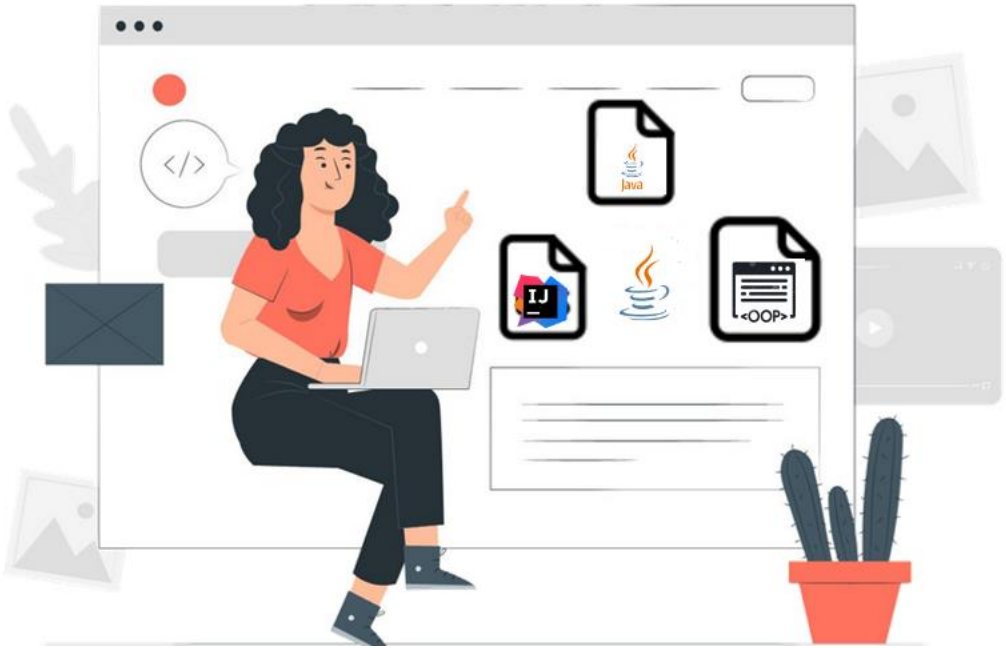


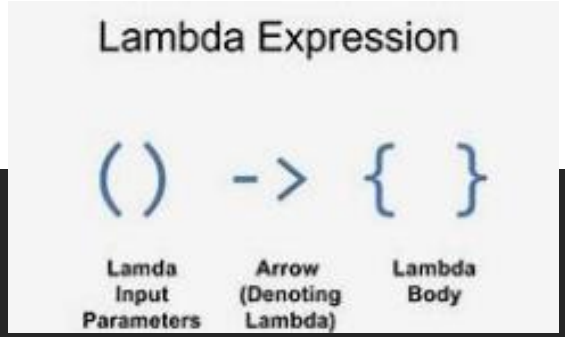

# Domain Driven Design

## Eliane Marion

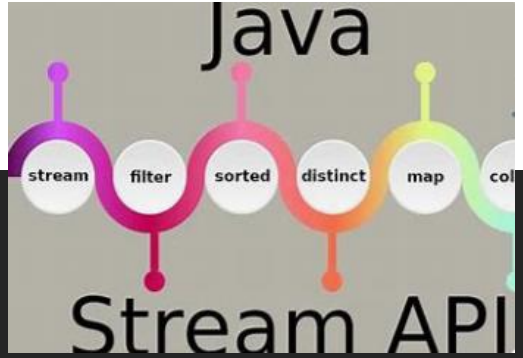
FIAP

2024

# AGENDA DE TRABALHO




LAMBDA




STREAM

# 01

## Lambda

# LAMBDA

- Introduzidas no Java 8.
- Permite uma escrita mais concisa.
- Sempre têm um parâmetro e, após a flecha **->** , há uma expressão que indica o que ocorrerá com esse parâmetro

# EXEMPLO

```
Aluno((parâmetro) -> expressao);
```

o parâmetro vai representar cada um dos alunos

O Java interpreta o parâmetro dependendo do contexto

{}

```
alunos.forEach(a -> a.getNotas().forEach(
                n -> System.out.println(n.getValor())));
```

# Lambda e Stream

As funções lambda não foram criadas simplesmente para encurtar o código.

O Java 8 introduziu uma API muito importante, a de **Streams**, que permite executar muitas operações interessantes.

A combinação entre funções lambda e Streams confere grande poder ao desenvolvimento Java.

Isso significa que em uma única linha conseguimos realizar diversas operações usando essas duas ferramentas..

# Stream

Um *Stream* é um fluxo de dados.

```
List<String> nomes =
Arrays.asList("Ana", "Maria",
"Paulo", "Rodrigo", "Eliane");
```

Essa lista de nomes representa um fluxo de dados. Com ele, podemos realizar um fluxo de operações encadeadas, o que é a grande vantagem dos Streams.

Cada operação gera um novo Stream, um novo fluxo, então podemos realizar uma operação neste fluxo, seguida de outra operação, e assim por diante.

# EXEMPLO

```
public class Principal {
    public static void main(String[] args) {
        List<String> nomes = Arrays.asList("Ana", "Maria",
                "Paulo", "Rodrigo", "Eliane");



nomes.stream()
        .sorted()
        .foreach(System.out::println);
```

A função sorted() é uma operação intermediária que gera um novo fluxo

# EXEMPLO

```
public class Principal {
    public static void main(String[] args) {
        List<String> nomes = Arrays.asList("Ana", "Maria",
                "Paulo", "Rodrigo", "Eliane");



nomes.stream()
        .sorted()
        .limit(3)
        .foreach(System.out::println);
```

A função limit() é uma operação intermediária que gera um novo fluxo

# EXEMPLO

```
public class Principal {
    public static void main(String[] args) {
        List<String> nomes = Arrays.asList("Ana", "Maria",
                "Paulo", "Rodrigo", "Eliane");



nomes.stream()
        .sorted()
        .limit(3)
        .filter(n -> n.startWhith("M"))
        .foreach(System.out::println);
```

A função limit() é uma operação intermediária que gera um novo fluxo

# EXEMPLO

```
public class Principal {
    public static void main(String[] args) {
        List<String> nomes = Arrays.asList("Ana", "Maria",
                "Paulo", "Rodrigo", "Eliane");

nomes.stream()
        .sorted()
        .limit(3)
        .filter(n -> n.startWhith("M"))
        .map(n -> n.toUpperCase())
        .foreach(System.out::println);
```

A função limit() é uma operação intermediária que gera um novo fluxo

# STREAM

As operações intermediárias são aquelas que podem ser aplicadas em uma stream e **retornam uma nova stream como resultado**. Essas operações não são executadas imediatamente, mas apenas quando uma operação final é chamada.

Exemplos e fluxos intermediários:

**Filter**: permite filtrar os elementos da stream com base em uma condição. Por exemplo, podemos filtrar uma lista de números para retornar apenas os números pares.

# STREAM

{}

```
List<Integer> numeros = Arrays.asList(1,2,3,4,5,6,7,8,9);

List<Integer> numerosPares = numeros.stream()
                                .filter(n -> n % 2 == 0)

                                .collect(Collectors.toList());

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

# STREAM

**Map**: permite transformar cada elemento da stream em outro tipo de dado.

```
List<String> palavras = Arrays.asList("Java", "Stream",
                        "Operações", "Intermediárias");

List<Integer> tamanhos = palavras.stream()
                        .map(s -> s.length())
                        .collect(Collectors.toList());

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

# STREAM

As operações **finais** são aquelas que encerram a stream e **retornam** um resultado concreto. Algumas operações finais comuns são o **forEach**, **collect** e **count**.

**ForEach**: permite executar uma ação em cada elemento da stream.

```
List<String> nomes = Arrays.asList("Ana", "Maria",
                "Paulo", "Rodrigo", "Eliane");
nomes.stream()
        .forEach(nome -> System.out.println("Olá," + nome));
```

# STREAM

**Collect**: permite coletar os elementos da stream em uma coleção ou em outro tipo de dado.

```
List<Integer> numeros = Arrays.asList(1,2,3,4,5,6,7,8,9);

Set<Integer> numerosPares = numeros.stream()
                                   .filter(n -> n % 2 == 0)

                                   .collect(Collectors.toSet());

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

# Stream

Além das operações intermediárias e finais mencionadas, existem muitas outras disponíveis

| Operações | Descrição |
| --- | --- |
| distinct | Remove elementos duplicados. |
| limit | Limita o número de elementos da stream. |
| skip | Pula os primeiros elementos da stream. |
| reduce | Combina elementos da stream em um único resultado. |

# STREAM

```
System.out.println("==========================================
==========");
System.out.println("Lista de episódios por temporada ordenado
por avaliação");
 dadosEpisodios = temporadas.stream()
        .flatMap(t -> t.getEpisodios().stream())
         .sorted(Comparator.comparing(Episodio::getAvaliacao).re
versed())
        .limit(5)
        .collect(Collectors.toList());

dadosEpisodios.forEach(System.out::println);
```

# STREAM

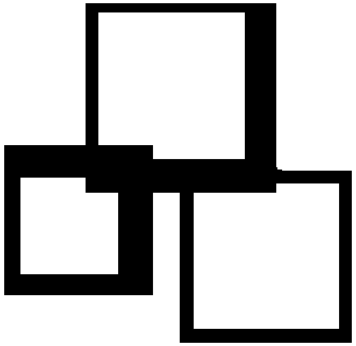
```
System.out.println("========================================
========");
System.out.println("Lista de episódios por temporada");
List<Episodio> episodios = temporadas.stream()
        .flatMap(t -> t.getEpisodios().stream()
                .map(d -> new Episodio(t.getNumeroTemporada(), d))
        ).collect(Collectors.toList());

episodios.forEach(System.out::println);
```

# Novidades

```
nomes.forEach(System.out::println);
```

É possível simplificar ainda mais o exemplo de código anterior, utilizando o recurso conhecido como Method Reference, que nada mais é do que uma forma reduzida de uma expressão lambda:



# OBRIGADO


To be continued...