

VEM SER

Módulo 01 - Java + OO

Aula 06 - Programação funcional

Conteúdo da aula

- CRUD
- Programação funcional no Java
- Interface funcional
- Function
- Lambda
- Lambda expressions
- StreamAPI
- Generics

CRUD

CRUD



Crédito da imagem: <https://www.luiztools.com.br/post/tutorial-crud-em-android-com-sqlite-e-recyclerview/>

CRUD

CREATE
(post)

Item adicionado

+ Adicionar item



READ
(get)

Item adicionado

UPDATE
(put / patch)

Item adicionado



DELETE
(delete)

Item adicionado



Programação funcional no Java

Programação funcional no Java

- A programação funcional é um paradigma de programação que se concentra em funções como a unidade fundamental de programação;
- As funções em programação funcional são funções puras, o que significa que não têm efeitos colaterais e sempre retornam o mesmo valor para os mesmos argumentos;
- Isso torna as funções muito fáceis de entender e testar.

Interfaces funcionais

- Interfaces funcionais são interfaces que contêm apenas um único método abstrato.
- No Java 8, foi introduzida a anotação ***@FunctionalInterface*** para indicar explicitamente que uma interface é funcional.

Vamos praticar!



- Criar uma interface funcional Calculo que retorne um número inteiro e receba dois parâmetros;
- Criar uma nova classe **Main** para testar onde:
 - Deve conter a implementação da interface funcional Calculo com a operação de soma;
 - Deve conter a implementação da interface funcional Calculo com a operação de multiplicação.

Interfaces funcionais do Java

Algumas interfaces funcionais importantes no pacote *java.util.function* incluem:

- **Function:** Representa uma função que recebe um argumento e retorna um resultado.
- **Predicate:** Representa uma função que recebe um argumento e retorna um valor booleano (verdadeiro ou falso).
- **Consumer:** Representa uma função que recebe um argumento e não retorna nenhum resultado.
- **Supplier:** Representa uma função que não recebe nenhum argumento, mas retorna um resultado.

Function

```
Function<Double, Double> log = (value) -> Math.log(value);  
Function<Double, Double> sqrt = (value) -> Math.sqrt(value);  
Function<Double, Double> logThenSqrt = sqrt.compose(log);  
System.out.println(logThenSqrt.apply(3.14));
```

// Output: 1.06

```
Function<Double, Double> sqrtThenLog = sqrt.andThen(log);  
System.out.println(String.valueOf(sqrtThenLog.apply(3.14)));
```

// Output: 0.57

Vamos praticar!



Utilize a interface Function do Java para:

- Criar uma função que realize a multiplicação por 10 de algum número;
- Criar outra função que realize a raiz quadrada;
- Crie uma outra função e junte as duas operações (andThen);
- Realize o cálculo e teste com a operação.

Lambda

Lambda

- Na década de 1930, o matemático Alonzo Church desenvolveu um sistema formal para expressar cálculos baseados na abstração de funções;
- Este modelo universal de computação veio a ser conhecido como Cálculo Lambda.

Lambda expressions

- As lambda expressions permitem criar instâncias de interfaces funcionais de forma concisa;
- Uma lambda expression é uma função anônima que pode ser passada como argumento para métodos, retornada de métodos ou atribuída a uma variável;
- A sintaxe básica é **(argumentos) -> expressão**.

Lambda expressions

```
Function<String, String> capitalizarFrase = s -> s.toUpperCase();  
System.out.println(capitalizarFrase.apply("hello world"));
```


Lambda expressions

```
Function<List<Integer>, Integer> somarImpares = numeros -> {  
    int soma = 0;  
    for (int numero : numeros) {  
        if (numero % 2 != 0) {  
            soma += numero;  
        }  
    }  
    return soma;  
};
```

Stream API

Stream API

- Os *streams* permitem processar coleções de elementos de forma declarativa, possibilitando operações como mapeamento, filtragem, redução e muito mais;
- A API de Stream oferece uma série de métodos intermediários e terminais para manipular e obter resultados a partir das coleções.



Stream API

Ao usar Streams, você descreve as operações que deseja executar nos elementos da coleção de forma declarativa, sem precisar se preocupar com a implementação detalhada dessas operações.

Stream API

- `Optional<T>` => classe responsável por tratar os valores nulos
- `.findFirst()` => retorna o primeiro elemento
- `.peek()` => executa o comando passado na expressão
- `.map(obj => outroObj)` => transforma um objeto em outro

Generics

- Em Java, os **Generics** são uma característica importante que permitem criar classes, interfaces e métodos que podem ser usados com **diferentes tipos de dados** de forma segura e flexível;
- O recurso de Generics permite que você crie **código reutilizável** que é **tipado de forma genérica**, proporcionando maior segurança de tipo em tempo de compilação e evitando a necessidade de fazer conversões e checagens de tipo em tempo de execução.

Vamos praticar!



- Na pasta “**modulo1**” criar um projeto chamado “**sistema-aluno**”;
- Elaborar um programa que cadastre alunos com os seguintes dados:
- Identificador, nome, idade, cidade, nota1, nota2 e nota3
- Crie uma lista com esses alunos cadastrados (para facilitar, deixe alguns previamente cadastrados nessa lista)
- Com essa lista faça alguns relatórios que o usuário poderá solicitar pelo menu interativo
- Relatório com todos os alunos que moram no Acre
- Relatório com alunos com mais de 18 anos
- Média das notas $[(n1 + n2 + n3) / 3]$ ordenada da maior para a menor média. Além disso, trazer todos os dados do aluno ao lado da média

Referências

<https://www.linkedin.com/pulse/lambda-expressions-em-java-papodev-thiago-ferreira-barbosa-ctfi/?originalSubdomain=pt>
https://www.w3schools.com/java/java_lambda.asp
<https://www.infoq.com/br/articles/java8-desmistificando-lambdas/>
<https://www.devmedia.com.br/como-usar-funcoes-lambda-em-java/32826>



Let's *Tech Up Together*