, map
- 1 opération terminale
 - Optimise et exécute les opérations intermédiaires
 - Consomme le Stream
 - Exemple : count, forEach

Les opérations intermédiaires

- Les opérations intermédiaires gardent le Stream ouvert et permettent d'effectuer d'autres opérations.
- Les méthodes `filter()` et `map()` sont des exemples d'opérations intermédiaires.
 - Elles retournent le Stream courant, ce qui permet de chaîner plusieurs opérations.

Opération intermédiaire : le filtre

- `Stream.filter` retourne un nouveau `Stream` qui contient les éléments qui correspondent au `Predicate`!
- Exemple ici, on retourne le nombre d'éléments du `Stream` qui sont inférieurs à 3 :

```
long elementsLessThanThree = Stream.of(1, 2, 3, 4)
    .filter(p -> p.intValue() < 3).count();
assertEquals(2, elementsLessThanThree)
```

Opération intermédiaire : map

- map transforme les éléments d'un Stream en utilisant la fonction fournie par l'interface fonctionnelle `java.util.function.Function`.
- Une Function, pour rappel, accepte un argument et produit un résultat. Très utilisé par exemple dans la transformation de Liste en Map.
- Exemple ici, on l'utilise pour remplacer tous les string null par un string valant « inconnu » :

```
List<String> strings = Stream.of("one", null, "three").map(s -> {
    if (s == null)
        return "[inconnu]";
    else
        return s;
}).collect(Collectors.toList());
assertThat(strings, contains("one", "[inconnu]", "three"));
```

Opération intermédiaire : flatmap

- Stream.flatMap est la combinaison d'une map et d'une opération flat.
- Avec flatMap, vous appliquez d'abord une fonction à vos éléments, puis l'aplatissez (flat)
- Stream.map (précédent) applique uniquement une fonction au flux sans aplatir le flux.
- Exemple d'aplatissement d'un flux :
 - Avec une structure comme [[1,2,3],[4,5,6],[7,8,9]] qui a « deux niveaux », Aplatir cela signifie le transformer en une structure « à un niveau »: [1,2,3,4,5,6,7,8,9] .

Opération intermédiaire : flatmap

- Exemple d'utilisation de flatmap :

```
List<Integer> together = Stream.of(asList(1, 2), asList(3, 4))  
    .flatMap(List::stream)  
    .map(integer -> integer + 1)  
    .collect(Collectors.toList());  
assertEquals(asList(2, 3, 4, 5), together);
```

Opération intermédiaire : peek

- Très utile lors du débogage.
- Il permet de lire une donnée du Stream avant qu'une action soit engagée.
- Dans l'exemple suivant, on filtre tous les String d'une taille supérieure à 4, on appelle peek dans le Stream avant que map ne soit appelé :

```
List<String> strings = Stream.of("Badgers", "finals", "four")  
.filter(s -> s.length() >= 4).peek(s -> System.out.println(s))  
.map(s -> s.toUpperCase()).collect(Collectors.toList());  
assertThat(strings, contains("BADGERS", "FINALs", "FOUR"));
```

Opération intermédiaire : distinct

- Ne retourne qu'une occurrence des valeurs d'un Stream, si, dans le Stream, on en trouve plusieurs, élimine en fait les doublons et plus.

- Exemple :

```
List<Integer> distinctIntegers = IntStream.of(5, 6, 6, 6, 3, 2, 2)
    .distinct()
    .boxed() //sert à « boxer » les types primitifs dans leur class Wrapper
    .collect(Collectors.toList());
assertEquals(4, distinctIntegers.size());
assertThat(distinctIntegers, contains(5, 6, 3, 2));
```


Opération intermédiaire : sorted

- Permet de retourner un Stream trié dans l'ordre naturel.
- Exemple :

```
List<Integer> sortedNumbers = Stream.of(5, 3, 1, 3, 6).sorted()  
.collect(Collectors.toList());  
assertThat(sortedNumbers, contains(1, 3, 3, 5, 6));
```

Opération intermédiaire : limit

- Très utile pour limiter ou tronquer le nombre d'éléments dans un Stream à une valeur donnée.
- Exemple :

```
List<String> vals = Stream.of("limit", "by", "two").limit(2)
                        .collect(Collectors.toList());
assertThat(vals, contains("limit", "by"));
```

Opération intermédiaire : le filtre

- Soit une liste de clients

```
List<Client> clients = Arrays.asList(
    new Client("Jean", "Dupont", 41),
    new Client("Yves", "Durant", 36),
    new Client("Yvan", "Lemeux", 15)
);
```

Client	
nom	String
prenom	String
age	int

- Besoin : *afficher les clients dont le nom commence par la lettre « D »*

```
Client{nom='Dupont', prenom='Jean', age=41}
Client{nom='Durant', prenom='Yves', age=36}
```

- En Java 7

```
for (Client client : clients) {
    if (client.getNom().startsWith("D")) {
        System.out.println(client);
    }
}
```

- En Java 8

```
clients.stream()
    .filter(c -> c.getNom().startsWith("D"))
    .forEach(System.out::println);
```

Opérations terminales

- Une opération terminale doit être l'opération finale effectuée sur un Stream.
- Une fois qu'une opération terminale est appelée, le Stream est consommé et il ne peut plus être utilisé.
- Dans les pages suivantes, des exemples d'opérations terminales.

Opération terminale forEach

- Version simplifiée de la boucle for.
- Exemple :
 - `Stream.of("Hello", "World").forEach(p -> System.out.println(p));`

Opération terminale toArray

- Convertit des Array en ArrayList, et des Collections en Array
- Exemple ici d'une conversion de Stream en Array :

```
Object[] objects = Stream.of(1, 2).toArray();  
assertTrue(objects.length == 2);
```

Opération terminale reduce

- Utilisé dans le cadre statistique.
- Exemple ici d'une somme des éléments du Stream

```
int sum = IntStream.of(1, 2, 3, 4).reduce(0, (a, b) -> a + b);
```

Résultat :10

Opération terminale collect

- Convertit un Stream en un Container tel qu'une Liste.
- On peut par exemple convertir un Stream en map.
- Exemple ci-dessous, transformation d'un Stream en Set
 - Rappel : A une valeur de Set ne peut contenir d'un seul objet, ci-dessous, le Set ne contiendra que 2 objets

```
Set<String> stringSet = Stream.of("some", "one", "some", "one")  
    .collect(Collectors.toSet());
```


Opération terminale min

■ min :

- Permet de retourner la valeur minimale d'un Stream
- Exemple:

```
OptionalInt minimum = IntStream.of(1, 2, 3).min();  
assertEquals(1, minimum.getAsInt());
```
- Voir ici l'utilisation de Optional, qui permet de tester la presence ou l'absence d'un résultat.

Opération terminale max

■ max :

- Permet de retourner la valeur maximale d'un Stream

- Exemple:

```
OptionalDouble max = Stream.of(1d, 2d, 3d).mapToDouble(Double::doubleValue).max();  
assertEquals(1, max.getAsDouble(), 0);
```

- Ici on appelle mapToDouble qui applique la fonction Double::doubleValue à chaque élément retournant un DoubleStream.

Opération terminale count

■ count:

- Permet de trouver le nombre d'éléments d'un Stream.
- Exemple, retournant 1 :

```
long count = Stream.of("one").count();
```

Opération terminale anymatch

■ anymatch:

- Cherche si au moins un des éléments du Stream correspond à un Predicate fourni.
- Par exemple ci-dessous , on teste si la longueur d'un String est supérieure ou égale à 5 :

```
boolean lengthOver5 = Stream.of("two", "three", "eighteen").anyMatch(  
    s -> s.length() > 5);  
assertTrue(lengthOver5);
```

Opération terminale allmatch

■ allmatch:

- Contrôle chaque élément du Stream et cherche si chacun d'entre eux correspond au Predicate fourni.
- Par exemple ci-dessous , on contrôle si chaque élément de la liste contient bien « satellite »

```
List<String> cookies = Lists.newArrayList("pas de satellite à mercure",  
    "pas de satellite à vénus", "un satellite à la terre");  
boolean containCookies = cookies.stream().allMatch(  
    p -> p.contains("satellite"));  
assertTrue(containCookies);
```

Opération terminale nonematch

■ nonematch:

- Exactement l'inverse de la page précédente, contrôle si aucun élément du Stream ne correspond au Predicate spécifié.
- Dans l'exemple qui suit, après avoir créé un IntStream, on contrôle qu'aucun des éléments ne vaut 5.

```
boolean noElementEqualTo5 = IntStream.of(1, 2, 3)
    .noneMatch(p -> p == 5);
assertTrue(noElementEqualTo5);
```

Opération terminale findFirst

- findFirst
 - Cherche le premier élément d'un Stream

```
Optional<String> val = Stream.of("one", "two").findFirst();  
assertEquals("one", val);
```

MapReduce

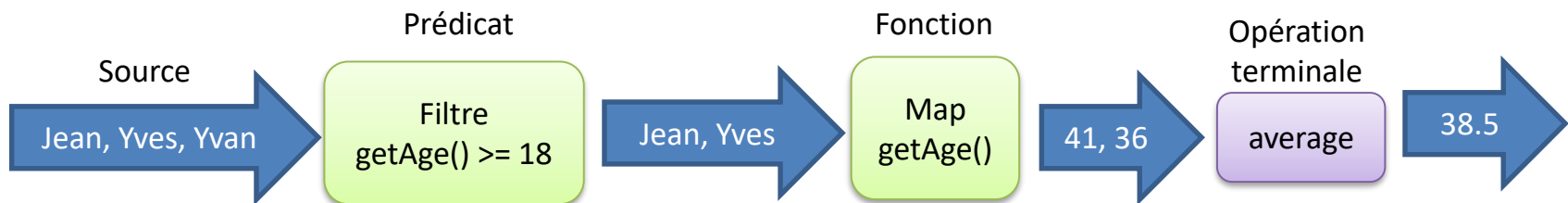
- Besoin : « Avec la même liste de clients, on souhaite calculer l'âge moyen des clients majeurs »

► En Java 7

```
int nbClient = 0, ageSum = 0;
for (Client client : clients) {
    if (client.getAge() >= 18) {
        nbClient++;
        ageSum += client.getAge();
    }
}
Double average = (double) ageSum / nbClient;
System.out.println(average);
```

► En Java 8

```
OptionalDouble average = clients.stream()
    .filter(c -> c.getAge() >= 18)
    .mapToInt(Client::getAge)
    .average();
System.out.println(average.getAsDouble());
```



Recherche

► Besoin : « *afficher le nom du 1^{er} client majeur de la liste précédente* »

► En Java 7

```
String nom = null;
for (Client client : clients) {
    if (client.getAge() >= 18) {
        nom = client.getNom();
        break;
    }
}
String msg = nom != null
    ? nom : "aucun résultat";
System.out.println(msg);
```

► En Java 8

```
String nom = clients.stream()
    .filter(c -> c.getAge() >= 18)
    .findFirst()
    .map(Client::getNom)
    .orElse("aucun résultat");
System.out.println(nom);
```

// Stream<Client>
// Optional<Client>
// Optional<String>

❖ **Optional** : conteneur pour une valeur qui peut être null

Réduction simple

► Besoin : « *rechercher le client le plus âgé* »

► En Java 7

```
Client doyen = null;
for (Client client : clients) {
    if (doyen == null) {
        doyen = client;
    } else {
        doyen = doyen.getAge() > client.getAge()
            ? doyen : client;
    }
}
if (doyen != null) System.out.println(doyen);
```

► En Java 8

```
clients.stream()
    .reduce((c1, c2) -> c1.getAge() > c2.getAge()
        ? c1 : c2)
    .ifPresent(System.out::println);
```

- ❖ Une opération de réduction combine tous les éléments d'un stream en un seul résultat
- ❖ L'opération **average()** du `IntStream` est une réduction prête à l'emploi

Collecte

► Besoin : « *récupérer une liste contenant le nom des clients* »

► En Java 7

```
List<String> noms = new ArrayList<>();
for (Client client : clients) {
    noms.add(client.getNom());
}
```

► En Java 8

```
List<String> noms = clients.stream()
    .map(Client::getNom)
    .collect(Collectors.toList());
```

❖ La méthode **collect** est une « réduction mutable »

⇒ Elle accumule les éléments d'un Stream dans un container

❖ La classe **Collectors** propose des implémentations prêtes à l'emploi de l'interface **Collector** :

⇒ `toList`, `toSet`, `toCollection`, `toMap`

Regroupement

► Besoin : « *regrouper les clients par la 1^{ière} lettre de leur nom* »

► En Java 7

```
Map<Character, List<Client>>
    map = new HashMap<>();
for (Client client : clients) {
    char initiale = client.getNom().charAt(0);
    List<Client> liste = map.get(initiale);
    if (liste == null) {
        liste = new ArrayList<>();
        map.put(initiale, liste);
    }
    liste.add(client);
}
```

► En Java 8

```
Map<Character, List<Client>> map = clients.stream()
    .collect(Collectors.groupingBy(
        c -> c.getNom().charAt(0)));
```

Autres exemples sur les Streams

- Partons de cette Liste :
 - `List<String> genre = new ArrayList<String>(Arrays.asList("rock", "pop", "jazz", "reggae"));`
- Opération terminale « Short Circuit » `allMatch`
 - `Boolean allMatch(Predicate<? Super T> predicate);`
 - Retourne true si toutes les valeurs de la collection retournent true pour l'expression lambda passée en predicate.
 - Exemple :
 - `genre.stream().allMatch(s -> !s.isEmpty())`
 - Retourne true dans ce cas. False si une entrée était vide (« » par exemple).

Autre exemple sur les Streams

- Opération terminale « Short Circuit » `anyMatch`
 - `Boolean anyMatch(Predicate<? Super T> predicate);`
 - Retourne true si une des quelconques valeurs retourne true pour l'expression lambda.
 - Exemple :
 - `genre.stream().anyMatch(s -> s.indexOf("r") == 0)`
 - Attention, retourne true dès la première occurrence trouvée.

Autre exemple sur les Streams

- Opération intermédiaire filter sans état sans modification des données source.

- `Stream<T> filter(Predicate<? super T> predicate)`
- Opération intermédiaire ne garde dans le Stream que les éléments qui correspondent au Predicate.
- Exemple :

```
System.out.println(genre.stream().filter(s -> s.length() <= 4).count());
```

//Tous les éléments passent sauf reggae, il donne ici donc 4.

//Opération sans état dans le sens où aucune information n'est stockée pour les éléments, et aucune modification des données de la liste n'est effectuée.

Autre exemple sur les Streams

- Opération intermédiaire avec état distinct
 - `Stream<T> distinct()`
 - Retourne des objets distincts. S'il y a des doublons, ils sont supprimés.
 - Repartons de l'exemple mais cette fois avec des duplicates :
 - `List<String> genre = new ArrayList<String>(Arrays.asList("rock", "pop", "jazz", "reggae", "pop"));`
 - Exemple :
 - `System.out.println(genre.stream().distinct().count());`
 ➤ Résultat : 4

- Opération intermédiaire de réduction `count`
 - `Long count()`
 - Retourne le nombre d'éléments dans le Stream.

Opérations sur les Streams

- Opération terminale sans modification des données `forEach`
 - `void forEach (Consumer<? Super T> action)`
 - Appelle l'expression Lambda pour chaque élément du Stream.
 - Exemple :
 - `genre.stream().forEach(System.out::println);`

Opérations sur les Streams

- Opération terminale de réduction reduce
 - `Optional<String> reduce(BinaryOperator<T> accumulator)`
 - Exemple :
 - **`Optional<String> combinedgenre = genre.stream().reduce((b, c) -> b.concat(",").concat(c));`**
 - Au final, ici combinedgenre va contenir une liste séparée par des virgules de tous les genres. On réduit donc ici tous les résultats en un seul.

Opérations sur les Streams

- Opération intermédiaire sans état et ne modifiant pas les données flatMap.
 - `<R> Stream <R> flatMap(Function<? Super T, ? Extends Stream<? Extends R>> mapper);`
 - Permet de réaliser un « merge » de plusieurs sous-Streams en un seul.
 - Exemple :
 - `Map<String, List<String>> artists = new HashMap<String, List<String>>();`
 - `artists.put("rock", new ArrayList<String>(Arrays.asList("rockArtistA", "rockArtistB")));`
 - `artists.put("pop", new ArrayList<String>(Arrays.asList("popArtistA", "popArtistB")));`
 - `artists.put("jazz", new ArrayList<String>(Arrays.asList("jazzArtistA", "jazzArtistB")));`
 - `artists.put("reggae", new ArrayList<String>(Arrays.asList("reggaeArtistA", "reggaerockArtistB")));`
 - `genre.stream().flatMap(s -> artists.get(s).stream()).forEach(s -> System.out.print(" " + s));`
 - `// prints rockArtistA rockArtistB popArtistA popArtistB jazzArtistA jazzArtistB`
 - `//reggaeArtistA reggaerockArtistB popArtistA popArtistB`

LES DATES



L'API DateTime

- L'API de Date a été revue en Java 8. Elle utilise le calendrier Grégorien.
- Les classes se trouvent dans le package `java.time`.

Classes importantes de DateTime

- `java.time.Period`:
 - Représente la partie date de datetime. Par exemple, 1 an, 2 mois and 5 jours
- `java.time.Duration`:
 - Représente la partie time de datetime. Par exemple, '29 secondes'
- `java.time.Instant`:
 - Représente un instant. Stocke le nombre de secondes pour une époque donnée, ainsi que le nombre de nanosecondes;
- `java.time.LocalDate`:
 - Stocke une date en terme d'année-mois-jours.years-months-days.

Classes importantes de DateTime

- `java.time.LocalDateTime`:
 - Stocke l'heure
- `java.time.LocalDateTime`:
 - Stocke la date et l'heure
- `java.time.ZonedDateTime`:
 - Stocke la `LocalDateTime` et l'info `TimeZone` en tant que objet `ZoneOffset`.
Peut être convertie en heure locale.
- `java.time.ZoneOffset`:
 - Stocke le décalage par rapport au temps universel.

Utilisation

- Création d'une date locale :
Instant now = Instant.now();
 - Contient : 2016-05-02T13:21:50.809Z → temps universel
 - Ne contient pas d'info de type timezone
- Secondes de type « Epoch »
System.out.println(now.getEpochSecond());
 - Contient 1462195507
 - Nombre de secondes depuis le 1^{er} janvier 1970

Utilisation

- Ajouter un temps à un Instant :
Instant tomorrow = now.plus(1,ChronoUnit.DAYS);
 - Contient 2016-05-03T13:28:06.580Z
 - Ici on a ajouté un jour.
 - On peut ajouter des : NANOS, MICROS, MILLIS, SECONDS, MINUTES, HOURS, HALF_DAYS, DAYS
- Soustraction d'un temps à un Instant :
Instant yesterday = now.minus(1,ChronoUnit.HALF_DAYS);
- Comparer deux Instants :
System.out.println(now.compareTo(tomorrow)); // affiche 1
 - Retourne -1 si la date passée est après, 1 si elle est avant

Utilisation

- Contrôler si une date est après l'autre

`System.out.println(now.isAfter(yesterday));` → affiche true

- Créer une `LocalDateTime`

`LocalDateTime localDateTime = LocalDateTime.now();`

- Affiche 2016-05-02T15:46:51.919, deux heures après le TU en France en été

- Obtenir le jour d'une semaine

`System.out.println(DayOfWeek.from(localDateTime));`

Affiche MONDAY

`System.out.println(DayOfWeek.from(localDateTime).getDisplayName(TextStyle.FULL, Locale.FRENCH));`

Affiche LUNDI

Utilisation

- Obtenir le jour de l'année à partir d'une DateTime.

```
System.out.println(localDateTime.get(ChronoField.DAY_OF_YEAR));
```

- Affiche 123

- Obtenir la partie LocalTime d'une LocalDateTime.

```
System.out.println(localDateTime.toLocalDate());
```

- Créer une LocalDateTime à partir d'une année, mois, jour, heure, minute.

```
System.out.println(LocalDate.of(2016, 04, 10, 10, 0));
```

- Affiche : 2016-04-10T10:00

Utilisation

- Créer une `LocalDateTime` à partir de l'analyse d'une String

```
LocalDateTime parsedLocalDateTime = LocalDateTime.parse("2014-01-01T11:00");
```

- Créer une `ZonedDateTime` :

```
ZonedDateTime zonedDateTime = ZonedDateTime.now();
```

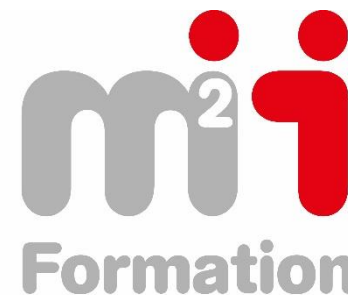
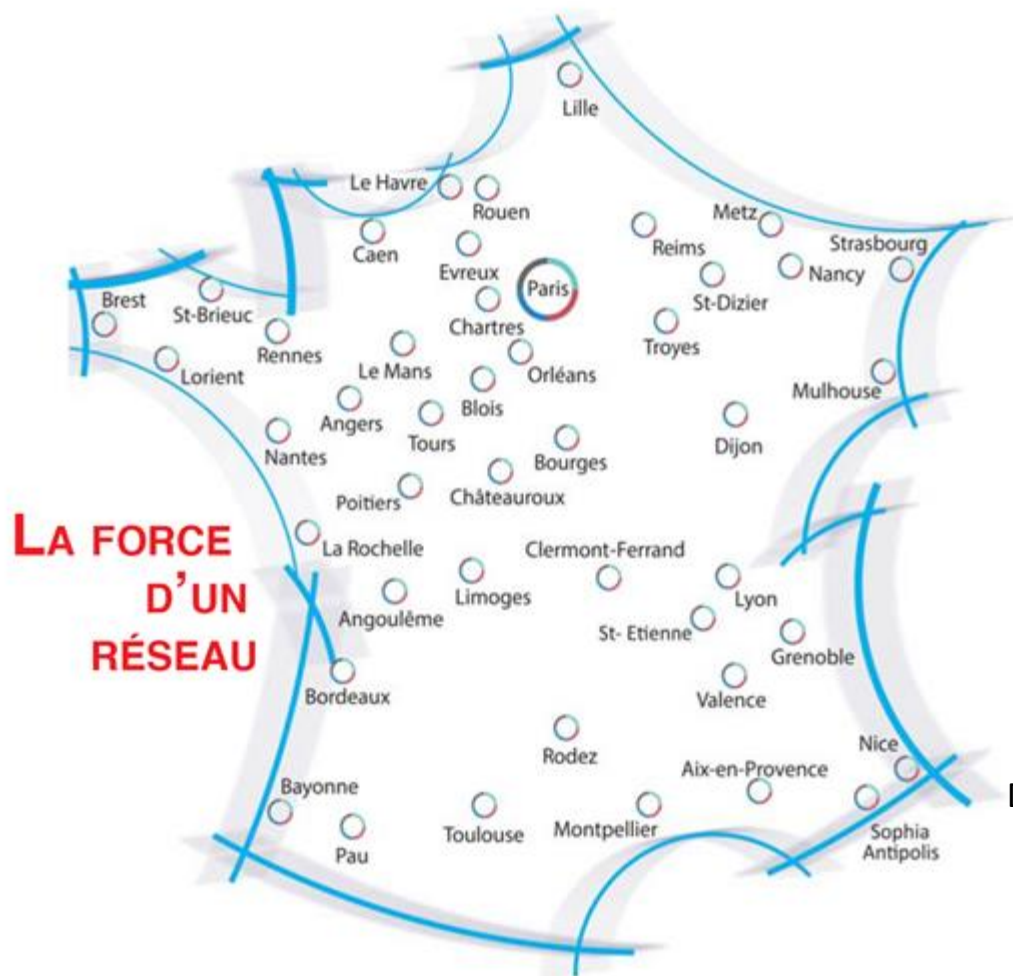
```
System.out.println(zonedDateTime);
```

Affiche : 2016-05-02T16:09:02.526+02:00[Europe/Berlin]

- Parser et formater une `DateTime` en utilisant `DateTimeFormatter` :

```
System.out.println(zonedDateTime.format(DateTimeFormatter.ofPattern("'le' dd 'jour de' MMM 'année' YYYY 'dans la zone' z")));
```

Affiche : le 02 jour de mai de année 2016 dans la zone CEST



Notre expertise est votre avenir

► **N°Azur 0 810 007 689**

PRIX D'UN APPEL LOCAL DEPUIS UN POSTE FIXE

Découvrez également l'ensemble des stages à votre disposition sur notre site

<http://www.m2information.fr>