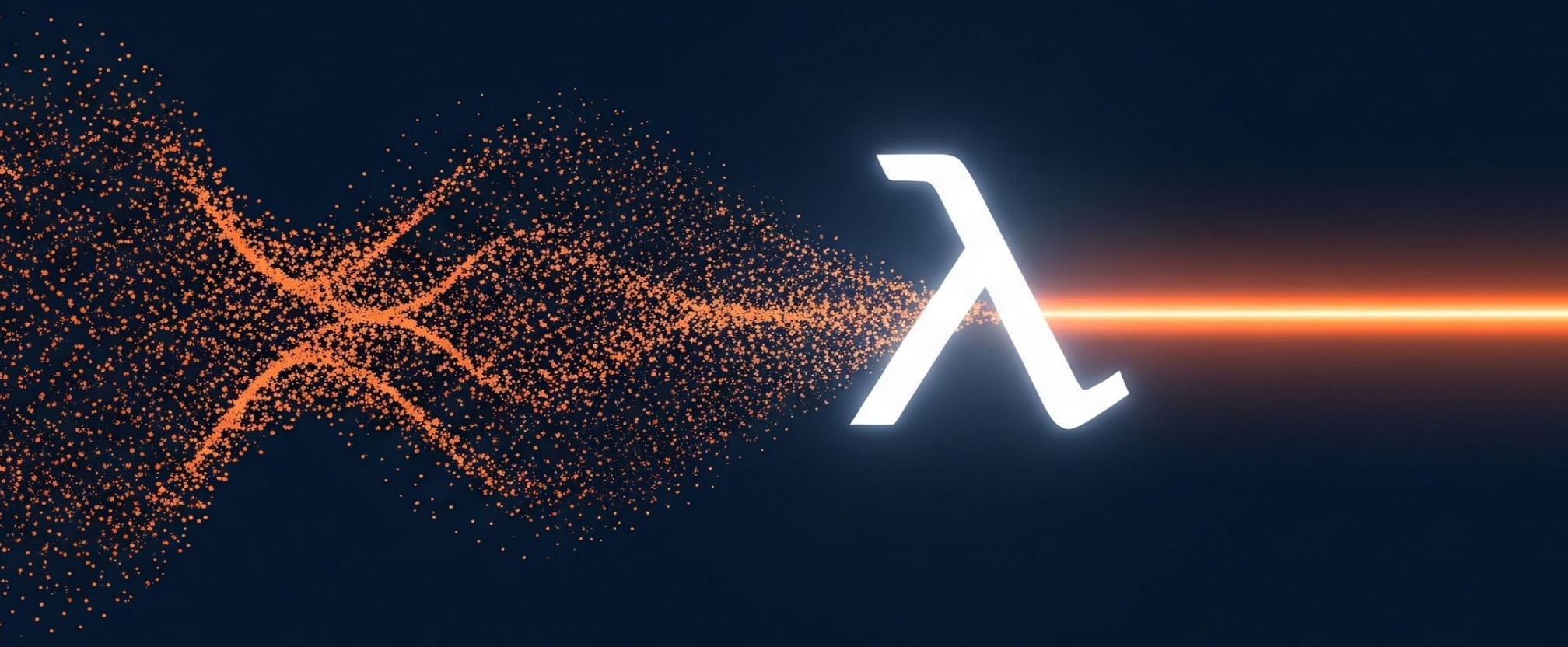
Java Funcional

Guia Prático de Streams e Lambdas



Aprenda a processar dados em Java de maneira clara e eficiente

ROGERIO RICARDO

BEM-VINDO AO FUTURO DO JAVA!

Se você já escreveu laços for para processar uma lista de dados e sentiu que deveria haver uma maneira mais inteligente, este guia é para você. Se você já ouviu os termos "lambda" e "stream" e quer entender o que eles realmente significam na prática, você está no lugar certo.

Este ebook foi criado para o desenvolvedor Java que já entende os fundamentos da linguagem (variáveis, laços, objetos), mas que deseja dar o próximo passo e aprender o estilo de programação funcional que revolucionou o Java a partir da versão 8.

O que você vai aprender:

- O "porquê": Entender as limitações do estilo antigo e por que a programação funcional se tornou essencial.
- Expressões Lambda: Dominar a sintaxe concisa e poderosa das lambdas para passar comportamento como se fosse um dado.
- A API de Streams: Aprender a criar pipelines de dados para filtrar, mapear, transformar e coletar informações de forma fluida e legível.
- Mão na Massa: Aplicar tudo o que aprendeu em um projeto prático e realista.

01

A VIDA ANTES DAS LAMBDAS

A Vida Antes das Lambdas

Antes de mergulharmos no novo, vamos entender o problema antigo. Imagine que você tem uma lista de nomes e quer imprimi-los. Simples, certo?

```
List<String> nomes = List.of("Ana", "Marcos", "Beatriz");
for (String nome : nomes) {
    System.out.println(nome);
}
```

Agora, imagine que você quer fazer algo um pouco mais complexo, como criar um botão em uma interface gráfica que imprime uma mensagem quando clicado. Antes do Java 8, você precisaria de algo chamado classe anônima:

```
// O jeito antigo e verboso
JButton botao = new JButton("Clique em mim!");
botao.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("O botão foi clicado!");
    }
});
```

Veja quantas linhas de código (new ActionListener(), @Override, a declaração do método...) apenas para passar uma única ação: System.out.println(...). Era verboso e desviava a atenção do que realmente importava.

Os desenvolvedores precisavam de uma forma mais direta para dizer "pegue esta ação e execute-a". Essa necessidade deu origem às expressões lambda.

02

EXPRESSÕES LAMBDAS: A SIMPLICIDADE EM AÇÃO

Uma expressão lambda é, essencialmente, uma função anônima que você pode tratar como um valor. É uma forma de passar um bloco de código para ser executado depois.

Vamos transformar o exemplo do botão usando uma lambda:

```
// O jeito novo, limpo e direto
JButton botao = new JButton("Clique em mim!");
botao.addActionListener(e → System.out.println("O botão foi clicado!"));
```

Impressionante, não é? Toda aquela cerimônia da classe anônima foi substituída pelo simples e -> ....

A Estrutura de uma Lambda:
Uma lambda tem três partes:

1. Parâmetros: (e) ou apenas e se for um só.

2. Seta: ->

3. Corpo: O código a ser executado. System.out.println(...)

Vamos voltar ao nosso primeiro exemplo da lista de nomes:

```
List<String> nomes = List.of("Ana", "Marcos", "Beatriz");

// Usando o método forEach com uma lambda
nomes.forEach(nome → System.out.println(nome));
```

Novamente, o código se torna mais declarativo. Em vez de dizer "crie um laço, pegue cada item e imprima", estamos dizendo "para cada nome na lista, imprima-o". É uma mudança sutil na linguagem, mas poderosa na prática.

03

OLA STREAMS

Agora que entendemos as lambdas, podemos conhecer sua parceira mais poderosa: a API de Streams.

O que é uma Stream?
Uma Stream é uma sequência de elementos vinda de uma fonte de dados (como uma List ou um Array) que suporta operações de agregação. Pense nela como uma linha de produção para seus dados.

Características principais:

- Não armazena dados: A stream não é uma estrutura de dados. Ela transporta os dados da fonte através de um pipeline de operações.

- É imutável: Uma operação em uma stream produz uma nova stream, sem modificar a coleção original.

- Operações em pipeline: Você pode encadear várias operações.

Como obter uma Stream?

É muito fácil. Quase toda coleção em Java tem o método .stream().

```
List<String> nomes = List.of("Ana", "Marcos", "Beatriz");
Stream<String> streamDeNomes = nomes.stream(); // Pronto!
```

Com nossa stream em mãos, podemos começar a nossa linha de produção. Uma linha de produção típica tem:

1. Fonte (A coleção original).

2. Operações Intermediárias (Filtros, transformações, etc. Ex: filter(), map()).

3. Uma Operação Terminal (Que inicia o processo e produz um resultado. Ex: forEach(), collect()).

04

A LINHA DE PRODUÇÃO: FILTER E MAP

1. Filtrando com filter()

filter é como um controle de qualidade. Ele recebe uma lambda que retorna true ou false (um Predicate). Se a lambda retornar true, o elemento passa para a próxima fase.

Exemplo: Pegar apenas os nomes que começam com a letra 'A'.

```
List<String> nomes = List.of("Ana", "Marcos", "Beatriz", "Amanda");

nomes.stream()
     .filter(nome → nome.startsWith("A")) // Operação intermediária
     .forEach(nome → System.out.println(nome)); // Operação terminal
```

Saída:

```
Ana
Amanda
```

2. Transformando com map()

map transforma cada elemento em outra coisa. Ele recebe uma lambda que aplica uma função a cada elemento.

Exemplo: Transformar cada nome em sua versão em maiúsculas e obter o seu tamanho.

```java
List<String> nomes = List.of("Ana", "Marcos", "Beatriz");

nomes.stream()
     .map(nome → nome.toUpperCase()) // Transforma em maiúsculas
     .map(nomeMaiusculo → nomeMaiusculo.length()) // Transforma em seu tamanho
     .forEach(tamanho → System.out.println(tamanho));
```

Saída:

05

COLETANDO OS RESULTADOS COM COLLECT

Até agora, apenas imprimimos os resultados com forEach. Mas e se quisermos criar uma nova lista com os resultados do nosso pipeline? Para isso, usamos a operação terminal collect().

Exemplo: Criar uma nova lista contendo apenas os nomes que começam com 'B'.

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

// A mágica acontece aqui
List<String> nomesComB = nomes.stream()
    .filter(nome → nome.startsWith("B"))
    .collect(Collectors.toList()); // Coleta os resultados em uma nova lista

System.out.println(nomesComB); // Imprime a nova lista
```

Saída:

```
[Beatriz, Bruno]
```

O método collect é extremamente versátil. Collectors.toList() é o coletor mais comum, mas você também pode coletar para outras estruturas de dados, como Set (Collectors.toSet()) ou Map.

06

PROJETO PRÁTICO: ANALISADOR DE FILMES

Vamos juntar tudo. Temos uma lista de filmes e queremos extrair algumas informações.

1. A Classe Filme

```java
class Filme {
    String titulo;
    int ano;
    double nota;

    // Construtor, getters...
}
```

2. A Fonte de Dados

```java
List<Filme> filmes = List.of(
    new Filme("O Poderoso Chefão", 1972, 9.2),
    new Filme("Parasita", 2019, 8.6),
    new Filme("O Senhor dos Anéis", 2001, 8.8),
    new Filme("Matrix", 1999, 8.7),
    new Filme("Interestelar", 2014, 8.6)
);
```

Desafio 1: Encontrar todos os filmes com nota maior ou igual a 9.0.

```
filmes.stream()
      .filter(filme → filme.getNota() ⩾ 9.0)
      .forEach(filme → System.out.println(filme.getTitulo()));
// Saída: O Poderoso Chefão
```

Desafio 2: Criar uma lista com os títulos de todos os filmes lançados após o ano 2000.

```
List<String> titulosRecentes = filmes.stream()
    .filter(filme → filme.getAno() > 2000)
    .map(filme → filme.getTitulo()) // Transforma de Filme para String (título)
    .collect(Collectors.toList());

System.out.println(titulosRecentes);
// Saída: [Parasita, O Senhor dos Anéis, Interestelar]
```

07

CONCLUSÕES E PRÓXIMOS PASSOS

Parabéns!

Você acaba de dar um passo gigantesco para se tornar um desenvolvedor Java mais moderno e eficiente.

O que você conquistou:

- Você aprendeu a simplificar seu código com expressões lambda.
- Você aprendeu a construir pipelines de dados elegantes e legíveis com a API de Streams.
- Você sabe como filtrar, transformar e coletar dados de coleções de forma declarativa.

O mais importante é que você mudou sua forma de pensar sobre o processamento de dados. Em vez de dizer ao computador "como" fazer cada passo em um laço, você agora pode simplesmente declarar "o que" você quer como resultado.

Próximos Passos:

- Explore mais operações: Dê uma olhada em sorted(), distinct() e reduce().

- Optional: Aprenda sobre a classe Optional, grande aliada da programação funcional para tratar valores nulos de forma segura.

- Streams Paralelos: Investigue o .parallelStream() para processar grandes volumes de dados de forma ainda mais rápida em máquinas com múltiplos processadores.

Continue praticando. Pegue seus projetos antigos que usam laços for e tente refatorá-los com streams. A prática constante é o caminho para a maestria.

Boa codificação!

AGRADECIMENTOS

OBRIGADO POR LER ATÉ AQUI

Esse Ebook foi gerado por IA, e diagramado por humano.
O passo a passo se encontra no meu Github

.

Esse conteúdo foi gerado com fins didáticos de construção, não foi realizado uma validação cuidadosa humana no conteúdo e pode conter erros gerados por uma IA.

https://github.com/rogerio22mr/prompts-recipe-to-create-a-ebook