Treinamento de programação funcional com Java 8+

Estudar uma nova linguagem de programação

Estudar uma nova linguagem = fácil

Estudar um novo paradígma

Estudar uma nova linguagem = fácil

Estudar um novo paradígma = difícil

- Como todos sabem, programação funcional é um novo paradigma.
- E basicamente uma forma representar nossos códigos através de funções.
- "Programação funcional" está mais para uma maneira de pensar do que um conjunto de ferramentas.
- Não é a linguagem que te faz programar funcional, eh o jeito que vc escreve codigo, a única diferença eh que algumas linguagens são mais functional-friendly que outras.

A maiores diferenças entre programação funcional e programação imperativa eh que:

- No mutation of variables
- No printing to the console or to any device
- No writing to files, databases, networks, or whatever
- No exception throwing

Alguns princípios de programação funcional q veremos a seguir:

- First-class functions
- Closures
- Partial application
- Currying
- HOF
- Lazy evaluation
- Memoization

Criando uma lista de numeros inteiros de forma imperativa

```
List<Integer> forLoopRange(int from, int limit) {
        List<Integer> numbers = new ArrayList<>(limit);
        for(int to = from+limit; from < to; ++from) {
                numbers.add(from);
        }
        return numbers;
}
```

Criando uma lista de numeros inteiros de forma funcional

```
List<Integer> streamRange(int from, int limit) {
        return IntStream.range(from, from+limit)
                .boxed()
                .collect(toList());
}
---
```

Composição de funções

```
* sao blocos de funcoes que sao compostos de outros bloco
* f(x) = x + 2 and g(x) = x * 2
* f(g(x)) = f(g(5)) = f(5 * 2) = 10 + 2 = 12
* g(f(x)) = g(f(5)) = g(5 + 2) = 7 * 2 = 14
```

Funções com vários argumentos

```
* f(x, y) = x + y
* f(3,5) = 3 + 5 = 8
```

Currying

```
* f(x)(y) = g(y)
* where g(y) = x + y
* f(x) = g
* g(y) = x + y
* f(3)(5) = g(5) = 3 + 5 = 8

* f(rate, price) = price / 100 * (100 + rate) = f(rate)(p
* g(price, rate) = price / 100 * (100 + rate) = g(price)(
```

Functions in java

```
public interface Function {
  int apply(int arg);
}
```

Criando a primeira função

```
Function triple = new Function() {

    @Override
    public int apply(int arg) {
        return arg * 3;
    }
};
```

Executando a primeira função

```
System.out.println(triple.apply(2)); // 6
```

Vamos criar nossa segunda função

```
Function square = new Function() {

    @Override
    public int apply(int arg) {
        return arg * arg;
    }
};
```

Qual o benefício destas funções ?

Qual o benefício destas funções?

Nenhum ainda!

Composição de funções

```
System.out.println(square.apply(triple.apply(2))); // 36
```

Composição de funções

```
System.out.println(square.apply(triple.apply(2))); // 36
```

> Ainda não é composição de funções.

Composição de funções

```
Function compose(final Function f1, final Function f2) {
  return new Function() {
    @Override
    public int apply(int arg) {
      return f1.apply(f2.apply(arg));
    }
  };
}

System.out.println(compose(triple, square).apply(3)); //2
```

Composição de funções

```
Function compose(final Function f1, final Function f2) {
  return new Function() {
    @Override
    public int apply(int arg) {
      return f1.apply(f2.apply(arg));
    }
  };
}
System.out.println(compose(triple, square).apply(3)); //2
```

> Problema: aceitamos somente inteiros :(

Funcoes Polimórficas

```
public interface Function<T, U> {
  U apply(T arg);
}
```

Reescrevendo nossas funções

```
Function<Integer, Integer> triple = new Function<Integer,
  @Override
  public Integer apply(Integer arg) {
    return arg * 3;
  }
};
Function<Integer, Integer> square = new Function<Integer,
  @Override
  public Integer apply(Integer arg) {
    return arg * arg;
  }
};
```

Reescrevendo o compose utilizando tipos

```
Function<Integer, Integer> compose(
        final Function<Integer, Integer> f1,
        final Function<Integer, Integer> f2) {
                return new Function<Integer, Integer>() {
                        @Override
                        public Integer apply(Integer arg)
                                return f1.apply(f2.apply(a
                        }
        };
}
```

Melhorando um pouco mais o método compose

```
Function<Integer, Integer> compose(
        final Function<Integer, Integer> f1,
        final Function<Integer, Integer> f2) {
                return  (Integer arg)  -> f1.apply(f2.app
}
```

Melhorando ainda mais

```
BiFunction<
        Function<Integer, Integer>,
        Function<Integer, Integer>,
        Function<Integer, Integer>> compose2 =
                ( f1, f2) ->  arg -> f1.apply(f2.apply(ar
```

Ou piorou? Vcs decidem.

```
BiFunction<
        Function<Integer, Integer>,
        Function<Integer, Integer>,
        Function<Integer, Integer>> compose2 =
                (f1, f2) ->  z -> f1.apply(f2.apply(z));

Integer compose2Result = compose2
        .apply(x -> x + x, y -> y * y)
        .apply(3);

System.out.println(compose2Result);
```

Composição de funções é muito legal e poderoso mas:

- Quando implementado em java pode ser perigoso
- Composição de funções não avaliam antes de ir para a próxima fn

Currying

Quebrar uma função com vários parametros em outras funções que recebem somente com 1 argumento.

```
Function<Integer, Function<Integer, Integer>> add = x -> y
Function<Integer, Function<Integer, Integer>> mult = x ->
```

Aplicando funções currieds

```
System.out.println(add.apply(3).apply(5)); //8

// Scala | Javascript
add(3)(5)
//Haskell
add 3 5
```

HOC

Função que recebe funçoes e retorna uma função

```
Function<Function<Integer, Integer>,
         Function<Function<Integer, Integer>,
                  Function<Integer, Integer>>> compose =
                                    x -> y -> z -> x.ap
```

Reescrevendo

```
Function<Integer, Integer> triple = x -> x * 3;
Function<Integer, Integer> square = x -> x * x;

Function<Integer, Integer> f =
        compose
                .apply(square)
                .apply(triple);
System.out.println(f.apply(2)); // 36
```

Ou

```
Function<Integer, Integer> f =
        compose
                .apply(x -> x * x)
                .apply(y -> y * 3);

System.out.println(f.apply(2)); // 36
```

Closures

- Basicamente funcões com estado.
- Um closure permite o acesso do scopo da função externa a partir da função interna.

Closures

- Basicamente funcões com estado.
- Um closure permite o acesso do scopo da função externa a partir da função interna.

> Meio doido isso.

Exemplo

```
static Function<
        Integer,
        Function<Integer, Integer>> closureExemplo =
        a -> b -> {
                                    int c = 10;
                                    return a + b + c;
};


Integer applyResult = closureExemplo.apply(2).apply(3);
System.out.println(applyResult); // 15
```

Algumas interfaces funcionais Java 8

> java.util.function.Function

Representa uma fn que recebe um único argumento e retorna um único valor

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

    public <R> apply(T parameter);
}
```

> java.util.function.Predicate

Representa uma fn que recebe um único argumento e retorna um boolean

```
public interface Predicate {
    boolean test(T t);
}
```

> java.util.function.Predicate

> Exemplo simples de uma implementação de Predicate

```
public class CheckForNull implements Predicate {
    @Override
    public boolean test(Object o) {
        return o != null;
    }
}
```

> java.util.function.Predicate

> Mais simples ainda

```
Predicate predicate = (value) -> value != null;
```

- java.util.function.Supplier

```
public interface Supplier<T> {
        T get();
}
```

> java.util.function.Consumer

```
public interface Consumer<T> {
        void accept(T t);
        default Consumer<T> andThen(Consumer<? super T> a
                Objects.requireNonNull(after);
                return (T t) -> { accept(t); after.accept
        }
}
```

> java.util.function.Consumer

Exemplo

```
public class ConsumerExample {
        public static void main(String[] args) {
                Consumer<Integer> consumer = i -> System.
                printList(Arrays.asList(1, 2, 3, 4, 5, 6)
        }

        private static void printList(List<Integer> asLis
                for (Integer value : asList) {
                        consumer.accept(value);
                }
        }
}
```

> java.util.function.Consumer

Exemplo

```
public class ConsumerExample2 {
        public static void main(String[] args) {
                Consumer<Integer> consumer = i -> System.
                printList(Arrays.asList(1, 2, 3, 4, 5, 6)
        }

        private static void printList(List<Integer> asLis
                asList.forEach(consumer);
        }
}
```

> java.util.function.Consumer

Exemplo

```
public class ConsumerExample3 {

        public static void main(String[] args) {
                printList(Arrays.asList(1, 2, 3, 4, 5, 6)
        }

        private static void printList(List<Integer> asLis
                asList.forEach(consumer);
        }
}
```

> java.util.function.Consumer

Como utilizar o method andThen()

```
public class ConsumerExample {
        public static void main(String[] args) {
                Consumer<Integer> consumer = i -> System.
                Consumer<Integer> consumerWithThen = cons
                printList(Arrays.asList(1, 2, 3, 4, 5, 6)
        }

        private static void printList(List<Integer> asLis
                asList.forEach(consumer);
        }
```

> java.util.function.UnaryOperator

```
public interface UnaryOperator<T> extends Function<T, T> 
        static <T> UnaryOperator<T> identity() {
                return t -> t;
        }
}
```

> java.util.function.BinaryOperator

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

        public static <T> BinaryOperator<T> minBy(Compara

        public static <T> BinaryOperator<T> maxBy(Compara

}
```

Monadas

Segundo Martin Odersky, criador da Linguagem Scala, tem a seguinte definição:
"Mônadas são tipos parametrizados com duas operações principais, flatMap e unit, e que seguem algumas leis algébricas". Ou seja, as mônadas são contêiner que encapsulam valores e/ou computações.

Exemplo de Monadas em Java

Optional

```
Optional<String> nome = Optional.ofNullable(null);

System.out.println("Existe um nome? " + nome.isPresent())

System.out.println(
        "Nome: " + nome.orElseGet(() -> "[sem nome]"));

System.out.println(
        nome
      .map( s -> "Olá " + s + "!" )
      .orElse( "Olá estranho!" ) );

//Existe um nome? false
//  Nome: [sem nome]
//  Olá estranho!
```

Implementando Monada em Java

```
public class Result<T> {

        private Optional<T> value;
        private Optional<String> error;

        private Result(T value, String error) {
                this.value = Optional.ofNullable(value);
                this.error = Optional.ofNullable(error);
        }
        // ... outros métodos
}
```

```java
public class Result<T> {

        // ... outros métodos
        public static <U> Result<U> ok(U value) {
                return new Result<>(value, null);
        }
        public static <U> Result<U> error(String error) {
                return new Result<>(null, error);
        }
        public boolean isError() {
                return error.isPresent();
        }
        public T getValue() {
                return value.get();
        }
        public String getError() {
                return error.get();
        }
}
```

```
public class Result<T> {
        // ... outros métodos
        public <U> Result<U> flatMap(Function<T, Result<U>
                if(this.isError()) {
                        return Result.error(this.getError
                }
                return mapper.apply(value.get());
        }
}
```

Utilizando nossa Monada Result

```
public static void main(String[] args) {
        Result<Double> businessOperation =
                                businessOperation("Nome1"
        if(businessOperation.isError()) {
                System.out.println(businessOperation.getE
        } else {
                System.out.println(businessOperation.getVa
        }
}
```

Muito obrigado pela atenção

> Gilluan Formiga