

TÓPICO 07 - PROGRAMAÇÃO FUNCIONAL EM JAVA

Disciplina de Backend - Professor Ramon Venson - SATC 2026.2

O que é Programação Funcional?

- É um **paradigma de programação** que trata o cálculo como avaliação de **funções**
- Em vez de manipular o estado de objetos, o foco está em **transformar dados**
- O código fica mais **declarativo**: dizemos *o que* queremos, não *como* fazer
- No Java, isso é feito principalmente com **lambdas** e a API de **Streams**

Lambdas

Uma **lambda** é uma forma compacta de representar uma função anônima (sem nome):

```
// Forma tradicional (classe anônima)
Runnable r1 = new Runnable() {
    public void run() {
        System.out.println("Olá!");
    }
};

// Com lambda
Runnable r2 = () -> System.out.println("Olá!");
```

A sintaxe básica é: (parâmetros) -> corpo

forEach: percorrendo coleções

O método `forEach` percorre todos os elementos de uma coleção e executa uma ação para cada um deles:

```
List<String> nomes = List.of("Ana", "Bruno", "Carla");

// Sem forEach
for (String nome : nomes) {
    System.out.println(nome);
}

// Com forEach (lambda)
nomes.forEach(nome -> System.out.println(nome));
```

Method References

Quando a lambda apenas chama um método, podemos usar uma sintaxe ainda mais curta: a **method reference** (`Classe::método`):

```
// Equivalente a: nome -> System.out.println(nome)
nomes.forEach(System.out::println);

// O mesmo vale para outros métodos
List<String> maiusculas = nomes.stream()
    .map(String::toUpperCase)
    .toList();
```

Streams

Um **Stream** representa uma sequência de elementos que pode ser processada com operações funcionais.

`list.stream()` cria um stream a partir de uma coleção. As operações se dividem em:

- **Intermediárias:** transformam o stream (ex.: `map`, `filter`, `sorted`, `distinct`)
- **Terminais:** produzem o resultado final (ex.: `toList()`, `forEach`, `count`)

```
List<Integer> numeros = List.of(3, 1, 4, 1, 5);  
numeros.stream().distinct().sorted().toList(); // [1, 3, 4, 5]
```

map: transformando elementos

map aplica uma função a cada elemento e gera um novo stream com os resultados:

```
List<String> nomes = List.of("ana", "bruno", "carla");  
List<String> maiusculas = nomes.stream()  
    .map(nome -> nome.toUpperCase())  
    .toList();  
// [ANA, BRUNO, CARLA]
```

filter: selecionando elementos

`filter` mantém apenas os elementos que satisfazem uma condição (uma expressão que retorna `boolean`):

```
List<Integer> numeros = List.of(1, 2, 3, 4, 5, 6);  
List<Integer> pares = numeros.stream()  
    .filter(numero -> numero % 2 == 0)  
    .toList();  
// [2, 4, 6]
```

sorted e distinct

- `sorted()` ordena os elementos usando a ordem natural
- `distinct()` elimina elementos duplicados

```
List<Integer> numeros = List.of(5, 3, 8, 3, 1);  
numeros.stream().sorted().distinct().toList(); // [1, 3, 5, 8]
```

Ordenação com lambda

Para ordenar objetos (que não possuem ordem natural), usamos um `Comparator`, que pode ser escrito como lambda:

```
List<Aluno> alunos = List.of(...);

// Ordena por nota, da maior para a menor
alunos.sort((a1, a2) -> Double.compare(a2.getNota(), a1.getNota()));

// O mesmo comportamento usando Comparator
alunos.sort(Comparator.comparing(Aluno::getNota).reversed());
```

Operações terminais

Além de `toList()`, existem outras operações terminais importantes:

```
long total = numeros.stream().count();           // conta elementos
numeros.stream().forEach(System.out::println);    // percorre o stream

// collect: versão clássica do toList()
List<Integer> lista = numeros.stream()
    .collect(Collectors.toList());
```

`toList()` (Java 16+) retorna uma lista **imutável**;

`collect(Collectors.toList())` retorna uma lista **mutável**.

reduce e agregações

`reduce` combina todos os elementos em um único valor (soma, produto, concatenação...):

```
List<Integer> numeros = List.of(1, 2, 3, 4, 5);  
  
int soma = numeros.stream().reduce(0, (a, b) -> a + b); // 15
```

Para valores numéricos, existem atalhos prontos:

```
int total = numeros.stream().mapToInt(n -> n).sum();  
double media = numeros.stream().mapToInt(n -> n).average().orElse(0);
```

Verificando condições

- `anyMatch` : algum elemento satisfaz a condição?
- `allMatch` : todos os elementos satisfazem a condição?
- `findFirst` / `findAny` : retorna um elemento (dentro de um `Optional`)

```
boolean alguemPassou = notas.stream().anyMatch(nota -> nota >= 6);  
boolean todosPassaram = notas.stream().allMatch(nota -> nota >= 6);  
Optional<Aluno> primeiro = alunos.stream().findFirst();
```

Exemplo completo

```
List<Aluno> alunos = List.of(  
    new Aluno("Ana", 8.5),  
    new Aluno("Bruno", 5.0),  
    new Aluno("Carla", 7.0)  
);  
  
// Nomes dos alunos aprovados, em ordem alfabética  
List<String> aprovados = alunos.stream()  
    .filter(aluno -> aluno.getNota() >= 6)  
    .map(Aluno::getNome)  
    .sorted()  
    .toList();  
// [Ana, Carla]
```

O que aprendemos hoje

- O paradigma funcional e a sintaxe de lambdas
- `forEach` e method references
- A API de Streams: `map`, `filter`, `sorted`, `distinct`
- Operações terminais: `toList()`, `collect()`, `count()`
- Agregação com `reduce` e atalhos numéricos (`sum`, `average`)
- Verificação com `anyMatch`, `allMatch`, `findFirst`
- Ordenação de objetos com lambdas