Développement JAVA

JAVA : LES NOUVEAUTES

Objectifs pédagogiques

- Connaître les nouveautés de Java et les mettre en pratique

Les nouveautés

Sommaire

- **L'évolution du langage**
  - JSR 335 : Les méthodes par défaut dans les interface
  - JSR 335 : Les expressions Lambda
  - JSR 335 : Les interfaces fonctionnelles
  - JEP 120 : Les annotations Répétable
  - JEP 395 : Les records (Java 16)

- **Les dépendances**
  - Les Streams
  - JSR 310 : Les Dates

Développement JAVA

LE LANGAGE

Les interfaces et default 1/4

- Introduction du mot clef **default** dans les interfaces
  - pour définir une implémentation par défaut
  - utile pour mettre en place les interfaces fonctionnelles, avec une écriture *expression lambda*
- Pratique pour les développeurs d'API

```
package com.formation.app;

public interface A {

   default void printSomething() {
      System.out.println("something");
   }
}
```

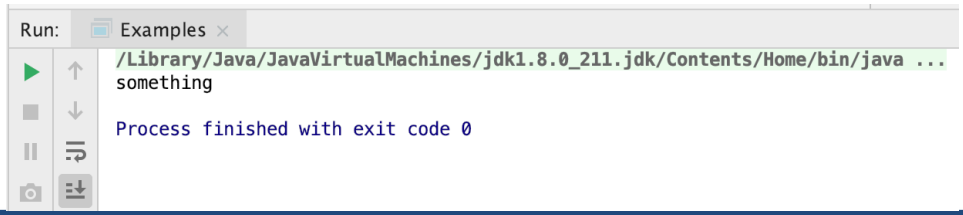
```
package com.formation.app;

public class Implementation implements A {}
```

```
package com.formation.app;

public class Examples {

   public static void main(String[] args) {
      new Implementation().printSomething();
   }
}
```



Les interfaces et default 2/4

- Deux interfaces implémentent la même méthode par défaut
- Une classe implémente ces deux interfaces

- **Impossible de compiler**

```
Error:(3, 8)  java: class com.formation.app.Implementation inherits unrelated defaults for
              printSomething() from types com.formation.app.A and com.formation.app.B
```

```java
package com.formation.app;

public interface B {
    default void printSomething() {
        System.out.println("something else");
    }
}
```

```java
package com.formation.app;

public class DoesNotCompile implements A,B {}
```

Les interfaces et default 3/4

- Obligation de surcharger la méthode dans l'implémentation pour résoudre le conflit

```
package com.formation.app;

public class Implementation implements A,B {

    @Override
    public void printSomething() {
        System.out.println("I can priint what I want !");
    }
}
```

Les interfaces et default 4/4

- Il est aussi possible de faire référence à une méthode par défaut

```
package com.formation.app;

public class Implementation implements A,B {

    @Override
    public void printSomething() {
        A.super.printSomething();
    }
}
```

```
package com.formation.app;

public class Implementation implements A,B {

    @Override
    public void printSomething() {
        B.super.printSomething();
    }
}
```

Les expressions Lambda

- Une expression lambda est une sorte de méthode “anonyme”
  - Évolutions très attendues

Avant Java 8

- Création d’une classe anonyme qui implémentait une interface avec une seule méthode abstraite **(Single Abstract Method)**

```
List<Integer> numbers = Arrays.asList(10, 1, 1000, 100);

Collections.sort(numbers, new Comparator<Integer>() {
   @Override
   public int compare(Integer a, Integer b) {
      return a.compareTo(b);
   }
});
```

Avec Java 8

- Le compilateur est capable de trouver les types des paramètres, pas besoin de les préciser
- Si la méthode tient sur une ligne, on peut omettre les **{}** et le **return**

```
Collections.sort(numbers, (a, b) -> a.compareTo(b));
```

Les interfaces fonctionnelles

- Les expressions lambda ne sont pas si **« anonymes »**
- Correspondent à des types spécifiés par des interfaces avec exactement une méthode abstraite
- C'est pour cela que les classes anonymes implémentant une interface de type SAM peuvent être remplacées par des expressions lambda
- Java 8 propose l'annotation **@FunctionalInterface** pour s'assurer qu'une interface ne déclare qu'une seule méthode abstraite

```
package com.formation.app;

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

Les interfaces fonctionnelles fournies

- De nombreuses **@FunctionalInterface** sont disponibles par défaut
- D'anciennes interfaces de l'API ont été migrées
  - Comparable et Runnable par exemple
  - Function
  - BiFunction
  - UnaryOperator
  - BinaryOperator
  - Predicate
  - BiPredicate
  - Supplier
  - Consumer
  - Comparator
  - etc

```
package java.util.function;
```

Function

- prend un argument et retourne un résultat

```
Function<Integer, Integer> multiplication = i1 -> i1 * 5;
int result = multiplication.apply(5);
System.out.println(result);
```

```
25
```

BiFunction

- Spécialisation de Function : deux arguments et retourne un résultat

```
BiFunction<Integer, String, String> concat = (Integer i, String s) -> s + ": " + i;
System.out.println(concat.apply(100, "price"));
```

```
price: 100
```

UnaryOperator

- *Function spécialisée* qui prend un argument et retourne un résultat du même type

```
UnaryOperator<String> minuscule = (value) -> value.toLowerCase();
System.out.println(minuscule.apply("TEST"));
```

```
test
```

BinaryOperator

- *BiFunction spécialisée* dont les paramètres et le résultat partagent le même type

```
BinaryOperator<String> concatStr = (s1, s2) -> s1.concat(" ").concat(s2);
System.out.println(concatStr.apply("Hello", "World"));
```

```
Hello World
```

Predicate

- prend un argument et retourne un booléen

```
String valueTestStr = "Boris";
Predicate<String> predicate = (x) -> x.isEmpty();
System.out.println(predicate.test(valueTestStr));
```

```
false
```

BiPredicate

- Spécialisation de *Predicate* : deux arguments et retourne un booléen

```
BiPredicate<String, Integer> filter = (x, y) -> x.length() == y;

boolean result = filter.test("Boris", 5);
System.out.println(result);  // true

boolean result2 = filter.test("Sauvage", 10);
System.out.println(result2); // false
```

```
true
false
```

Supplier

- Un *Supplier* ne prends pas d'argument et produit un résultat

```
Supplier<String> emptyString = () -> "Result";
System.out.println(emptyString.get());
```

```
Result
```

Consumer

- Un *Consumer* prend un argument mais ne retourne pas de résultat

```
Consumer<String> print = s -> System.out.println(s);
Consumer<String> hello = s ->
System.out.printf("Hello %s !", s);

print.accept("String to display");
hello.accept("Boris");
```

```
String to display
Hello Boris !
```

Java 8 : Les nouveautés

Comparator

- Les *Comparator* sont devenus des *@FunctionalInterface*

```
int a = ascending.compare(1000, 50);
System.out.println(a);

int b = ascending.compare(1000, 1000);
System.out.println(b);

int c = ascending.compare(1000, 5000);
System.out.println(c);

int asc = ascending.compare(1, 2);
System.out.println(asc);

Comparator<Integer> descending = ascending.reversed();
int desc = descending.compare(1, 2);
System.out.println(desc);
```

Pour résumer

- Sans retour : utilisation d'un **Consumer**
- Retourner un booléen : utilisation d'un **Predicate**
- Retourner une valeur sans prendre d'arguments : utilisation d'un **Supplier**
- Produire un numérique primitif : utilisation d'un **(Type)ToIntFunction**

- Fonction avec deux arguments : utilisation d'une **Bi(….)**
- Fonction retournant une valeur de même type que son argument : **UnaryOperator**
- Fonction sans retour et avec un argument primitif : **Obj(Int| Double | Long) Consumer**

Exercice ensemble 1/1

- Écrire un programme qui permet de créer un IntStream de 6 entiers aléatoires (valeur entre 0 et 20), puis réaliser les opérations suivantes :

1. Trier cette liste par ordre croissant et afficher le résultat avec la méthode forEach()

2. Trier cette liste par ordre croissant et afficher que le premier élément

3. Afficher la somme des nombres supérieur à 3.

Les références de méthodes

- Introduction du mot clef `::` pour extraire des références de méthodes

```
Function<Integer, String> toString1 = n -> String.valueOf(n);
Function<Integer, String> toString2 = String::valueOf;
```

- Utile pour remplacer des expressions lambda appelant des méthodes existantes

- Il est possible de faire référence à des méthodes d'instance et à des constructeurs :

```
String hello = "hello";
Predicate<String> startsWith = hello::startsWith;
System.out.println(startsWith.test("he"));
```

```
Supplier<String> newString = String::new;
Comment comment = Comment::new;
```

Comment is not a functional interface

Déterminer les méthodes compatibles

Fonctionnement statique

- La méthode possède la même signature que la méthode unique de la **@FunctionalInterface**
- **On peut écrire les trois étapes suivantes :**

```
Function<Integer, String> toString1 = new Function<Integer, String>() {
    @Override
    public String apply(Integer integer) {
        return String.valueOf(integer);
    }
};

Function<Integer, String> toString2 = integer -> {
    return String.valueOf(integer);
};

Function<Integer, String> toString3 = String::valueOf;
```

1

2

3

Statique

Déterminer les méthodes compatibles

Fonctionnement avec les instance

- Ce type de référence permet de capturer une instance qui sera utilisée lors de l'évaluation de l'expression lambda

- **On peut écrire les trois étapes suivantes :**

```
String hello = "hello";
Predicate<String> startsWith1 = new Predicate<String>() {
   @Override
   public boolean test(String s) {
      return hello.startsWith(s);
   }
};

Predicate<String> startsWith2 = s -> hello.startsWith(s);

Predicate<String> startsWith3 = hello::startsWith;
```

1

2

3

Instance

Déterminer les méthodes compatibles

Fonctionnement avec les instance arbitraire

- Ce type de référence permet de pointer des méthodes sur des instances découvertes à l'exécution
  - l'instance en question sera le premier argument de l'expression lambda et le reste des paramètres seront ceux passés à la méthode en référence

- **On peut écrire les trois étapes suivantes :**

```
BiFunction<String, String, Integer> concat1 = new BiFunction<String, String, Integer>() {
   @Override
   public Integer apply(String self, String argument) {
      return self.compareToIgnoreCase(argument);
   }
};

BiFunction<String, String, Integer> concat2 = (self, argument) -> self.compareToIgnoreCase(argument);

BiFunction<String, String, Integer> concat3 = String::compareToIgnoreCase;
```

1

2

3

Instance arbitraire

Les streams

- Un **Stream** est une séquence d'éléments sur laquelle on peut effectuer des opérations

- Un **Stream** se compose
  - d'une source *(un tableau, une collection, etc*)
  - d'une ou plusieurs opérations intermédiaires, voir aucune *(transformation du Stream en un autre via filter par exemple) et d'une opération terminale (qui produit le résultat)*

- Les calculs ne sont effectués qu'à l'initialisation de l'opération finale et la source est consommée que si c'est nécessaire

Java 8 : Les nouveautés

Création des Streams

- Il est possible créer **un Stream** depuis une collection

```
List<Person> persons = List.of(new Person("Boris", "S", 30), new Person("Giovanna", "E", 20), new Person("Arthur", "D", 15));
System.out.println(persons.stream().count());
```

- Il est possible créer **un Stream** numérique, ou utiliser de nombreuses autres méthodes

```
int sumInt = IntStream.range(0, 10).sum();
System.out.println(sumInt);

double sumDouble = DoubleStream.of(1.5d, 3.5d).sum();
System.out.println(sumDouble);

long sumLong = LongStream.range(100l, 200l).sum();
System.out.println(sumLong);

Stream.of("a", "b", "c").forEach(System.out::println);

Stream.builder().add("a").add("b").add("c")
.build()
.forEach(System.out::println);
```

```
// Stream infini avec une opération stoppante
Random random = new Random();
Stream.generate(() -> random.nextInt())
      .limit(10)
      .forEach(System.out::println);

new Random().ints()
      .limit(10)
      .forEach(System.out::println);
```

Streams : les opérations

- **forEach**
- **filter**
- **sorted**
- **map**
- **allMatch, anyMatch, noneMatch**
- **count**
- **sum**
- **reduce**
- **collect**
- **concat**
- **findFirst, findAny**
- **flatMap**
- **limit, skip**
- **min, max**
- **peek**

Streams : forEach

- Effectue une opération sur chacun des éléments en utilisant un ***Consumer***

- Opération terminale qui consomme le **Stream**
  - On ne peut pas appeler d'autres opérations

```
List<Person> persons = Arrays.asList(
        new Person("Boris", "S", 30),
        new Person("Giovanna", "E", 20),
        new Person("Arthur", "D", 15)
);

persons.stream()
        .forEach(p -> System.out.println(p.getFirstame() + " " + p.getLastname()));
```

```
Boris S
Giovanna E
Arthur D
```

Java 8 : Les nouveautés

Streams : filter

- Accepte un ***Predicate*** pour filtrer les éléments

- C'est une opération intermédiaire, permet de chaîner d'autres opérations à sa suite

```
List<Person> persons = Arrays.asList(
        new Person("Boris", "S", 30),
        new Person("Giovanna", "E", 20),
        new Person("Arthur", "D", 15)
);

persons.stream()
        .filter(p -> p.getLastname().startsWith("S"))
        .forEach(System.out::println);
```

```
Person{firstame='Boris', lastname='S', age=30}
```

Streams : sorted

- Opération intermédiaire qui permet de trier les éléments avec un ***Comparable***

```
List<Person> persons = Arrays.asList(
        new Person("Boris", "S", 30),
        new Person("Giovanna", "E", 20),
        new Person("Arthur", "D", 15)
);

persons.stream()
        .sorted((p1, p2) -> p1.getFirstname().compareTo(p2.getFirstname()))
        .forEach(System.out::println);
```

```
Person{firstame='Arthur', lastname='D', age=15}
Person{firstame='Boris', lastname='S', age=30}
Person{firstame='Giovanna', lastname='E', age=20}
```

Streams : map

- Opération intermédiaire qui applique une ***Function*** sur les éléments

```
List<Person> persons = Arrays.asList(
        new Person("Boris", "S", 30),
        new Person("Giovanna", "E", 20),
        new Person("Arthur", "D", 15)
);

persons.stream()
        .map(Person::getAge)
        .sorted()
        .forEach(System.out::println);
```

```
15
20
30
```

Streams : allMatch, anyMatch, noneMatch

- Plusieurs méthodes permettant de vérifier que **zéro/un/des** éléments vérifient un ***Predicate***

- Toutes ces opérations sont terminales

```
boolean assert1 = persons.stream()
        .allMatch(p -> p.getFirstname().startsWith("B"));
System.out.println("assert1 = " + assert1);

boolean assert2 = persons.stream()
        .noneMatch(p -> p.getAge() == 35);
System.out.println("assert2 = " + assert2);


boolean assert3 = persons.stream()
        .anyMatch(p -> "S".equals(p.getLastname()));
System.out.println("assert3 = " + assert3);
```

```
assert1 = false
assert2 = true
assert3 = true
```

Streams : count

- Opération terminale qui retourne le nombre d'élément dans un ***Stream***

```
boolean sizeTwo = persons
        .stream()
        .filter(p -> p.getAge() >= 20)
        .count() == 2;

System.out.println("sizeTwo = " + sizeTwo);
```

```
sizeTwo = true
```

Streams : sum

- Opération terminale qui retourne la somme de tous les éléments

```
boolean sumIsFifty = IntStream.rangeClosed(1, 10).sum() == 55;
System.out.println("sumIsFifty = " + sumIsFifty);
```

```
sumIsFifty = true
```

Java 8 : Les nouveautés

Streams : reduce

- Opération terminale qui réduit les éléments du Stream avec un ***BinaryOperator***
  - On obtient au final le résultat sous la forme d'un ***Optional***
- Produit un seul résultat à partir d'une séquence d'éléments

```
List<Person> persons = Arrays.asList(
        new Person("Boris", "S", 30),
        new Person("Giovanna", "E", 20),
        new Person("Arthur", "D", 15)
);

persons
        .stream()
        .reduce((p1, p2) -> new Person(p1.getFirstname(), p1.getLastname() + " " + p2.getLastname(), p1.getAge() + p2.getAge()))
        .ifPresent(System.out::println);
```

```
Person{firstame='Boris', lastname='S E D', age=65}
```

Streams : reduce

- Opération terminale qui réduit les éléments du Stream avec un ***BinaryOperator***

- On obtient au final le résultat sous la forme d'un ***Optional***

- Il existe trois signatures de méthode pour reduce
  - **reduce(BinaryOperator<T> accumulator)**
  - **reduce(T identity, BinaryOperator<T> accumulator)**
  - **reduce(T identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<T> accumulator)**

Streams : reduce

- **reduce(BinaryOperator<T> accumulator)**

```
persons.stream()
        .reduce((p1, p2) -> new Person(p1.getFirstame(), p1.getLastname() + " " + p2.getLastname(), p1.getAge() + p2.getAge()))
        .ifPresent(System.out::println);
```

```
Person{firstame='Boris', lastname='S E D', age=65}
```

Streams : reduce

- **reduce(T identity, BinaryOperator<T> accumulator)**
  - **L'identity** ne doit pas avoir d'effet sur la fonction d'accumulation
  - **L'accumulator** définit l'opération effectué sur chacun des éléments et retient ce résultat

- https://en.wikipedia.org/wiki/Identity_element

```
int sum = persons.stream()
        .map(Person::getAge)
        .reduce(0, Integer::sum);
System.out.println(sum);
```

65

Streams : reduce

- **reduce(T identity, BiFunction<T, U, R> accumulator, BinaryOperator<T> combiner)**
  - **L'identity** ne doit pas avoir d'effet sur la fonction d'accumulation
  - **L'accumulator** définit l'opération effectuée sur chacun des éléments et retient ce résultat
  - **Le combiner** définit l'opération effectuée pour combiner deux résultats de la fonction **accumulator**

```
int sum = persons
      .parallelStream()
      .reduce(0,
            new BiFunction<Integer, Person, Integer>() {
              @Override
              public Integer apply(Integer result, Person person) {
                System.out.println("1) " + result + " + " + person.getAge());
                return result + person.getAge();
              }
            }, new BinaryOperator<Integer>() {
              @Override
              public Integer apply(Integer result1, Integer result2) {
                System.out.println("2) " + result1 + " + " + result2);
                return result1 + result2;
              }
            });
System.out.println(sum);
```

```
1) 0 + 15
1) 0 + 20
1) 0 + 30
2) 20 + 15
2) 30 + 35
65
```

Streams : concat

- Opération pour concaténer des **Stream**

```
IntStream stream1 = IntStream.range(0, 4);
IntStream stream2 = IntStream.range(4, 9);

IntStream.concat(
        stream1,
        stream2
).forEach(System.out::print);
```

Streams : findFirst, findAny

- Opérations terminales qui permettent de retourner le premier élément du **Stream**

- Le retour est un **Optional**

```
List<Person> persons = Arrays.asList(
        new Person("Boris", "S", 30),
        new Person("Giovanna", "E", 20),
        new Person("Arthur", "D", 15)
);

persons.stream()
        .findFirst()
        .ifPresent(System.out::print);

System.out.print("\n --\n");

persons.parallelStream()
        .findAny()
        .ifPresent(System.out::print);
```

```
Person{firstame='Boris', lastname='S', age=30}
 --
Person{firstame='Giovanna', lastname='E', age=20}
```

Streams : flatMap 1/2

- Opération intermédiaire qui permet de mettre à plat un **Stream**
  - Utile pour manipuler une sous collection de la collection du stream

- Exemple 1 :

```
Person p1 = new Person("Boris", new HashSet<>(asList("Lille", "Nice", "Paris")));
Person p2 = new Person("Coraline", new HashSet<>(asList("Honfleur", "Saint-Tropez")));
Person p3 = new Person("Arthur", new HashSet<>(asList("Paris", "Reims")));
List<Person> persons = asList(p1, p2, p3);

Set<String> collect =
      persons.stream()
            .map(x -> x.getCities())                  // Stream<Set<String>>
            .flatMap(x -> x.stream())                 // Stream<String>
            .filter(x -> !x.contains("Paris"))        // filter Paris city
            .collect(Collectors.toSet());             // remove duplicated

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

```
Lille
Nice
Honfleur
Saint-Tropez
Reims
```

Streams : flatMap 2/2

- Exemple 2 :
  - transformer un Stream<List<Person>> en Stream<Person> via une Function<List<Person>, Stream<Person>>

```
List<Person> persons = Arrays.asList(
        new Person("Boris", "S", 30), new Person("Giovanna", "E", 20), new Person("Arthur", "D", 15)
);
```

```
Stream<List<Person>> persons2 = Stream.<List<Person>>builder()
        .add(persons)
        .add(List.of(new Person("Jacques", "S", 29)))
        .build();

persons2.flatMap(personList -> personList.stream())
        .filter(person -> "S".equals(person.getLastname()))
        .forEach(System.out::println);
```

```
Person{firstame='Boris', lastname='S', age=30}
Person{firstame='Jacques', lastname='S', age=29}
```

Streams : limit, skip

- Permet de se déplacer ou bloquer le nombre d'élèments d'un Stream

- Ce sont des opérations intermédiaires

```
List<Person> persons = Arrays.asList(
        new Person("Boris", "S", 30),
        new Person("Giovanna", "E", 20),
        new Person("Arthur", "D", 15)
);

persons.stream()
        .limit(2)
        .skip(1)
        .findFirst()
        .ifPresent(System.out::print);
```

```
Person{firstame='Giovanna', lastname='E', age=20}
```

Streams : min, max

- Permet de se trouver le minimum ou le maximum d'un **Stream**

- Ce sont des opérations terminales qui renvoient un **Optional**

```
int min = persons
        .stream()
        .map(Person::getAge)
        .min(Comparator.naturalOrder()).orElse(0);

System.out.println(min);

int max = IntStream.rangeClosed(1, 10)
        .max()
        .getAsInt();
System.out.println(max);
```

```
15
10
```

Streams : peek

- Méthode utile pour débuguer entre les opérations effectuées sur un ***Stream***
- C'est une opération intermédiaire qui exécute un ***Consumer*** sur chacun des éléments

```
List<Person> persons = Arrays.asList(
        new Person("Boris", "S", 30),
        new Person("Giovanna", "E", 20),
        new Person("Arthur", "D", 15),
        new Person("Jacques", "S", 25)
);

persons.stream()
        .filter(p -> "S".equals(p.getLastname()))
        .peek(p -> System.out.println("Peek 1) " + p))
        .filter(p -> p.getAge() < 28)
        .peek(p -> System.out.println("Peek 2) " + p))
        .collect(Collectors.toSet());
```

```
Peek 1) Person{firstame='Boris', lastname='S', age=30}
Peek 1) Person{firstame='Jacques', lastname='S', age=25}
Peek 2) Person{firstame='Jacques', lastname='S', age=25}
```

Streams : collect

- Opération terminale pour réunir tous les éléments en utilisant un ***Collector***

```
List<Person> persons = Arrays.asList(
        new Person("Boris", "S", 30),
        new Person("Giovanna", "E", 20),
        new Person("Arthur", "D", 15)
);

List<String> firstnames = persons.stream()
        .map(Person::getFirstname)
        .collect(Collectors.toList());

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

```
Boris
Giovanna
Arthur
```

- *La classe* ***Collectors*** possède de nombreuses implémentations
  - https://docs.oracle.com/javase/9/docs/api/java/util/stream/Collectors.html

Streams : les Collectors

- *La classe* ***Collectors*** possède de nombreuses implémentations
  - https://docs.oracle.com/javase/9/docs/api/java/util/stream/Collectors.html

- ***toList()***
- ***toSet()***
- ***toCollection(TreeSet::new)***
- ***joining(***delimiter, prefix, suffix***)***
- **summingInt(**ToIntFunction**)**
- **summingInt(**ToIntFunction**)**
- ***groupingBy(****Function****)***
- ***partitioningBy(***Predicate***)***
- **…**

Streams : Collect + toList()

- Accumule les éléments dans une liste

```
List<Person> persons = Arrays.asList(
        new Person("Boris", "S", 30),
        new Person("Boris", "S", 30),
        new Person("Coraline", "R", 25),
        new Person("Arthur", "D", 15)
);

persons.stream()
        .map(Person::getFirstname)
        .collect(Collectors.toList())
        .forEach(System.out::println);
```

```
Boris
Boris
Coraline
Arthur
```

Streams : Collect + toSet()

- Accumule les éléments dans un Set

```
List<Person> persons = Arrays.asList(
        new Person("Boris", "S", 30),
        new Person("Boris", "S", 30),
        new Person("Coraline", "R", 25),
        new Person("Arthur", "D", 15)
);

persons.stream()
        .map(Person::getFirstname)
        .collect(Collectors.toSet())
        .forEach(System.out::println);
```

```
Boris
Coraline
Arthur
```

Streams : Collect + toCollection()

- Accumule les éléments dans une collection précisée
  - L'implémentation de **Comparable** peut être utilisée pour influencer le tri du TreeSet

```
List<Person> persons = Arrays.asList(
        new Person("Boris", "S", 30),
        new Person("Boris", "S", 30),
        new Person("Coraline", "R", 25),
        new Person("Arthur", "D", 15)
);

persons.stream()
        .map(Person::getFirstname)
        .collect(Collectors.toCollection(TreeSet::new))
        .forEach(System.out::println);
```

```
Boris
Coraline
Arthur
```

Streams : Collect + joining()

- Permet de concaténer les éléments avec un séparateur

```
List<Person> persons = Arrays.asList(
        new Person("Boris", "S", 30),
        new Person("Coraline", "R", 25),
        new Person("Arthur", "D", 15)
);

String result = persons.stream()
        .map(Person::getFirstname)
        .collect(Collectors.joining(", "));

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

```
Boris, Coraline, Arthur
```

Streams : Collect + summingInt()

- Effectue la somme des éléments avec un **IntFunction**

```
List<Person> persons = Arrays.asList(
        new Person("Boris", "S", 30),
        new Person("Coraline", "R", 25),
        new Person("Arthur", "D", 15)
);

String result = persons.stream()
        .map(Person::getAge)
        .collect(Collectors.summingInt(Integer::intValue));

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

```
65
```

Streams : Collect + groupingBy()

- Regroupe les éléments en fonction d'un groupe défini par une **Function**

```
List<Person> persons = Arrays.asList(
        new Person("Boris", "S", 30),
        new Person("Coraline", "R", 25),
        new Person("Arthur", "D", 15),
        new Person("Loïc", "D", 30)
);

Map<Integer, List<Person>> result =  persons
        .stream()
        .collect(Collectors.groupingBy(Person::getAge));

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

```
{
        20=[Coraline],
        30=[Boris, Loïc],
        15=[Arthur]
}
```

Streams : Collect + partitioningBy()

- Sépare les élément selon un **Predicat**

```
List<Person> persons = Arrays.asList(
        new Person("Boris", "S", 30),
        new Person("Coraline", "R", 25),
        new Person("Arthur", "D", 15),
        new Person("Loïc", "D", 30)
);

Map<Boolean, List<Person>> result =  persons
        .stream()
        .collect(Collectors.partitioningBy(p -> p.getAge() > 18));

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

```
{
        true=[Coraline , Boris, Loïc],
        false=[Arthur]
}
```

Les streams parallèles

- Les opérations effectuées sur un ***Stream*** sont séquentielles

- Il est possible de paralléliser les processus avec la méthode ***parallel***
  - On peut changer l'état d'un ***Stream*** au cours de son utilisation en utilisant ***sequential*** et ***parallel***
  - L'utilisation de ***parallel*** n'est pas systématique et son utilisation doit être pertinente

```
List<Person> persons = List.of(
        new Person("Boris", "S", 30),
        new Person("Giovanna", "E", 20),
        new Person("Arthur", "D", 15),
        new Person("Jacques", "S", 25)
);

persons.stream()
        .parallel()
        .findAny()
        .ifPresent(System.out::print);
```

```
Person{firstame='Arthur', lastname='D', age=15}
```

Les Map

- Les ***Map*** ne sont pas compatibles avec les Stream

- Elles ont quand même le droit à leur lot de nouvelles fonctionnalités :

- **putIfAbsent**
- **forEach**
- **computeIf**
- **compute**
- **computeIfAbsent**
- **computeIfPresent**
- **remove**
- **getOrDefault**
- **merge**
- **replace, replaceAll**

Les Map : putIfAbsent

- Associe une valeur avec une clef seulement si la clef n'existe pas encore ou si la valeur associée à la clef vaut ***null***

```
Map<String, String> map = new HashMap<>();

boolean a = map.putIfAbsent("key", null) == null;

boolean b = map.get("key") == null;

boolean c = map.putIfAbsent("key", "value") == null;

boolean d = map.get("key").equals("value");

boolean e = map.putIfAbsent("key", "new-value").equals("value");

boolean f = map.get("key").equals("value");
```

Les Map : forEach

- Utilise un ***BiConsumer*** pour consommer tous les couples d'une ***Map***

```
Map<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.put("two", 2);
map.put("three", 3);

map.forEach((key, value) -> System.out.printf("%s(%d) ", key, value));
```

```
one(1) two(2) three(3)
```

Les Map : computeIf

- Permet d'appliquer une ***BiFunction*** sur les valeurs de la ***map***

```
Map<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.put("two", 2);
map.put("three", 3);

map.forEach((key, value) -> System.out.printf("%s(%d) ", key, value));
System.out.println();

map.computeIfPresent("three", (key, val) -> val + 100);
System.out.println(map);
```

```
one(1) two(2) three(3)
{one=1, two=2, three=103}
```

Les Map : compute

- Si la ***Function*** retourne ***null***, le couple est supprimé de la ***map***

```
Map<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.put("two", 2);
map.put("three", 3);
System.out.println(map);

map.compute("one", (key, value) -> null);
System.out.println(map);
```

```
{one=1, two=2, three=3}
{two=2, three=3}
```

Les Map : computeIfAbsent

- Tente la création d'une valeur pour une clef si celle-ci n'est associée à aucune valeur (ou si elle est ***null***) en utilisant une ***Function***

```
Map<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.put("two", 2);
map.put("three", 3);
System.out.println(map);

map.computeIfAbsent("four", key -> 4);
System.out.println(map);

map.computeIfAbsent("five", key -> null);
System.out.println(map);
```

```
{one=1, two=2, three=3}
{four=4, one=1, two=2, three=3}
{four=4, one=1, two=2, three=3}
```

Les Map : computeIfPresent

- Tente de mettre à jour la valeur associée à une clef si la valeur en question est non-null avec une ***BiFunction***
- Si la ***Function*** retourne ***null***, le couple est supprimé de la ***Map***

```
Map<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.put("two", 2);
map.put("three", 3);
map.put("four", null);
System.out.println(map);

map.computeIfPresent("three", (key, value) -> value + 5);
System.out.println(map);

map.computeIfPresent("one", (key, value) -> null);
System.out.println(map);

map.computeIfPresent("four", (key, value) -> value + 5);
System.out.println(map);
```

```
{four=null, one=1, two=2, three=3}
{four=null, one=1, two=2, three=8}
{four=null, two=2, three=8}
{four=null, two=2, three=8}
```

Les Map : remove

- Supprime l'**Entry** pour la clef en argument si et seulement si la valeur associée à cette clef est égale à celle en argument

```
Map<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.put("two", 2);
map.put("three", 3);
System.out.println(map);

map.remove("one", 1);
System.out.println(map);

map.remove("two", 3);
System.out.println(map);
```

```
{one=1, two=2, three=3}
{two=2, three=3}
{two=2, three=3}
```

Les Map : getOrDefault

- Retourne une valeur par défaut si aucune valeur n'est associée à la clef en argument

```
Map<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.put("two", 2);
map.put("three", 3);
System.out.println(map);

int v1 = map.getOrDefault("one", -1);
System.out.println(v1);

int v6 = map.getOrDefault("six", -1);
System.out.println(v6);
```

```
{one=1, two=2, three=3}
1
-1
```

Les Map : merge

- Associe la valeur non-null en argument à une clef si cette clef n'est pas déjà associée à une valeur (ou si la valeur est ***null***), sinon applique la ***BiFunction*** sur la valeur existante

```
Map<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.put("two", 2);
map.put("three", 3);
System.out.println(map);

map.merge("example", 2, (key, value) -> null);
System.out.println(map);

map.merge("example", 2, (key, value) -> 2 + 2);
System.out.println(map);
```

```
{one=1, two=2, three=3}
{one=1, two=2, three=3, example=2}
{one=1, two=2, three=3, example=4}
```

Les Map : replace, replaceAll

- Remplace une valeur associée à une clef si cette valeur est égale à celle en argument

```
Map<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.put("two", 2);
map.put("three", 3);
System.out.println(map);

map.replace("one", 1, 5);
System.out.println(map);

map.replace("two", 3, 5);
System.out.println(map);

map.replaceAll((key, value) -> "one".equals(key) ? value + 1 : value + 2);
System.out.println(map);
```

```
{one=1, two=2, three=3}
{one=5, two=2, three=3}
{one=5, two=2, three=3}
{one=6, two=4, three=5}
```

Exercice 1/1

- Charger le projet sur votre IDE

- Effectuez les exercices concernant les Streams

Java 8 : Les nouveautés

Les Optionals

- Conteneur pour une valeur qui peut être **null**
- Contient 2 états :
  - Conteneur avec une valeur
  - Conteneur sans valeur
- Retourner un **Optional** plutôt que **null** oblige le traitement *du cas sans valeur*

```
String value = "hello";
Optional<String> hello = Optional.of(value);

System.out.println("with value :" + hello.isPresent());
System.out.println("value is : " + hello.get());
```

```
with value :true
value is : hello
```

```
Optional<Object> absent = Optional.ofNullable(null);

System.out.println("with value :" + absent.isPresent());
System.out.println("value is : " + absent.get());
// throws java.util.NoSuchElementException
}
```

```
with value :false
Exception in thread "main" java.util.NoSuchElementException: No value present
                at java.util.Optional.get(Optional.java:135)
                at com.formation.app.Main.main(Main.java:19)
```

Java 8 : Les nouveautés

Les Optionals

- La classe **Optional** dispose de nombreuses méthodes pour :
  - transformer/filtrer la valeur qu'il contient
  - Retourner une valeur par défaut
  - Lever une exception s'il ne contient pas de valeur

- https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Optional.html

static <T> Optional<T>	empty()
boolean	equals(Object obj)
Optional<T>	filter(Predicate<? super T> predicate)
<U> Optional<U>	flatMap(Function<? super T,? extends Optional<? extends U>> mapper)
T	get()
int	hashCode()
void	ifPresent(Consumer<? super T> action)
void	ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)
boolean	isEmpty()
boolean	isPresent()
<U> Optional<U>	map(Function<? super T,? extends U> mapper)
static <T> Optional<T>	of(T value)
static <T> Optional<T>	ofNullable(T value)
Optional<T>	or(Supplier<? extends Optional<? extends T>> supplier)
T	orElse(T other)
T	orElseGet(Supplier<? extends T> supplier)
T	orElseThrow()
<X extends Throwable> T	orElseThrow(Supplier<? extends X> exceptionSupplier)
Stream<T>	stream()
String	toString()

Exercice 1/1

- Charger le projet sur votre IDE

- Effectuez les exercices concernant les Optionals

Les records 1/4

- nouveau type de classe dans le langage Java
  - Introduit en Java 16

But :

- offre une syntaxe concise pour aider les développeurs à se concentrer sur la modélisation de données immuables plutôt que sur un comportement extensible
- Implémente automatiquement des méthodes orientées données telles que les accesseurs et les méthodes equals(), hashCode() et toString()

Les records 2/4

- c'est une classe final (non héritable)
- chaque élément de la description est encapsulé dans un champ private et final pour garantir l'immutabilité
- un getter public est proposé pour chaque élément
- un constructeur public qui possède la même signature que celle de la description qui initialise chaque élément avec la valeur correspondante fournie en paramètre
- une redéfinition des méthodes equals() et hashCode() qui garantit que deux instances sont égales si elles sont du même type et qu'elles contiennent les mêmes éléments
- une redéfinition de la méthode equals() qui contient le nom et la valeur de chaque élément encapsulé
-

Les records 3/4

- Exemple d'un record représentant un employé :

```
public record Employe(String nom, String prenom) {
}
```

- Nous pouvons visualiser l'équivalent du record sur slide suivante

Les records 4/4

```
public final class Employe {

   private final String nom;
   private final String prenom;

   public Employe(String nom, String prenom) {
      super();
      this.nom = nom;
      this.prenom = prenom;
   }

   public String getNom() { return nom;    }

   public String getPrenom() {       return prenom;    }

 @Override
   public int hashCode() {
      final int prime  = 31;
      int       result = 1;
      result = prime * result + ((nom == null) ? 0 : nom.hashCode());
      result = prime * result + ((prenom == null) ? 0 :
prenom.hashCode());
      return result;
   }
```

```
@Override
   public boolean equals(Object obj) {
      if (this == obj)
         return true;
      if (obj == null)
         return false;
      if (getClass() != obj.getClass())
         return false;
      Employe other = (Employe) obj;
      if (nom == null) {
         if (other.nom != null)
            return false;
      } else if (!nom.equals(other.nom))
         return false;
      if (prenom == null) {
         if (other.prenom != null)
            return false;
      } else if (!prenom.equals(other.prenom))
         return false;
      return true;
   }

   @Override
   public String toString() {
      return "Employe [nom=" + nom + ", prenom=" + prenom + "]";
   }
```

Date

- Une toute nouvelle API pour les dates est disponible dans le package **java.time.**
  - Elle s'inspire de **Joda-Time**
  - toutes les classes de cette API sont désormais immuables et threadsafe

- Il existe des différentes classes pour manipuler les dates :

- **LocalTime**
- **LocalDate**
- **LocalDateTime**
- **MonthDay**
- **ZonedDateTime**
- **ZonedId**
- **Instant**
- **Clock**
- **Period**
- **Duration**
- **ChronoUnit**
- **DayOfWeek**

Date : LocalDate

- Représente une date sans information d'heures, c'est ce qu'on utilise pour une date de naissance

```
LocalDate today = LocalDate.now();
System.out.println(today);

LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);
System.out.println(tomorrow);

LocalDate yesterday = today.minusDays(1);
System.out.println(yesterday);

LocalDate birthday = LocalDate.of(2014, Month.DECEMBER, 18);
boolean isThursday = birthday.getDayOfWeek() == DayOfWeek.THURSDAY;
System.out.println(isThursday);
```

```
DateTimeFormatter formatter = DateTimeFormatter
      .ofLocalizedDate(FormatStyle.LONG)
      .withLocale(Locale.FRANCE);
LocalDate date = LocalDate.parse("14 juillet 2014", formatter);

System.out.println(date.getDayOfMonth() == 14);
System.out.println(date.getMonth() == Month.JULY);
System.out.println(date.getYear() == 2014);
```

```
2015-06-07
2015-06-08
2015-06-06
true
2014-07-14
true
true
true
```

Java 8 : Les nouveautés

Date : LocalTime

- Classe représentant l'heure de l'horloge dans une journée
  - sans information de **TimeZone**

```
LocalTime time1 = LocalTime.of(13, 37, 26);
LocalTime time2 = LocalTime.of(15, 47, 54);

boolean isBefore = time1.isBefore(time2);
System.out.println(isBefore);

long timeBetween1 = ChronoUnit.HOURS.between(time1, time2);
System.out.println(timeBetween1);

LocalTime time3 = time1.plusMinutes(5);
long timeBetween2 = ChronoUnit.MINUTES.between(time1, time3);
System.out.println(timeBetween2);

System.out.println(time1);
String formatedDate1 =
time1.format(DateTimeFormatter.ISO_LOCAL_TIME);
System.out.println(formatedDate1);
```

```
System.out.println(time3);
String formatedDate2 =
time3.format(DateTimeFormatter.ofPattern("Ha"));
System.out.println(formatedDate2);

DateTimeFormatter formatter = DateTimeFormatter
      .ofLocalizedTime(FormatStyle.SHORT)
      .withLocale(Locale.FRANCE);
LocalTime time4 = LocalTime.parse("17:17", formatter);

System.out.println(time4.getHour());
System.out.println(time4.getMinute());
```

```
true
2
5
13:37:26
13:37:26
13:42:26
13PM
17
17
```

Date : LocalDateTime

- Représente une date avec des heures
  - sans information de **TimeZone**

```
LocalDateTime time = LocalDateTime.of(2014, Month.DECEMBER, 25, 12, 10);
System.out.println(time);
System.out.println(time.get(ChronoField.YEAR) == 2014);

String formatedTime = time.format(DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm"));
System.out.println(formatedTime);
```

```
2014-12-25T12:10
true
25/12/2014 12:10
```

- T est un séparateur entre la date et le temps

- https://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html

Date : MonthDay

- Représente une combinaison d'un mois et d'un jour du mois
  - Implémente l'interface Comparable

```
MonthDay a = MonthDay.now();
MonthDay b = MonthDay.of(Month.FEBRUARY, 29);
System.out.println(a.isAfter(b));
System.out.println(a.isBefore(b));
System.out.println(b.isValidYear(2022));
```

```
false
true
false
```

- https://docs.oracle.com/javase/8/docs/api/java/time/MonthDay.html

Date : ZonedDateTime

- Représente une date avec des heures et une **TimeZone**

```
ZonedDateTime todayEurop = ZonedDateTime.now(ZoneId.of("Europe/Paris"));
System.out.println(todayEurop.format(DateTimeFormatter.ISO_ZONED_DATE_TIME));

ZonedDateTime todayAmerica = ZonedDateTime.now(ZoneId.of("America/New_York"));
System.out.println(todayAmerica.format(DateTimeFormatter.ISO_ZONED_DATE_TIME));
```

```
2015-06-07T22:04:23.113443+02:00[Europe/Paris]
2015-06-07T16:04:23.122157-04:00[America/New_York]
```

Date : ZoneId

- Représente les **TimeZone** manipulables dans l'API

```
Set<String> availableZoneIds = ZoneId.getAvailableZoneIds();
System.out.println(availableZoneIds);
System.out.println(availableZoneIds.size());

ZoneId paris = ZoneId.of("Europe/Paris");
String fullDisplay = paris.getDisplayName(TextStyle.FULL, Locale.FRENCH);
System.out.println(fullDisplay);

String shortDisplay = paris.getDisplayName(TextStyle.SHORT, Locale.FRENCH);
System.out.println(shortDisplay);
```

```
[Asia/Aden, America/Cuiaba, Etc/GMT+9, Etc/GMT+8, Africa/Nairobi, America/Marigot, …
600
heure d'Europe centrale
CET
```

- **CET (Central European Time)** est l'heure normale d'Europe centrale (nom du fuseau UTC+1)

Date : Instant

- Représente un **instant** au cours du temps
  - Utilisable pour une valeur provenant d'une machine

```
Instant instant = Instant.parse("2007-12-03T10:15:30.00Z");
System.out.println(instant.toString());

Instant now = LocalDateTime.now().atZone(ZoneId.systemDefault()).toInstant();
System.out.println(now);

LocalDate dateNow = LocalDate.from(now);
System.out.println(dateNow.toString());
```

```
2007-12-03T10:15:30Z
2015-06-07T20:12:57.534201Z
Sun Jun 07 22:12:57 CEST 2015
```

- Z (Zulu Time) est le nom militaire pour désigner UTC (Coordinated Universal Time)
- UTC est l'heure de référence dans le monde, basé sur le Méridien de Greenwich
- GMT n'est plus utilisé depuis 1982 (Union Internationale des Télécommunications)

Date : Clock

- Permet l'accès à l'**Instant** courant

```
Instant now = Clock.system(ZoneId.of("Europe/Paris")).instant();
System.out.println(now.toString());

long millis = Clock.systemDefaultZone().millis();
System.out.println(millis);
```

```
2020-06-07T20:16:54.038430Z
1591561014048
```

Java 8 : Les nouveautés

Date : Period

- Représente une période de temps avec l'année, le mois et le jour
  - Utilisable pour une valeur lisible par l'Homme
  - Permet de trouver la période entre deux LocalDate

```
LocalDate from = LocalDate.of(2021, 5, 4);
LocalDate to = LocalDate.of(2021, 10, 10);
Period period = Period.between(from, to);
System.out.println(period.getYears() + " années");
System.out.println(period.getMonths() + " mois");
System.out.println(period.getDays() + " jours");
```

```
0 années
5 mois
6 jours
```

Date : Duration

- Représente une durée de temps, accompagnée de l'année, le mois et le jour
  - Permet de trouver la durée entre deux LocalDateTime

```
LocalDateTime from = LocalDateTime.of(2021, 10, 4, 10, 20 ,55);
LocalDateTime to = LocalDateTime.of(2021, 10, 10, 10, 21 ,01);
Duration duration = Duration.between(from, to);
System.out.println(duration.toMinutes() + " minutes");
System.out.println(duration.getSeconds() + " secondes");
```

```
8630 minutes
518406 secondes
```

Java 8 : Les nouveautés

Date : ChronoUnit

- Représente une unité utilisable pour la comparaison de des dates
  - Permet de trouver la durée entre plusieurs types de dates, ou ajouter du temps à une date

```
LocalDateTime from = LocalDateTime.of(2021, 10, 4, 10, 20 ,55);
LocalDateTime to = LocalDateTime.of(2021, 11, 10, 10, 21 ,01);

long years = ChronoUnit.YEARS.between(from, to);
System.out.println(years + " années");
long months = ChronoUnit.MONTHS.between(from, to);
System.out.println(months + " mois");
long days = ChronoUnit.WEEKS.between(from, to);
System.out.println(days + " jours");
long milliseconds = ChronoUnit.MILLIS.between(from, to);
System.out.println(milliseconds + " millisecondes");
long nano = ChronoUnit.NANOS.between(from, to);
System.out.println(nano + " nano");
```

```
0 années
1 mois
5 semaines
37 jours
...
3196806000 millisecondes
3196806000000000 nano
```

Date : Month

- Représente une énumération des 12 mois de l'année
  - Chaque jour de la semaine contient une valeur entière

```
LocalDate localDate = LocalDate.of(2017, Month.JANUARY, 25);
DayOfWeek dayOfWeek = DayOfWeek.from(localDate);
System.out.println(dayOfWeek.get(ChronoField.DAY_OF_WEEK));
```

```
3
```

Date : DayOfWeek

- Représente une énumération des 7 jours de la semaine
  - Chaque jour de la semaine contient une valeur entière

```
LocalDate localDate = LocalDate.of(2017, Month.JANUARY, 25);
DayOfWeek dayOfWeek = DayOfWeek.from(localDate);
System.out.println(dayOfWeek.get(ChronoField.DAY_OF_WEEK));
```

```
3
```

Références

- Présentation Pdf
  - https://aseigneurin.github.io/downloads/pres-java8-bdxio/index.html#30
- Conférence DEVOXX France (par José Paumard)
  - https://www.youtube.com/watch?v=lRDskUlCGIg
- Article d'un blog
  - https://blog.axopen.com/2014/05/java-8-api-stream-introduction-collections/#lrsquoutilisation-de-map-sur-les-stream
- Article de Jean-Michel Doudoux
  - https://www.jmdoudoux.fr/java/dej/chap-lambdas.htm
- Article de Jean Christophe Gay
  - https://jeanchristophegay.com/posts/java8-lambda-stream/

Java 8 : Les nouveautés

Exercice 1/1

- Charger le projet sur votre IDE

- Effectuez les exercices concernant les Dates

- FIN -