Programação Avançada - AULA 08

Matheus Moresco
Engenharia de Software - 5º Período
2025/01

Objetivos da Aula

- **Compreender o conceito de expressões lambda** e sua aplicação na programação funcional.
- **Explorar a API de Streams do Java** e seus principais métodos.
- **Implementar soluções utilizando expressões lambda e streams** para manipulação eficiente de coleções.
- **Comparar abordagens tradicionais vs. programação funcional** para otimizar código.

Expressões Lambda em Java

- As **expressões lambda** são uma forma concisa de escrever funções anônimas em Java. Elas permitem criar **funções sem precisar definir explicitamente uma classe ou método**, facilitando a programação funcional.

Sintaxe de uma Expressão Lambda

- Uma expressão lambda em Java segue o seguinte formato:

```
(parametros) -> { corpo da função }
```

- Exemplo simples:

```
// Expressão lambda para somar dois números
(int a, int b) -> { return a + b; }
```

Sintaxe de uma Expressão Lambda

Além do exemplos simples do uso das funções lambdas, podemos abstrair ainda mais o função, das seguintes maneiras:

- Omitindo tipos dos parâmetros (inferência de tipo):

```
(a, b) -> a + b  // O Java infere que são inteiros
```

- Sem parâmetros:

```
() -> System.out.println("Olá, mundo!")
```

- Um parâmetro (sem necessidade de parênteses):

```
nome -> "Olá, " + nome
```

Exemplo com Interface Funcional

- Uma **interface funcional** é uma interface que possui **apenas um método abstrato**. Expressões lambda são frequentemente usadas com interfaces funcionais.

Usando classes anônimas

```
interface Saudacao {
    void mensagem();
}

public class App {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        Saudacao saudacao = new Saudacao() {
            public void mensagem() {
                System.out.println(x:"Olá, mundo!");
            }
        };
        saudacao.mensagem();
    }
}
// Output: Olá, mundo!

```

Usando classes abstratas

```
interface Saudacao {
    void mensagem();
}

public class App {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        Saudacao saudacao = () -> System.out.println(x:"Olá, mundo!");
        saudacao.mensagem();
    }
}
// Output: Olá, mundo!

```

Principais Aplicações das Expressões Lambda

- Uso com Streams:

```
List<String> nomes = Arrays.asList(...a:"Alice", "Bruno", "Carlos");
nomes.stream().forEach(nome -> System.out.println(nome));
```

- Uso com Threads:

```
new Thread(() -> System.out.println(x:"Thread em execução!")).start();
```

- Uso com Comparator (Ordenação de Listas):

```
List<String> lista = Arrays.asList(...a:"Banana", "Maçã", "Uva");
lista.sort((s1, s2) -> s1.compareTo(s2));
System.out.println(lista);
```

Vantagens das Expressões Lambda

Expressões lambda são uma forma simplificada de escrever funções anônimas, tornando o código mais limpo e funcional. Elas são amplamente utilizadas em **Streams**, **Threads**, e **estruturas de dados** em Java!

- ✓Código mais **curto** e **legível**.
- ✓Facilita a **programação funcional**.
- ✓Reduz a necessidade de **classes anônimas**.

Streams em Java

- Os **Streams** foram introduzidos no Java 8 e fornecem uma maneira funcional e eficiente de **processar coleções de dados**.
- Principais características:
  - Processamento Funcional – Usa métodos como **map**(), **filter**(), **reduce**() e **forEach**().
  - Operações em Pipeline – Encadeia operações para manipulação de dados.
  - Execução Paralela – Usa **parallelStream**() para melhorar desempenho.
  - Imutabilidade – Não modifica a coleção original, gerando um novo resultado.

Criando uma Stream em Java

- Um **Stream** pode ser criado a partir de coleções, arrays, arquivos e até mesmo gerado manualmente.
- Criando um Stream a partir de uma lista:

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class App {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        List<String> nomes = Arrays.asList(...a:"Ana", "Bruno", "Carlos");

        Stream<String> streamNomes = nomes.stream();
        streamNomes.forEach(System.out::println);
    }
}
```

Operações Principais com Streams

- As operações em Streams podem ser **intermediárias** (transformam o Stream) ou **terminais** (produzem um resultado).

  - **filter()** – Filtrando elementos;
  - **map()** – Transformando Dados
  - **sorted()** – Ordenando Elementos
  - **forEach()** – Iterando sobre os elementos
  - **reduce()** – Reduzindo um Stream a um único valor

Operações Streams - Filter

- Filtra os elementos com base em uma condição.
- **Exemplo**: Obter apenas os números pares de uma lista.

```java
public class App {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        List<Integer> numeros = Arrays.asList(...a:1, 2, 3, 4, 5, 6);

        List<Integer> pares = numeros.stream()
                .filter(n -> n % 2 == 0)
                .collect(Collectors.toList());

        System.out.println(pares); // Saída: [2, 4, 6]
    }
}

```

Operações Streams - Map

- Transforma cada elemento do Stream aplicando uma função.
- **Exemplo**: Converter uma lista de palavras para maiúsculas.

```
5  public class App {
       Run | Debug | Run main | Debug main
6      public static void main(String[] args) {
7          List<String> palavras = Arrays.asList(...a:"java", "stream", "lambda");
8
9          List<String> maiusculas = palavras.stream()
10                 .map(String::toUpperCase)
11                 .collect(Collectors.toList());
12
13         System.out.println(maiusculas); // Saída: [JAVA, STREAM, LAMBDA]
14     }
15 }
16
```

Operações Streams - Sorted

- Ordena os elementos do Stream.
- **Exemplo**: Ordenar uma lista de números em ordem decrescente.

```
5   public class App {
        Run | Debug | Run main | Debug main
6       public static void main(String[] args) {
7           List<Integer> numeros = Arrays.asList(...a:5, 3, 8, 1, 9);
8
9           List<Integer> ordenados = numeros.stream()
10                  .sorted((a, b) -> b - a)
11                  .collect(Collectors.toList());
12
13          System.out.println(ordenados); // Saída: [9, 8, 5, 3, 1]
14      }
15  }
16
```

Operações Streams – For Each

- Executa uma ação para cada elemento do Stream.
- **Exemplo**: Exibir os nomes de uma lista.

```
4   public class App {
        Run | Debug | Run main | Debug main
5       public static void main(String[] args) {
6           List<String> nomes = Arrays.asList(...a:"Alice", "Bob", "Carlos");
7
8           nomes.stream()
9               .forEach(nome -> System.out.println("Olá, " + nome));
10      }
11  }
12
```

Operações Streams - Reduce

- Faz uma operação de redução, combinando os elementos do Stream.
- **Exemplo**: Somar todos os números de uma lista.

```java
public class App {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        List<Integer> numeros = Arrays.asList(...a:1, 2, 3, 4, 5);

        int soma = numeros.stream()
                .reduce(identity:0, (acumulador, n) -> acumulador + n);

        System.out.println(soma); // Saída: 15
    }
}
```

Streams Paralelos

- Os Streams podem ser **executados em paralelo**, distribuindo a carga entre múltiplas threads.
- Melhor desempenho para grandes conjuntos de dados!
- **Exemplo:** Processando uma lista em paralelo

```java
import java.util.Arrays;
import java.util.List;

public class App {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        List<String> nomes = Arrays.asList(...a:"Ana", "Bruno", "Carlos", "Daniel");

        nomes.parallelStream()
            .forEach(nome -> System.out.println(Thread.currentThread().getName() + " - " + nome));
    }
}

```

Streams vs. Abordagem Tradicional

- Vamos comparar como resolver o mesmo problema com e sem Streams.
  - Tradicional (for loop):

```
5       public static void main(String[] args) {
6           List<String> nomes = Arrays.asList(...a:"Ana", "Bruno", "Carlos");
7           for (String nome : nomes) {
8               if (nome.startsWith(prefix:"A")) {
9                   System.out.println(nome);
10              }
11          }
12      }
```

  - Com Streams (mais simples e legível):

```
5       public static void main(String[] args) {
6           List<String> nomes = Arrays.asList(...a:"Ana", "Bruno", "Carlos");
7           nomes.stream()
8               .filter(nome -> nome.startsWith(prefix:"A"))
9               .forEach(System.out::println);
10      }
```

Resumo

- ✓ **Streams** facilitam o processamento de coleções de maneira funcional.
- ✓ Permitem **filtrar, transformar, ordenar e reduzir** dados de forma eficiente.
- ✓ Podem ser **executados em paralelo** para melhor desempenho.
- ✓ São imutáveis e não alteram a coleção original.

Exemplo Prático

- Criar uma lista de Pessoas e usar o stream para filtrar, percorrer e manipular as pessoas.

Exercício

1. Na classe Empresa, crie uma lista de elementos da classe Produto.
2. Crie uma stream de dados para filtrar essa lista por preço, quantidade de produtos e nome do produto.