Formation Java 8

TP n°3 : Les interfaces fonctionnelles

SOMMAIRE

I- Prérequis :

- JDK 8.
- Eclipse.

II- Objectifs

✓ Comprendre l'annotation @FunctionnalInterface.
✓ Définir une interface fonctionnelle.
✓ Comment utiliser une interface fonctionnelle.
✓ Les interfaces fonctionnelles du package java.util.function.

III- Les interfaces fonctionnelles

❖ Une interface fonctionnelle (functional interface) est une interface dans laquelle **une seule méthode abstraite est définie**. Elle doit respecter certaines contraintes :

- elle ne doit avoir qu'une seule méthode déclarée abstraite
- les méthodes définies dans la classe Object ne sont pas prises en compte comme étant des méthodes abstraites
- toutes les méthodes doivent être public
- elle peut avoir des méthodes par défaut et static

Avant Java 8 un certain nombre d'interfaces ne définissaient qu'une seule méthode : ces interfaces sont désignées par le terme Single Abstract Method (SAM), par exemple :

- Comparator<T> qui définit la méthode int compare(T o1, T o2)
- Callable<V> qui définit la méthode V call() throws exception
- Runnable qui définit la méthode void run()
- ActionListener qui définit la méthode void actionPerformed(ActionEvent)
- ...

Elles se prêtent bien à l'utilisation d'une classe anonyme interne. Si elles ne sont pas modifiées alors elles sont utilisables comme interfaces fonctionnelles même si elles ne sont pas toutes annotées avec @FunctionalInterface puisque ce type de méthode respecte le contrat des interfaces fonctionnelles. Il est donc possible d'utiliser une expression lambda à la place d'une classe anonyme. Exemple :

```
package ma.formation.interfacesfonctionnelles;

import java.util.function.BiConsumer;
import java.util.function.BiFunction;
import java.util.function.Consumer;

public class Test1 {

public static void main(String[] args) {
```

```
Consumer<String> afficher = (message) -> { System.out.println(message); };
BiConsumer<Integer, Integer> additionner = (x, y) -> System.out.println(x + y);
BiFunction<Integer, Integer, Long> additionner2 = (x, y) -> (long) x + y;

afficher.accept("Bonjour");
additionner.accept(3, 4);
System.out.println(additionner2.apply(33, 44));
 }
}
```

a. L'annotation @FunctionnalInterface

- Les interfaces fonctionnelles peuvent être annotées avec @FunctionalInterface : cette annotation permet de préciser l'intention que l'interface soit fonctionnelle.
  L'utilisation de cette annotation est optionnelle mais elle apporte deux avantages :
  - Elle indique au compilateur que l'interface est fonctionnelle : celui-ci va pouvoir vérifier que toutes les règles soient respectées pour qu'elle soit effectivement fonctionnelle
  - L'outil javadoc va utiliser l'annotation lors de la génération de la documentation.

  L'annotation @FunctionalInterface ne peut être utilisée que sur une interface.

```
package ma.formation.interfacesfonctionnelles;

public class Test2 {

	public static void main(String[] args) {
MonInterface traitement = () -> System.out.println("Exemple d'impélmentation d'une interface fonctionnelle");
		traitement.traiter();
	}
}
@FunctionalInterface
interface MonInterface {
	public void traiter();

}
```

```
<terminated> Test2 (1) [Java Application] C:\Java\jdk1.8.0_121\bin\javaw.exe (21 déc. 2021 à 14:28:41)
Exemple d'impélmentation d'une interface fonctionnelle
```

- L'annotation ***@FunctionalInterface*** permet d'indiquer explicitement au compilateur que l'interface est une interface fonctionnelle : celui-ci pourra alors vérifier que les contraintes sont respectées. Son utilisation est facultative car le compilateur peut déterminer automatiquement si une interface correspond aux contraintes des interfaces fonctionnelles.

- ❖ Si l'annotation est utilisée sur une classe, une énumération, une annotation ou sur une interface qui ne respecte pas les contraintes d'une interface fonctionnelle alors le compilateur émettra une erreur.

b. La définition d'une interface fonctionnelle

- ❖ Une interface peut contenir la définition de différents types de méthodes :
  - Méthode abstraite.
  - redéclarer une méthode de la classe Object pour par exemple utiliser un commentaire Javadoc particulier.
  - Méthode par défaut à partir de Java 8.

Pour être une interface fonctionnelle, une interface ne doit avoir qu'une seule méthode abstraite déclarée. Elle peut avoir aucune, une ou plusieurs redéfinitions de méthodes de la classe Object ou des méthodes par défaut.
Une interface fonctionnelle ne peut pas avoir plus d'une méthode abstraite.
L'exemple ci-dessous est une interface fonctionnelle correcte :

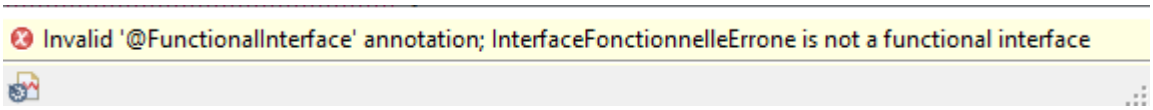
```
package ma.formation.interfacesfonctionnelles;

public class Test3 {
	public static void main(String[] args) {
		MonInterfaceFonctionnelle traitement=()->"il s'agit bien d'une
interface fonctionnelle";
		System.out.println(traitement.executer());
	}
}

@FunctionalInterface
 interface MonInterfaceFonctionnelle {
  String executer();
  static void init() {}
  default void init2() {}
  default void init3() {}

}
```

```
<terminated> Test3 (2) [Java Application] C:\Java\jdk1.8.0_121\bin\javaw.exe (21 déc. 2021 à 14:39:18)
il s'agit bien d'une interface fonctionnelle
```

Par contre, l'exemple ci-dessous ne se compile pas :

```
@FunctionalInterface
 public interface InterfaceFonctionnelleErrone {
  String executer();
  String effacer();
  static void init() {}
  default void init2() {}
  default void init3() {}
```

```
}
```



- Les méthodes publiques de la classe Object peuvent cependant être redéfinies dans une interface fonctionnelle.

```
@FunctionalInterface
 public interface ExampleInterfaceFonctionnelle {
        String executer();
        boolean equals(Object obj);
}
```

- Ci-dessous, bien que la méthode clone() soit déclarée dans la classe Object, l'interface fonctionnelle ne compile pas car la méthode clone() de la classe Object est déclarée protected et non public.

```
package ma.formation.interfacesfonctionnelles;

public class Test6 {
      public static void main(String[] args) {

      }
}

@FunctionalInterface
  interface InterfaceFonctionnelleErrone2 {
        String executer();
        boolean equals(Object obj);
        Object clone();
}
```

c. L'utilisation d'une interface fonctionnelle

- Une interface fonctionnelle définit une méthode qui pourra être utilisée pour passer en paramètre :
  - Une référence sur une méthode d'une instance
  - Une référence sur une méthode statique
  - Une référence sur un constructeur
  - Une expression lambda
  - Une classe anonyme interne
- Une interface fonctionnelle définit un type qui peut être utilisé dans plusieurs situations :

- Assignation :

```
package ma.formation.interfacesfonctionnelles;

import java.util.function.Predicate;

public class Test7 {
	public static void main(String[] args) {
		Predicate<String> estVide = String::isEmpty;
		System.out.println(estVide.test("kkk"));
	}
}
```

- Directement en paramètre d'une méthode :

```
package ma.formation.interfacesfonctionnelles;

import java.util.stream.Stream;

public class Test8 {
public static void main(String[] args) {
	Stream<String> liste=Stream.of("Maroc","Egypt","France","USA");
	liste.filter(e -> e.length() >=4).forEach(a -> System.out.println(a));
	}
}
```

- Une interface fonctionnelle est avant tout une interface : elle peut donc être utilisée comme telle dans le code.

```
package ma.formation.interfacesfonctionnelles;

public class Test9 {

	@FunctionalInterface
	public interface ExempleIF {
	  void executer();
	}

	public static void executer(ExempleIF monInterface) {
	    monInterface.executer();
	  }

	  public static void main(String[] args) {
	    executer(new ExempleIF() {

	      @Override
	      public void executer() {
	        System.out.println("test");
	      }
	    });
	  }
	}
```

Il est aussi possible de profiter de la simplicité de la syntaxe d'une expression lambda puisqu'une expression lambda peut être utilisée partout où un objet de type interface fonctionnelle est attendu.

```
package ma.formation.interfacesfonctionnelles;

public class Test10 {

	@FunctionalInterface
	public interface ExempleIF {
	  void executer();
	}

	public static void executer(ExempleIF monInterface) {
	    monInterface.executer();
	  }

	  public static void main(String[] args) {
	    executer(()->System.out.println("test"));

	}
}
```

- L'exemple ci-dessous tri un tableau de chaînes de caractères en utilisant la méthode sort() de la classe Arrays. Elle attend en paramètre le tableau à trier et une instance de type Comparator. Comme l'interface Comparator est une interface fonctionnelle, il est possible d'utiliser une expression lambda.

```
package ma.formation.interfacesfonctionnelles;

import java.util.Arrays;

public class Test11 {

	public static void main(String[] args) {

	    String[] elements = new String[] { "aaa","zzz","fff","mmm"};
	    Arrays.sort(elements, (o1, o2) -> o1.compareTo(o2));
	    System.out.println(Arrays.toString(elements));
	  }
}
```

- C'est le compilateur qui se charge d'effectuer toutes les opérations requises.
- Il n'est pas possible d'assigner une expression lambda à un objet de type Object.

```
package ma.formation.interfacesfonctionnelles;

import java.util.Arrays;
```

```
public class Test12 {

	public static void main(String[] args) {

		Object tri = (o1, o2) -> o1.compareTo(o2);
	  }
}
```

- Cette assignation n'est pas légale car le compilateur ne peut pas déterminer la méthode qui devra être invoquée puisque Object n'est pas une interface fonctionnelle.
- Par contre, il est possible d'assigner à un objet d'un type d'une interface fonctionnelle une expression lambda qui respecte la déclaration de la méthode abstraite.

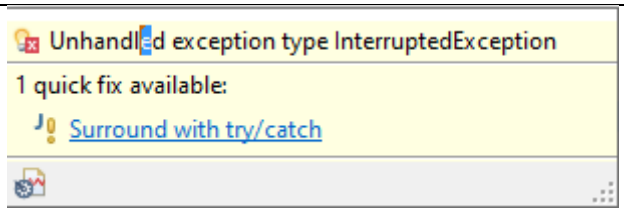
```
package ma.formation.interfacesfonctionnelles;

import java.util.Arrays;
import java.util.Comparator;

public class Test13 {

	public static void main(String[] args) {

		Comparator<String> tri = (o1, o2) -> o1.compareTo(o2);
	    String[] elements = new String[] { "aaa","zzz","fff","mmm"};
	    Arrays.sort(elements, tri);
	    System.out.println(Arrays.toString(elements));
	  }
}
```

- Il est aussi impératif de préciser le type generic de l'interface fonctionnelle pour permettre au compilateur d'inférer le type des paramètres de l'expression lambda et de vérifier que ce type possède bien une méthode compareTo().

```
package ma.formation.interfacesfonctionnelles;

import java.util.Comparator;

public class Test14 {

	public static void main(String[] args) {
		Comparator tri = (o1, o2) -> o1.compareTo(o2);
	  }
}
```

- Si une exception de type checked est levée dans le corps de l'expression lambda sans être gérée alors il est nécessaire que la méthode de l'interface fonctionnelle correspondante déclare la levée de cette exception sinon une erreur est émise à la compilation.

```java
package ma.formation.interfacesfonctionnelles;

public class Test15 {

	public static void main(String[] args) {
		Runnable monTraitement = () -> {
			System.out.println("debut");
			Thread.sleep(1000);
			System.out.println("fin");
		};
	}
}
```



- La méthode run() de l'interface Runnable ne déclare pas pouvoir lever une exception.
  Pour résoudre ce problème, il y a plusieurs solutions :

  - ✓ ne pas lever d'exception
  - ✓ gérer l'exception dans le corps de l'expression lambda en utilisant un bloc try/catch
  - ✓ déclarer que la méthode de l'interface fonctionnelle peut lever une exception (impossible dans le cas de l'interface Runnable)
  - ✓ utiliser une autre interface fonctionnelle : dans le cas ci-dessous, Callable.

```java
package ma.formation.interfacesfonctionnelles;

import java.util.concurrent.Callable;

public class Test16 {

	public static void main(String[] args) {
		Callable<Object> monTraitement = () -> {
			System.out.println("debut");
			Thread.sleep(1000);
			System.out.println("fin");
			return null;
		};

		try {
			monTraitement.call();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

d. Les interfaces fonctionnelles du package java.util.function

❖ Le package java.util.function propose en standard des interfaces fonctionnelles d'usage courant. Toutes les interfaces de ce package sont annotées avec @FunctionalInterface.
Le nom des interfaces du package java.util.function respecte une convention de nommage selon leur rôle :

- ✓ **Function** : une fonction unaire qui permet de réaliser une transformation. Elle attend un ou plusieurs paramètres et renvoie une valeur. La méthode se nomme apply()
- ✓ **Consumer** : une fonction qui permet de réaliser une action. Elle ne renvoie pas de valeur et attend un ou plusieurs paramètres. La méthode se nomme accept()
- ✓ **Predicate** : une fonction qui attend un ou plusieurs paramètres et renvoie un booléen. La méthode se nomme test()
- ✓ **Supplier** : une fonction qui renvoie une instance. Elle n'attend pas de paramètre et renvoie une valeur. La méthode se nomme get(). Elle peut être utilisé comme une fabrique

Le nom des interfaces fonctionnelles qui attendent en paramètre une ou plusieurs valeurs primitives sont préfixées par le type primitif.
Le nom des interfaces fonctionnelles qui renvoient une valeur primitive sont suffixées par toXXX.

❖ **Les interfaces de type Function :**

Interface fonctionnelle	Description
BiFunction<T,U,R>	Représente une opération qui requiert deux objets de type T et U et renvoie un résultat de type R
DoubleFunction<R>	Représente une fonction qui attend en paramètre une valeur de type double et renvoie un résultat de type R
DoubleToIntFunction	Représente une opération qui attend en paramètre un double et renvoie un int comme résultat
DoubleToLongFunction	Représente une opération qui attend en paramètre un double et renvoie un long comme résultat
Function<T,R>	Représente une fonction qui attend un paramètre de type T et renvoie un résultat de type R
IntFunction<R>	Représente une fonction qui attend en paramètre une valeur de type int et renvoie un résultat de type R
IntToDoubleFunction	Représente une fonction qui attend en paramètre une valeur de type int et renvoie un double
IntToLongFunction	Représente une fonction qui attend en paramètre une valeur de type int et renvoie un long
LongFunction<R>	Représente une fonction qui attend en paramètre une valeur de type long et renvoie un résultat de type R

LongToDoubleFunction	Représente une fonction qui attend en paramètre une valeur de type long et renvoie un double
LongToIntFunction	Représente une fonction qui attend en paramètre une valeur de type long et renvoie un int
ToDoubleBiFunction<T,U>	Représente une fonction qui attend deux paramètres de type T et U et renvoie un double
ToDoubleFunction<T>	Représente une fonction qui attend un paramètre de type T et renvoie un double
ToIntBiFunction<T,U>	Représente une fonction qui attend deux paramètres de type T et U et renvoie un int
ToIntFunction<T>	Représente une fonction qui attend un paramètre de type T et renvoie un int
ToLongBiFunction<T,U>	Représente une fonction qui attend deux paramètres de type T et U et renvoie un long
ToLongFunction<T>	Représente une fonction qui attend un paramètre de type T et renvoie un long
BinaryOperator<T>	Représente une opération qui attend deux paramètres de type T et renvoie une instance de type T
DoubleBinaryOperator	Représente une opération qui attend en paramètre deux valeurs de type double et renvoie une valeur de type double
DoubleUnaryOperator	Représente une opération qui attend en paramètre un double et renvoie un double comme résultat
IntBinaryOperator	Représente une opération qui attend en paramètres deux valeurs de type int et renvoie une valeur de type int
IntUnaryOperator	Représente une opération qui attend en paramètre un int et renvoie un int comme résultat
LongBinaryOperator	Représente une opération qui attend en paramètres deux valeurs de type long et renvoie une valeur de type long
LongUnaryOperator	Représente une opération qui attend en paramètre un objet de type long et renvoie une valeur de type long
UnaryOperator<T>	Représente une opération qui attend en paramètre un objet de type T et renvoie une instance de type T. Elle hérite de Function<T, T>

- Les interfaces de type Consumer :

Interface fonctionnelle	**Description**
BiConsumer<T,U>	Représente une opération qui requiert deux objets et ne renvoie aucun résultat

Consumer<T>	Représente un consommateur d'un unique paramètre qui ne renvoie aucune valeur
DoubleConsumer	Représente un consommateur d'une valeur de type double
IntConsumer	Représente un consommateur d'une valeur de type int
LongConsumer	Représente un consommateur d'une valeur de type long
ObjDoubleConsumer<T>	Représente un consommateur qui attend en paramètres un objet de type T et un double
ObjIntConsumer<T>	Représente un consommateur qui attend en paramètres un objet de type T et un int
ObjLongConsumer<T>	Représente un consommateur qui attend en paramètres un objet de type T et un long

❖ Les interfaces de type Predicate :

Interface fonctionnelle	**Description**
BiPredicate<T,U>	Représente un prédicat qui attend deux paramètres et renvoie un booléen
DoublePredicate	Représente un prédicat qui attend en paramètre un argument de type double et renvoie un booléen
IntPredicate	Représente un prédicat qui attend en paramètre un argument de type int et renvoie un booléen
LongPredicate	Représente un prédicat qui attend en paramètre un argument de type long et renvoie un booléen
Predicate<T>	Représente un prédicat qui attend en paramètre un argument de type T et renvoie un booléen

❖ Les interfaces de type Supplier :

Interface fonctionnelle	**Description**
BooleanSupplier	Représente un fournisseur d'une valeur booléenne qui n'attend aucun paramètre
DoubleSupplier	Représente un fournisseur d'une valeur de type double
IntSupplier	Représente un fournisseur de valeur qui n'attend aucun paramètre et renvoie un entier de type int
LongSupplier	Représente un fournisseur de valeur qui n'attend aucun paramètre et renvoie un entier de type long
Supplier<T>	Représente un fournisseur de valeur qui n'attend aucun paramètre et renvoie une instance de type T

Ces interfaces fonctionnelles couvrent de nombreux besoins courants mais il est aussi possible de définir ses propres interfaces fonctionnelles.

d.1. Les interfaces fonctionnelles de types Consumer

❖ L'interface fonctionnelle Consumer<T> définit une fonction qui effectue une opération sur un objet et ne renvoie aucune valeur. Son exécution engendre généralement des effets de bord.
Elle définit la méthode fonctionnelle accept(T t) qui ne renvoie aucune valeur.
Elle définit aussi une méthode par défaut :

Méthode	Rôle
default Consumer andThen(Consumer< ? super T>	Renvoyer un Consumer qui exécute en séquence l'instance courante et celle fournie en paramètre

```
package ma.formation.interfacesfonctionnelles;

import java.util.function.Consumer;

public class Test17 {

	public static void main(String[] args) {
	    Consumer<String> c = System.out::print;
	    c.andThen(c).accept("bonjour ");
	  }
}
```

❖ Lors de l'invocation de la méthode andThen(), si une exception est levée par un des deux Consumer, celle-ci est propagée dans la méthode englobante.

```
package ma.formation.interfacesfonctionnelles;

import java.util.concurrent.atomic.AtomicInteger;
import java.util.function.Consumer;

public class Test18 {

	public static void main(String[] args) {
	    AtomicInteger i = new AtomicInteger(0);
	    Consumer<String> c = (x) -> {
	      i.addAndGet(1);
	      System.out.println(x);
	      if (i.get() == 1) {
	        throw new RuntimeException();
	      }
```

```
            };
            c.andThen(c).accept("bonjour");
        }
}
```

```
bonjour
Exception in thread "main" java.lang.RuntimeException
        at ma.formation.interfacesfonctionnelles.Test18.lambda$0(Test18.java:14)
        at java.util.function.Consumer.lambda$andThen$0(Consumer.java:65)
        at ma.formation.interfacesfonctionnelles.Test18.main(Test18.java:17)
```

- Si une exception est levée lors de l'exécution du Consumer courant, le second Consumer n'est pas exécuté.

```
package ma.formation.interfacesfonctionnelles;

import java.util.function.Consumer;

public class Test19 {

	public static void main(String[] args) {
	    Consumer<String> c = (x) -> {
	      System.out.println(x);
	      throw new RuntimeException();
	    };
	    c.andThen(c).accept("bonjour");
	  }
}
```

```
bonjour
Exception in thread "main" java.lang.RuntimeException
        at ma.formation.interfacesfonctionnelles.Test19.lambda$0(Test19.java:10)
        at java.util.function.Consumer.lambda$andThen$0(Consumer.java:65)
        at ma.formation.interfacesfonctionnelles.Test19.main(Test19.java:12)
```

- L'interface fonctionnelle @BiConsumer :

```
package ma.formation.interfacesfonctionnelles;

import java.util.function.BiConsumer;

public class Test20 {

	public static void main(String[] args) {

	  math(1, 1, (x, y) -> System.out.println(x + y));    // 2
	  math(1, 1, (x, y) -> System.out.println(x - y));    // 0
	  math(1, 1, (x, y) -> System.out.println(x * y));    // 1
	  math(1, 1, (x, y) -> System.out.println(x / y));    // 1
```

```
    }

    static <Integer> void math(Integer a1, Integer a2, BiConsumer<Integer, Integer> c) {
        c.accept(a1, a2);
    }

}
```

d.2. Les interfaces fonctionnelles de types Function

❖ L'interface fonctionnelle Function<T, R> définit une fonction qui effectue une opération sur un objet de type T et renvoie une valeur de type R.
Elle définit la méthode fonctionnelle apply(T t) qui renvoie une valeur de type R.
Elle définit aussi plusieurs méthodes par défaut et static :

Méthode	Rôle
default <V> Function<T, V> andThen(Function< ? super R, ? extends V >	Renvoyer une Function qui exécute en séquence l'instance courante et celle fournie en paramètre
default <V> Function <V, R> compose(Function< ? super V, ? extends T>	Renvoyer une Function qui exécute l'instance fournie en paramètre et applique l'instance courante sur le résultat
static <T> Function<T, T> identity()	Renvoyer une Function qui renvoie toujours la valeur fournie en paramètre

```
package ma.formation.interfacesfonctionnelles;

import java.util.function.Function;

public class Test21 {

        public static void main(String[] args) {
           Function<Integer,Long> doubler = (i) -> (long) i * 2;
           System.out.println(doubler.apply(2));
         }
}
```

Le résultat est 4.

❖ Les méthodes andThen() et compose() permettent de mixer deux Function.

```
package ma.formation.interfacesfonctionnelles;

import java.util.function.Function;

public class Test22 {
```

```
        public static void main(String[] args) {
            Function<Long, Long> doubler = (i) -> {
              long resultat = (long) i * 2;
              System.out.println("doubler=" + resultat);
              return resultat;
            };

            Function<Long, Long> laMoitie = (i) -> {
              long resultat = i / 2;
              System.out.println("laMoitie=" + resultat);
              return resultat;
            };

            System.out.println(doubler.andThen(laMoitie).apply(10L));
           System.out.println(doubler.compose(laMoitie).apply(10L));
          }
}
```

```
<terminated> Test22 [Java Application] C:\Java\jdk1.8.0_121\bin\javaw.exe (22 déc. 2021 à 18:39:30)
doubler=20
laMoitie=10
10
laMoitie=5
doubler=10
10
```

- Un autre exemple :

```
package ma.formation.interfacesfonctionnelles;

import java.util.function.Function;

public class Test23 {

      public static void main(String[] args) {
            Function<String,String> upperCase=String::toUpperCase;
            Function<String,String> trim=String::trim;
            Function<String,String> concat=(s)->s+" every body";
            String string=" Hi ";
            Function<String, String> upperCaseThenTrim =
upperCase.andThen(trim).andThen(concat);
            System.out.println(upperCaseThenTrim.apply(string));
        }
}
```

Le résultat :

```
HI every body
```

- Un autre exemple :

```
package ma.formation.interfacesfonctionnelles;

import java.util.function.Function;
```

```
public class Test24 {

	public static void main(String[] args) {
		Function<String,String> upperCase=String::toUpperCase;
		Function<String,String> trim=String::trim;
		Function<String,String> concat=(s)->s+" every body";
		String string=" Hi ";
		Function<String, String> trimBeforeUpperCase = upperCase.compose(trim).andThen(concat);
		System.out.println(trimBeforeUpperCase.apply(string));
	 }
}
```

Le résultat est le même. La différence réside dans le fait que la fonction trim est exécutée avant la fonction upperCase, alors que dans l'exemple précédent, le fonction trim est exécutée après la fonction upperCase.

- L'interface fonctionnelle BiFunction définit une opération qui attend deux paramètres et renvoie une valeur. C'est une spécialisation de l'interface fonctionnelle Function qui attend deux paramètres.
  L'interface BiFunction<T,U,R> est typée avec trois generics qui précisent respectivement les types des deux arguments de l'opération de la fonction et le type de la valeur de retour.
  Elle définit la méthode fonctionnelle accept(T t, U u) et renvoie une valeur de type R.
  Elle définit aussi une méthode par défaut :

Méthode	Rôle
default <V> BiFunction<T, U, V> andThen(Function< ? super R, ? extends V>	Renvoyer une BiFunction qui exécute en séquence l'instance courante et celle fournie en paramètre

```
package ma.formation.interfacesfonctionnelles;

import java.util.function.BiFunction;

public class Test25 {

	public static void main(String[] args) {
	    BiFunction<String, String, String> concatener = (x, y) -> x + y;
	    System.out.println(concatener.apply("Bonjour", " Java"));
	  }

}
```

Le résultat est :

```
Bonjour Java
```

❖ Un autre exemple :

```
package ma.formation.interfacesfonctionnelles;

import java.util.function.BiFunction;
import java.util.function.Function;

public class Test26 {

	public static void main(String[] args) {

	  String result = powToString(2, 4, (a1, a2) -> Math.pow(a1, a2),
	          (r) -> "Result : " + String.valueOf(r));

	  System.out.println(result); // Result : 16.0

    }

    public static <R> R powToString(Integer a1, Integer a2,
                                    BiFunction<Integer, Integer, Double> func,
                                    Function<Double, R> func2) {
        return func.andThen(func2).apply(a1, a2);

    }

}
```

d.3. Les interfaces fonctionnelles de types Predicate

❖ Les interfaces fonctionnelles de type Predicate (Predicate, BiPredicate, DoublePredicate, IntPredicate, LongPredicate) définissent des fonctions qui attendent différents types de paramètres et renvoient une valeur booléenne.
L'interface fonctionnelle Predicate<T> définit une fonction qui effectue une opération sur un objet et renvoie une valeur booléenne.
Elle définit la méthode fonctionnelle test(T t) qui renvoie un booléen.
Elle définit aussi plusieurs méthodes par défaut ou static :

Méthode	Rôle
default Predicate<T> and(Predicate< ? super T>)	Renvoyer un Predicate qui exécute en séquence l'instance courante et celle fournie en paramètre en effectuant un ET logique sur leurs résultats. Si le prédicat courant est false alors celui fourni en

	paramètre n'est pas évalué. Une exception levée par l'un des deux est propagée à l'appelant. Si une exception est levée lors de l'exécution du premier prédicat, le second n'est pas évalué.
static <T> Predicate<T> isEqual(Object	Renvoyer un Predicate qui teste l'égalité de l'objet fourni en paramètre et de celui passé en paramètre de la méthode test() en utilisant la méthode Objects.equals()
default Predicate<t> negate()	Renvoyer un Predicate qui exécute l'instance courante en effectuant un NOT logique sur son résultat
default Predicate<T> or(Predicate< ? super T>)	Renvoyer un Precidate qui exécute en séquence l'instance courante et celle fournie en paramètre en effectuant un OU logique sur leurs résultats. Si le prédicat courant est true alors celui fourni en paramètre n'est pas évalué. Une exception levée par l'un des deux est propagée à l'appelant. Si une exception est levée lors de l'exécution du premier prédicat, le second n'est pas évalué.

```
package ma.formation.interfacesfonctionnelles;

import java.util.Objects;
import java.util.function.Predicate;

public class Test27 {

	public static void main(String[] args) {
		Predicate<String> possedeTailleTrois = s -> s.length() == 3;
		Predicate<String> contientX = s -> s.contains("X");
		Predicate<String> estNonNull = Objects::nonNull;
		Predicate<String> contientXOuTaille3 = contientX.or(possedeTailleTrois);
		Predicate<String> estSMS = Predicate.isEqual("SMS");

		System.out.println("1 " + contientX.negate().test("WXYZ"));
		System.out.println("2 " + contientX.or(possedeTailleTrois).test("WWW"));
		System.out.println("3 " + contientX.or(possedeTailleTrois).test("WX"));
		System.out.println("4 " + contientX.and(possedeTailleTrois).test("WXY"));
		System.out.println("5 " + contientX.and(possedeTailleTrois).test("WWW"));
		System.out.println("6 " + estNonNull.test(null));
		System.out.println("7 " +
estNonNull.and(contientX).and(possedeTailleTrois).test("WWW"));
		System.out.println("8 " +
estNonNull.and(contientX).and(possedeTailleTrois).test("XX"));
		System.out.println("9 " +
estNonNull.and(contientX).and(possedeTailleTrois).test(null));
		System.out.println("10 " + estNonNull.and(contientXOuTaille3).test("WWW"));
		System.out.println("11 " + estNonNull.and(contientXOuTaille3).test("XX"));
		System.out.println("12 " + estNonNull.and(contientXOuTaille3).test(null));
		System.out.println("13 " +
estNonNull.and(contientX.or(possedeTailleTrois)).test("WWW"));
		System.out.println("14 " +
estNonNull.and(contientX.or(possedeTailleTrois)).test("XX"));
		System.out.println("15 " +
estNonNull.and(contientX.or(possedeTailleTrois)).test(null));
		System.out.println("16 " + estSMS.test("SMS"));
```

```
            System.out.println("17 " + estSMS.test("ABC"));
            System.out.println("18 " + estSMS.test(null));
      }
}
```

```
1 false
2 true
3 true
4 true
5 false
6 false
7 false
8 false
9 false
10 true
11 true
12 false
13 true
14 true
15 false
16 true
17 false
18 false
```

❖ L'interface fonctionnelle BiPredicate définit une opération qui attend deux paramètres et renvoie une valeur booléenne. C'est une spécialisation de l'interface fonctionnelle Predicate qui attend deux paramètres.
L'interface BiPredicate<T,U> est typé avec deux generic qui précisent respectivement le type du premier et du second argument de l'opération de la fonction.
Elle définit la méthode fonctionnelle test(T t, U u) qui renvoie un booléen.
Elle définit aussi plusieurs méthodes par défaut :

Méthode	Rôle
default BiPredicate<T, U> and(BiPredicate< ? super T, ? super U>)	Renvoyer un BiPredicate qui exécute en séquence l'instance courante et celle fournie en paramètre en effectuant un ET logique sur leurs résultats. Si le prédicat courant est false alors celui fourni en paramètre n'est pas évalué. Une exception levée par l'un des deux est propagée à l'appelant. Si une exception est levée lors de l'exécution du premier prédicat, le second n'est pas évalué.
default BiPredicate<T, U> negate()	Renvoyer un BiPredicate qui exécute l'instance courante en effectuant un NOT logique sur son résultat
default BiPredicate<T, U> or(BiPredicate< ? super T, ? super U>)	Renvoyer un BiPrecidate qui exécute en séquence l'instance courante et celle fournie en paramètre en effectuant un OU logique sur leurs résultats. Si le prédicat courant est true alors celui fourni en paramètre n'est pas évalué. Une exception levée par l'un des deux est propagée à l'appelant. Si une exception est levée lors de

	l'exécution du premier prédicat, le second n'est pas évalué.

```
package ma.formation.interfacesfonctionnelles;

import java.util.function.BiPredicate;

public class Test28 {

	public static void main(String[] args) {
		BiPredicate<Integer, Integer> estSupOuEgal = (x, y) -> x >= y;
		BiPredicate<Integer, Integer> estLaMoitie = (x, y) -> x == y * 2;

		System.out.println("1 " + estSupOuEgal.test(2, 3));
		System.out.println("1 " + estSupOuEgal.test(3, 2));

		System.out.println("2 " + estSupOuEgal.and(estLaMoitie).test(4, 2));
		System.out.println("2 " + estSupOuEgal.and(estLaMoitie).test(3, 2));

		System.out.println("3 " + estSupOuEgal.negate().test(3, 2));

		System.out.println("4 " + estSupOuEgal.or(estLaMoitie).test(1, 1));
		System.out.println("4 " + estSupOuEgal.or(estLaMoitie).test(4, 2));
		System.out.println("4 " + estSupOuEgal.or(estLaMoitie).test(2, 4));
	}
}
```

```
1 false
1 true
2 true
2 false
3 false
4 true
4 true
4 false
```

d.4. Les interfaces fonctionnelles de types Supplier

- L'interface fonctionnelle Supplier définit une fonction qui renvoie une valeur dont le type correspond au type générique.
  Elle définit la méthode fonctionnelle get() qui renvoie un objet de type T.

```
package ma.formation.interfacesfonctionnelles;

import java.util.function.Supplier;

public class Test29 {

	public static void main(String[] args) {
		Supplier<String> message = ()->"Bonjour";
```

```
            System.out.println(message.get());
        }
}
```

```
package ma.formation.interfacesfonctionnelles;

import java.util.function.Supplier;

public class Test30 {

        public static void main(String[] args) {
                Supplier<ClassA> objet = ClassA::new;
            System.out.println(objet.get());
        }
}

class ClassA {
}
```

- L'interface Supplier ne permet de renvoyer que des objets. Plusieurs interfaces fonctionnelles la spécialisent pour retourner des valeurs primitives : BooleanSupplier, DoubleSupplier, IntSupplier et LongSupplier.
  L'interface fonctionnelle BooleanSupplier est une spécialisation de l'interface Supplier pour une valeur primitive de type booléenne.
  Elle définit la méthode fonctionnelle getAsBoolean() qui renvoie une valeur de type boolean.

```
package ma.formation.interfacesfonctionnelles;

import java.util.function.BooleanSupplier;

public class Test31 {

        public static void main(String[] args) {
                int a = 10;
                int b = 12;
                BooleanSupplier aInferieurAB = () -> a <= b;
                System.out.println(aInferieurAB.getAsBoolean());
        }
}
```

- L'interface fonctionnelle DoubleSupplier est une spécialisation de l'interface Supplier pour une valeur de type double.
  Elle définit la méthode fonctionnelle getAsDouble() qui renvoie une valeur flottante de type double.

```
package ma.formation.interfacesfonctionnelles;
```

```java
import java.util.function.DoubleSupplier;

public class Test32 {

	public static void main(String[] args) {
		DoubleSupplier pi = () -> 3.14116;
	    System.out.println(pi.getAsDouble());
	}
}
```

- L'interface fonctionnelle IntSupplier est une spécialisation de l'interface Supplier pour une valeur de type int.
  Elle définit la méthode fonctionnelle getAsInt() qui renvoie un entier de type int

```java
package ma.formation.interfacesfonctionnelles;

import java.util.function.IntSupplier;

public class Test33 {

	public static void main(String[] args) {
		int a = 10;
		   int b = 12;
		   IntSupplier aPlusB = () -> a + b;
		   System.out.println(aPlusB.getAsInt());
	}
}
```

- L'interface fonctionnelle LongSupplier est une spécialisation de l'interface Supplier pour une valeur de type long.
  Elle définit la méthode fonctionnelle getAsLong() qui renvoie un entier de type long.

```java
package ma.formation.interfacesfonctionnelles;

import java.util.function.LongSupplier;

public class Test34 {

	public static void main(String[] args) {
		int a = 100000000;
		LongSupplier aAuCarre = () -> (long) a * a;
		System.out.println(aAuCarre.getAsLong());
	}
}
```

Fin du TP 3.