Functional Programming with Java

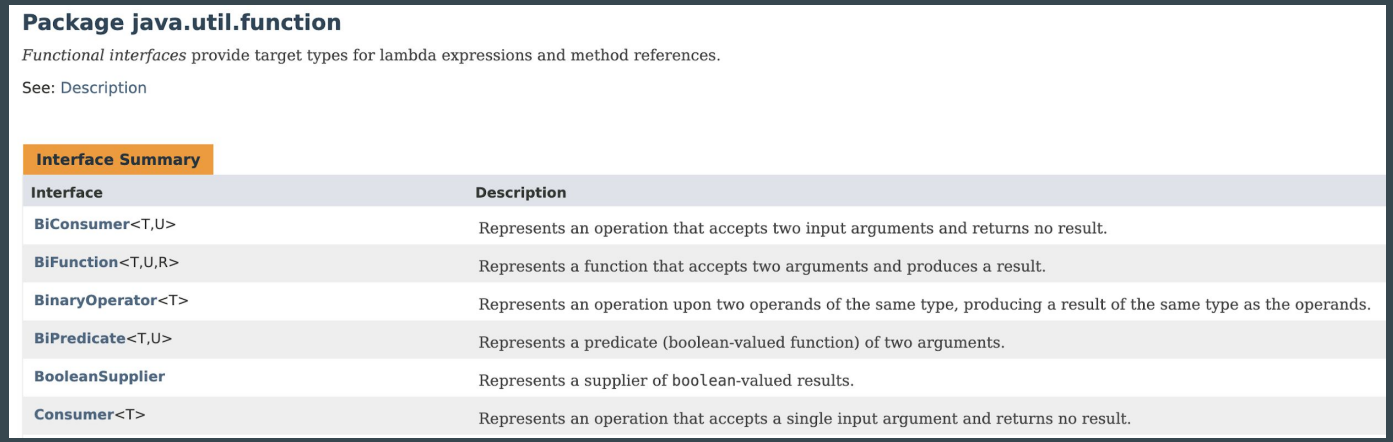
Oséas de Jesus Santana

Summary

- Inspiration
- Imperative vs Declarative
- Functional Programming
- Functional Interfaces, Lambda & Function
- Streams & Optionals
- Consumers, Suppliers & Predicates
- Functions and Callbacks
- Combinator Pattern
- Hands on

Inspiration

- Code reviews comments on Pull Requests;
- Some Slack feedbacks;
- Kyrius Menthoring: https://medium.com/@oseasjs/mentoria-desenvolvedor-java-parte-1-9f2b58e51aaa
- Amigos Code - Youtube channel: https://www.youtube.com/watch?v=VRpHdSFWGPs
- Java Functional Interfaces: https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html



Package java.util.function

Functional interfaces provide target types for lambda expressions and method references.

See: Description

Interface Summary

Interface	Description
BiConsumer<T,U>	Represents an operation that accepts two input arguments and returns no result.
BiFunction<T,U,R>	Represents a function that accepts two arguments and produces a result.
BinaryOperator<T>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
BiPredicate<T,U>	Represents a predicate (boolean-valued function) of two arguments.
BooleanSupplier	Represents a supplier of `boolean`-valued results.
Consumer<T>	Represents an operation that accepts a single input argument and returns no result.

Imperative vs Declarative

*"**Imperative programming** is a programming paradigm that uses statements that change a program's state."*

https://en.wikipedia.org/wiki/Imperative_programming

*"**Declarative programming** is a programming paradigm that expresses the logic of a computation without describing its control flow."*

https://en.wikipedia.org/wiki/Declarative_programming

*"**Imperative programming** is like **how** you do something, and **declarative programming** is more like **what** you do."*

https://tylermcginnis.com/imperative-vs-declarative-programming/

Functional Programming

"...formal system in mathematical logic for expressing computation based on function abstraction”

"With the help of a declarative programming style, FP tries to bind our code in pure mathematical functions to build evaluable expressions, instead of statements."

"Java was designed as a general-purpose programming language with class-based object-orientation at its core. With the release of version 8 in 2014, a more functional programming style became viable."

https://medium.com/better-programming/functional-programming-with-java-an-introduction-daa783355731

Functional Interfaces, Lambda & Function

"Java 8 brought a powerful new syntactic improvement in the form of lambda expressions. A **lambda** is an anonymous function that can be handled as a first-class language citizen, for instance passed to or returned from a method."

*"All **functional interfaces** are recommended to have an informative @FunctionalInterface annotation."*

"Any interface with a SAM (Single Abstract Method) is a functional interface, and its implementation may be treated as lambda expressions."

*"The most simple and general case of a lambda is a functional interface with a method that receives one value and returns another. This function of a single argument is represented by the **Function** interface which is parameterized by the types of its argument and a return value:"*

https://www.baeldung.com/java-8-functional-interfaces

Stream & Optional

```java
public List<Person> findByNameStartingWith(List<Person> list, String name) {
    return list.stream()
            .filter(p -> p.getName().startsWith(name))
            .collect(Collectors.toList());
}
```

```java
public String getPersonName(Optional<Person> person) {
    return person
            .map(p -> p.getName())
            .orElseThrow();
}
```

Consumer, Supplier & Predicate

```
Consumer<Person> checkIsAnAdult = person ->
    Optional
      .ofNullable(person)
      .filter(p -> p.getAge() > 18)
      .orElseThrow(() -> new RuntimeException(INVALID_PERSON_AGE));
```

```
static Supplier<List<Person>> adhocPersonList =
  () -> List.of(
    new Person("Matt", 20),
    new Person("Jane", 25)
);
```

```
static Predicate<Person> isPersonAnAdult =
    person -> person.getAge() > 18;
```

Function & Callbacks

```
Function<Person, Person> incrementOneYearOnPersonAge =
  (person) -> {
    person.setAge(person.getAge() + 1);
    return person;
};
```

```
public void filterPersonByNameAndDoSomethingWithIt(String personName,
                                List<Person> list, Consumer<Person> callback) {
    Person personFound = list
            .stream()
            .filter(p -> p.getName().equals(personName))
            .findFirst()
            .orElseThrow(() -> new RuntimeException(Person.NOT_FOUND_MESSAGE));
    callback.accept(personFound);
}
```

Combinator Pattern

```
static ValidatorCombinator isNameValid() {
    return person -> person.getName().startsWith("J") ?
        ValidationResult.SUCCESS : ValidationResult.IS_NOT_VALID_NAME;
}

static ValidatorCombinator isAnAdult() {
    return person -> person.getAge() > 18 ?
      ValidationResult.SUCCESS : ValidationResult.IS_NOT_AN_ADULT;
}

default ValidatorCombinator and (ValidatorCombinator other) {
    return person -> {
        ValidationResult result = this.apply(person);
        return result.equals(ValidationResult.SUCCESS) ? other.apply(person) : result;
    };
}
```

More details: https://gtrefs.github.io/code/combinator-pattern/

Hands on

Github: https://github.com/oseasjs/functional-programming