Writing Data Processing Functions with Lambdas in Java 8

José Paumard

@JosePaumard | blog.paumard.org

Agenda

- Functional interfaces
- The `java.util.function` package
- The `Predicate` example

Functional Interfaces

How to define a type for lambda expressions

Functional Interfaces

- A lambda expression is an instance of a *functional interface*

```
public interface Predicate<T> {

   boolean test(T t);
}
```

- At this point, a *functional interface* is an interface with only one method

Functional Interfaces

- Let us implement Predicate (the JDK 7 way):

```
Predicate<String> p = new Predicate<String>() {

    public boolean test(String s) {
        return s.length() < 20;
    }
}
```

- And with a lambda expression:

Functional Interfaces

- Let us implement Predicate (the JDK 7 way):

```
Predicate<String> p = new Predicate<String>() {

    public boolean test(String s) {
        return s.length() < 20;
    }
}
```

- And with a lambda expression:

```
Predicate<String> p = (String s) -> s.length() < 20;
```

Functional Interfaces

- Let us implement Predicate (the JDK 7 way):

```
Predicate<String> p = new Predicate<String>() {

    public boolean test(String s) {
        return s.length() < 20;
    }
}
```

- And with a lambda expression:

```
Predicate<String> p = s -> s.length() < 20;
```

How Does It Work Under the Hood?

- The Java 8 compiler is smart!
  - The interface is *functional*, so there is only one method to implement
  - The type of the variable gives the type of the lambda expression
  - The parameters & return types must be compatible
  - The same for the exceptions, if any

- If all this holds, then the compiler can guess everything it needs

```
Predicate<String> p = s -> s.length() < 20;
```

A Lambda Is Still an Interface

- A lambda expression is still an implementation of an interface

```
Predicate<String> p = new Predicate<String>() {

    public boolean test(String s) {
        return s.length() < 20;
    }
}
```

```
Predicate<String> predicate = s -> s.length() < 20;
```

```
System.out.println(predicate.test("Hello World!"));
```

Functional Interface: Definition

- A functional interface is an interface:
  - With only one abstract method
  - Default methods do not count
  - Static methods do not count
  - Methods from the Object class do not count

Functional Interface: Definition

- A functional interface may be annotated with `@FunctionalInterface`
  - It is not mandatory, for legacy reasons
  - The compiler will tell us if an annotated interface is functional or not

The `java.util.function` Package

The functional interfaces toolbox

The `java.util.function`

- A new package from Java 8, with the most useful functional interfaces
- There are 43 of them!
- Four categories:

1) The Consumers
2) The Supplier
3) The Functions
4) The Predicates

The Consumers

- A consumer consumes an object, and does not return anything

```
public interface Consumer<T> {

   public void accept(T t);
}
```

```
Consumer<String> printer = s -> System.out.println(s);
                         = System.out::println;
```

The Consumers

- A consumer consumes an object, and does not return anything

```
public interface Consumer<T> {

   public void accept(T t);
}
```

```
public interface BiConsumer<T, V> {

   public void accept(T t, V v);
}
```

The Supplier

- A supplier provides an object, takes no parameter

```
public interface Supplier<T> {

   public T get();
}
```

```
Supplier<Person> personSupplier = () -> new Person();
                                = Person::new;
```

The Functions

- A function takes an object an returns another object

```
public interface Function<T, R> {

   public R apply(T t);
}
```

```
Function<Person, Integer> ageMapper = person -> person.getAge();
                                    = Person::getAge;
```

The Functions

- A function takes an object an returns another object

```
public interface Function<T, R> {

   public R apply(T t);
}
```

```
public interface BiFunction<T, V, R> {

   public R apply(T t, V v);
}
```

The Functions

- A function takes an object an returns another object

```
public interface UnaryOperator<T> extends Function<T, T> {
}
```

```
public interface BinaryOperator<T> extends BiFunction<T, T, T> {
}
```

The Predicates

- A predicate takes an object an return a boolean

```
public interface Predicate<T> {

   public boolean test(T t);
}
```

```
Predicate<Person> ageGT20 = person -> person.getAge() > 20;
```

The Predicates

- A predicate takes an object and returns a boolean

```
public interface Predicate<T> {

   public boolean test(T t);
}
```

```
public interface BiPredicate<T, U> {

   public boolean test(T t, U u);
}
```

Function Interfaces for Primitive Types

- Other functional interfaces have been defined, for instance:
- IntPredicate
- IntFunction
- IntToDoubleFunction

- Etc…

Live Coding

How to create new API in Java 8

Using functional interfaces

Using lambda expressions

Live Coding Summary

- The Predicate example:
  - From a simple functional interface
  - To a complete predicate API

- 3 things were used:
  - Lambda expressions
  - Default methods
  - Static methods

Summary

- How to write functional interfaces
  - How to build API in a new way
- New opportunities for our old legacy code
- New opportunities for our new API