

# Java 8 - quelques nouveautés

# Introduction

- Comme Java 5, à son époque, Java 8 introduit un certains nombres de nouveautés
  - langage
  - compilateur
  - librairies
  - outils
  - JVM

# Langage - les interfaces fonctionnelles

- Interface qui possède une seule méthode abstraite
  - l'annotation `@FunctionalInterface` permet au compilateur de vérifier que l'interface ne comporte qu'une seule méthode abstraite

```
@FunctionalInterface
public interface Operation
{
    int compute(int x, int y);
}
```

- peut posséder des méthodes par défaut
- les méthodes de la classe `Object` ne sont pas prises en compte

# Langage - les interfaces fonctionnelles

- Objectif : définir une signature de méthode qui pourra être utilisée pour passer en paramètre
  - une référence vers une méthode statique
  - une référence vers une méthode d'instance
  - une référence vers un constructeur
  - une expression lambda

# Langage - les interfaces fonctionnelles

- Classiquement, cette interface est implémentée par une classe
  - ou par création d'une classe anonyme

```
public class Addition implements Operation
{
    @Override
    public int compute(int x, int y)
    {
        return x+y;
    }
}
```

- Une classe utilise l'interface

```
public class Calcul
{
    public static int compute(int x, int y, Operation op){
        return op.compute(x,y);
    }
}
```

# Langage - les interfaces fonctionnelles

- Exemple

```
public class CalcuLambda
{
    public static void main(String[] args)
    {
        int r = Calcul.compute(2,3, new Addition());

        r = Calcul.compute(5, 6, new Operation()
        {
            @Override
            public int compute(int x, int y)
            {
                return x-y;
            }
        });
    }
}
```

# Langage - les expressions lambda

- Une expression lambda permet d'implémenter une nouvelle opération sans avoir à créer une nouvelle classe
  - le type des arguments et de la valeur de retour doivent correspondre à ceux de la méthode de l'interface fonctionnelle

```
r = Calcul.compute(10, 2, (x,y) -> x*y);
```

# Langage - les expressions lambda

- Fonction lambda, fonction anonyme, ou closure
  - fonction n'ayant pas de nom
  - les instructions de la fonction lambda suivent une syntaxe particulière
  - il est possible de
    - la stocker lambda dans une variable
    - la retourner depuis une méthode, ou une autre fonction
- JSR 335



# Langage - les expressions lambda

- Expression lambda

```
String[] tabString = {"a","b","c","d","e","f"};  
Arrays.asList(tabString).forEach(s -> System.out.println(s));
```

- une expression lambda est une notation abrégée d'une méthode fonctionnelle
- une fonction lambda peut renvoyer une valeur
  - si lambda à une seule expression, la valeur renvoyée est l'expression elle-même
  - si lambda à un bloc de code, la valeur renvoyée est fournie par un ou plusieurs `return`

# Langage - les expressions lambda

- Syntaxe de base `arguments -> corps`
  - nouvel opérateur, l'opérateur flèche: ->
    - tiret suivi du signe >
  - les arguments doivent être entre ( )
    - si pas d'argument, parenthèses vides
    - si un seul argument, les parenthèses peuvent être omises
    - le type des arguments peut être indiqué
  - le corps
    - est une simple expression, la valeur de cette expression est alors renvoyée
    - un bloc d'instructions, entre { }, peut contenir un ou plusieurs `return`

# Langage - les expressions lambda

- Exemples

```
() -> System.out.println("hi")

(x,y) -> x+y

(Point p, int x) -> p.add(x)

Arrays.asList(tabString).sort((e1,e2)->e1.compareTo(e2));

(x,y) -> {if(x<y)
        return y;
        else
        return x;}

Operation op = (x,y) -> x % y;
```

# Langage - interface méthode par défaut

- Java 8 permet d'ajouter des méthodes par défaut dans une interface
  - cette méthode contient du code
  - objectif : permettre l'ajout de nouvelles méthodes dans une interface tout en préservant le code déjà écrit
  - la méthode par défaut peut être redéfinie

# Langage - interface méthode par défaut

- Exemple

```
public interface Roulable
{
    default void rouler(){
        System.out.println("Objet pouvant rouler par défaut");
    }
}
```

```
public class De implements Roulable
{
    @Override
    public void rouler()
    {
        System.out.println("valeur obtenue : "+((int)(Math.random()*6)+1));
    }
}
```

```
public class Ballon implements Roulable
{
}
```

# Langage - interface méthode statique

- Java 8 permet de déclarer une méthode statique dans une interface
  - l'implémentation doit être fournie

```
public class RoulableFactory
{
    static Roulable create(Supplier<Roulable> supplier){
        return supplier.get();
    }
}
```

```
public static void main(String[] args)
{
    Roulable rounable = RoulableFactory.create(Ballon::new);
    rounable.rouler();
}
```

# Langage - référence à une méthode

- Syntaxe permettant de référencer directement une méthode ou un constructeur
  - opérateur ::
  - référence à un constructeur par défaut
    - sans paramètre
  - référence à une méthode
    - statique ou non

```
System.out::println
```

est équivalent à

```
o -> System.out.println(o)
```

# Langage - référence à une méthode

- Exemples

```
public class Voiture
{
    public static Voiture create(final Supplier<Voiture> supplier){
        return supplier.get();
    }

    public void rouler(){
        System.out.println("Rouler");
    }
}
```

```
public static void main(String[] args)
{
    List<Voiture> voitures = new ArrayList<>();
    voitures.add(Voiture.create(Voiture::new));
    voitures.add(Voiture.create(Voiture::new));
    voitures.add(Voiture.create(Voiture::new));

    voitures.forEach(Voiture::rouler);
}
```



# Librairies - Optional

- Objectifs :
  - solution au `NullPointerException`
  - initié dans Google Guava
- `Optional` est un container contenant ou non une valeur
  - `isPresent()` renvoie `true` si la valeur est présente
  - `get()` renvoie la valeur
  - plusieurs autres méthodes permettent de retourner ou exécuter des traitements si la valeur n'est pas présente

# Librairies - Optional

- Exemples

```
Optional< String > nom = Optional.ofNullable( null );  
System.out.println( "Nom présent? " + nom.isPresent() );  
System.out.println( "Nom : " + nom.orElseGet( () -> "[none]" ) );  
System.out.println( nom.map( s -> "Bonjour " + s + "!" ).orElse( "Bonjour inconnu !" ) );
```

```
Nom présent? false  
Nom : [none]  
Bonjour inconnu !
```

```
Optional< String > nom = Optional.ofNullable("Gaston" );  
System.out.println( "Nom présent? " + nom.isPresent() );  
System.out.println( "Nom : " + nom.orElseGet( () -> "[none]" ) );  
System.out.println( nom.map( s -> "Bonjour " + s + "!" ).orElse( "Bonjour inconnu !" ) );
```

```
Nom présent? true  
Nom : Gaston  
Bonjour Gaston!
```

# Librairies - Stream API

- Programmation fonctionnelle
- Permet un code clair, concis
- Simplifie le traitement sur les grands volumes d'informations
  - collections, flux entrée/sortie
- `Stream` est une interface paramétrée
  - `package java.util.stream`

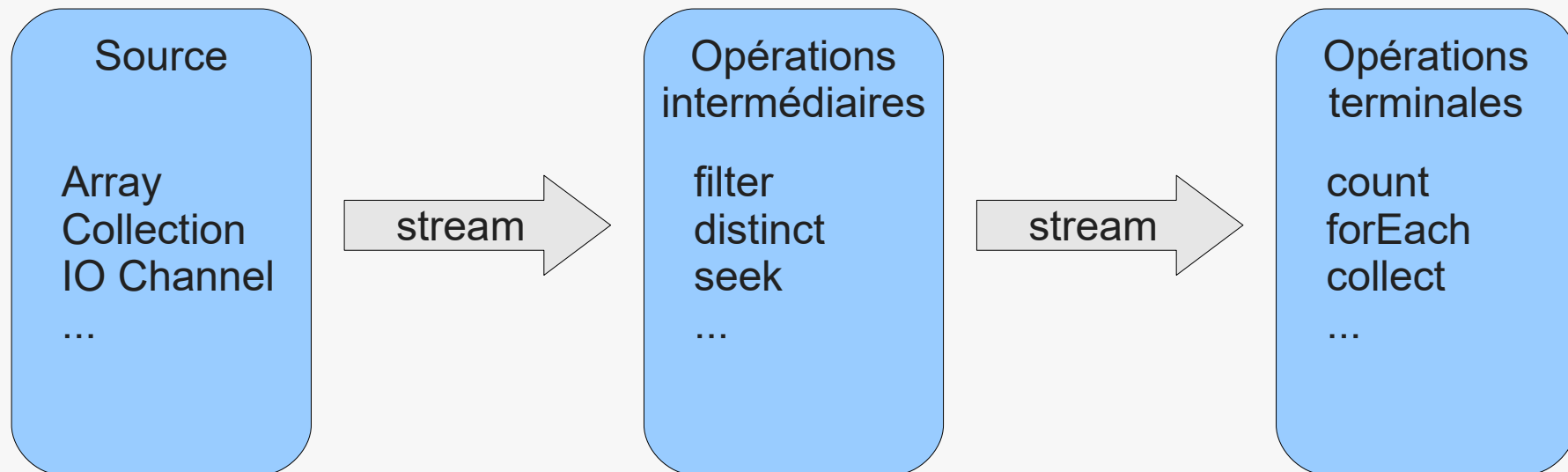
```
public interface Stream<T> extends BaseStream<T, Stream<T>>{  
    ...  
}
```

# Librairies - Stream API

- Un stream permet de traiter les données
  - en parallèle - exploitation des processeurs multicœurs
  - en pipeline - évite les calculs intermédiaires
- Permet de définir des opérations
  - le `Stream` ne possède pas les données qu'il traite (source)
  - il ne modifie pas les données qu'il traite
  - il traite les données en "une passe"

# Librairies - Stream API

- Cycle de vie d'un stream



# Librairies - Stream API

- Opération `forEach()`
  - effectue une action sur chaque donnée du stream
  - opération terminale (renvoie `void`)

```
List<Point> points = new ArrayList<>();  
initPoints(points);  
  
Stream<Point> stream = points.stream();  
  
stream.forEach(p -> System.out.println(p));
```

- note : l'expression lambda peut être remplacée par une référence vers la méthode `println`

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

# Librairies - Stream API

- Opération `peek()`
  - opération intermédiaire (renvoie `Stream<T>`)
  - doit être suivie par une opération terminale
    - opération `count()` dans l'exemple - renvoie un `long`

```
List<String> types = new ArrayList<>(Arrays.asList("rock", "trip hop", "jazz", "pop", "rock"));

long nb = types.stream().peek(t -> System.out.println(t)).count();
System.out.println("Nb éléments : "+nb);
```

# Librairies - Stream API

- Opération `filter()`
  - filtre selon un prédicat
  - opération intermédiaire
  - ne crée pas un nouveau jeu de données
    - les streams ne portent pas les données

```
List<String> types = new ArrayList<>(Arrays.asList("rock", "trip hop", "jazz", "pop", "rock"));  
types.stream().filter(s -> s.length() > 3 ).forEach(System.out::println);
```



# Librairies - Stream API

- Les opérations intermédiaires ne sont pas effectuées tant que l'opération terminale n'est pas appelée
  - "lazy evaluation"

```
List<String> types = new ArrayList<>(Arrays.asList("rock", "trip hop", "jazz", "pop", "rock"));
List<String> result = new ArrayList<>();

// BAD PRACTICE
types.stream().filter(s -> s.length() > 3 ).peek(result::add);

result.stream().forEachOrdered(System.out::println);
```

Pas d'opération terminale

Aucun affichage  
La liste `result` est vide

# Librairies - Stream API

- Opération `collect()`
  - opération terminale
  - transforme le stream en collection résultat
    - `List`, `Set`, `Map`

```
List<String> types = new ArrayList<>(Arrays.asList("rock", "trip hop", "jazz", "pop", "rock"));

List<String> result = types.stream().filter(s -> s.length() > 3 ).collect(Collectors.toList());

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

```
rock
trip hop
jazz
rock
```

# Librairies - Stream API

- Opération `map()`
  - retourne un nouveau stream

```
List<Point> points = new ArrayList<>();  
initPoints(points);  
  
Stream<Point> stream = points.stream();  
Stream<Integer> xs = stream.map(p -> p.getX());  
  
xs.forEach(System.out::println);
```

# Librairies - Stream API

- La librairie Stream API est très riche
  - comptage, min, max, regroupement, réduction, tri, ...
- Parcourez la documentation

# Librairies - package `java.util.function`

- `java.util.function` contient un ensemble d'interfaces fonctionnelles
- Quatre interfaces principales
  - `Consumer`
  - `Supplier`
  - `Predicate`
  - `Function`

# Librairies - package `java.util.function`

- Les méthodes de `Stream` utilisent les interfaces fonctionnelles de `java.util.function`
  - `forEach(Consumer)`
  - `peek(Consumer)`
  - `filter(Predicate)`
  - `map(Function)`

# Librairies - Consumer

- Abstraction d'une opération acceptant un seul argument et ne renvoyant aucun résultat
  - méthodes
    - `accept(T t)` : exécute une opération sur le paramètre
      - méthode abstraite
    - `andThen(Consumer<? super T> after)` : exécute l'opération `accept` puis l'opération `accept` sur `after`

# Librairies - Consumer

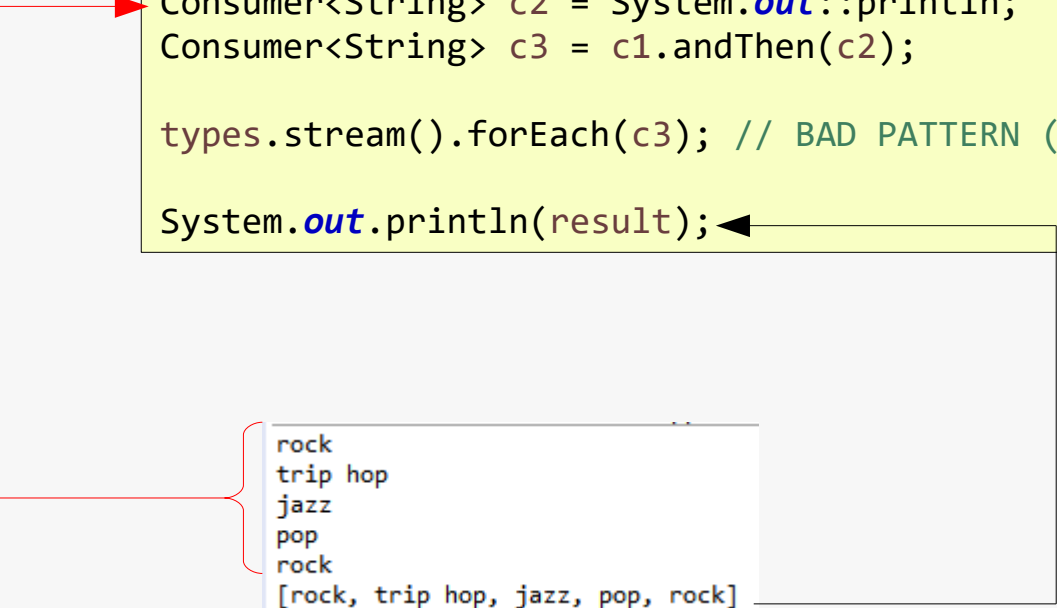
- Exemple

```
List<String> types = new ArrayList<>(Arrays.asList("rock", "trip hop", "jazz", "pop", "rock"));
List<String> result = new ArrayList<>();

Consumer<String> c1 = result::add;
Consumer<String> c2 = System.out::println;
Consumer<String> c3 = c1.andThen(c2);

types.stream().forEach(c3); // BAD PATTERN (result::add)

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



```
rock
trip hop
jazz
pop
rock
[rock, trip hop, jazz, pop, rock]
```



# Librairies - Supplier

- Abstraction d'une fonction retournant une valeur
  - le `Supplier` produit une valeur
    - abstraction d'un factory
  - le `Consumer` consomme une valeur
  - méthode
    - `T get()` : produit une valeur de type `T`
      - méthode abstraite

# Librairies - Supplier

- Exemples

```
Supplier<Point> pointSupplier = Point::new;
```

```
Point p1 = pointSupplier.get();  
Point p2 = pointSupplier.get();
```

```
Supplier<Point> pointSupplier = new Supplier<Point>(){  
    private int i = 0;  
    @Override  
    public Point get(){  
        i++;  
        return new Point(i,i);  
    }  
};
```

```
Point p1 = pointSupplier.get();  
Point p2 = pointSupplier.get();
```

# Librairies - Function

- Abstraction d'une fonction acceptant un paramètre et retournant un résultat
  - `Function<T, R> : T argument, R résultat`
  - méthodes
    - `R apply(T t) : applique la fonction à l'argument t`
    - `<T> Function<T, T> identity() : retourne toujours l'argument`
    - `<V> Function<R, V> andThen(Function<R, V> after) : retourne une fonction qui applique la fonction, puis applique after sur le résultat`
    - `<V> Function<V, R> compose(Function<V, T> before) : retourne une fonction qui applique before, puis applique la fonction sur le résultat de before`

# Librairies - Function

- Exemple

```
List<Point> points = new ArrayList<>();  
initPoints(points);  
  
Function<Point,Integer> f1 = p -> p.getX();  
  
Stream<Point> stream = points.stream();  
Stream<Integer> xs = stream.map(f1);  
  
xs.forEach(System.out::println);
```

# Librairies - Predicate

- Abstraction d'un prédicat (fonction retournant `true` ou `false`)
  - méthodes
    - `test(T t)` : évalue le prédicat sur `t`
      - méthode abstraite
    - `and()`, `or()`, `isEqual()`, `negate()` : permet l'évaluation et le chaînage des prédicats

# Librairies - Predicate

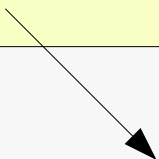
- Exemple

```
List<String> types = new ArrayList<>(Arrays.asList("rock", "trip hop", "jazz", "pop", "rock"));

Predicate<String> p1 = s -> s.length() > 3;
Predicate<String> p2 = s -> s.length() < 5;

List<String> result = types.stream().filter(p1.and(p2)).collect(Collectors.toList());

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



[rock, jazz, rock]

# Librairies - Date Time API

- Java 8 améliore nettement la manipulation des dates
  - Date-Time API - JSR 310
    - issu du projet Joda-Time
  - `package java.time`
  - cf. <http://docs.oracle.com/javase/tutorial/datetime/iso/index.html>
  - possibilités
    - analyse et formatage de dates
    - calcul sur les dates
    - gestion des dates au format ISO

# Librairies - Date Time API

- Exemples

```
Clock clock = Clock.systemDefaultZone();  
System.out.println(clock);  
System.out.println(clock.instant());
```

```
SystemClock[Europe/Paris]  
2015-11-25T15:11:16.147Z
```

```
LocalDate today = LocalDate.now();  
LocalDate payDay = today.with(TemporalAdjusters.lastDayOfMonth()).minusDays(2);  
System.out.println(payDay);  
  
Locale locale = Locale.getDefault();  
System.out.println(today.getDayOfWeek().getDisplayName(TextStyle.FULL, locale));  
  
LocalDate anotherDay = LocalDate.of(2016,1,1);  
System.out.println(anotherDay.getDayOfWeek().getDisplayName(TextStyle.FULL, locale));
```

```
2015-11-28  
mercredi  
vendredi
```