

# MODO DECLARATIVO: O CÓDIGO OCULTO DA MATRIX



# Java Moderno

## A Força da Programação Declarativa

A programação declarativa é um paradigma que se concentra em descrever o que deve ser feito, e não como deve ser feito.

Em vez de controlar o fluxo do programa com loops e condicionais, o desenvolvedor expressa intenções — e o Java, por meio de Streams, Lambdas e APIs funcionais, executa o processamento de forma eficiente e concisa. Esse estilo de programação se tornou essencial nas versões modernas do Java (a partir do Java 8), pois permite código mais legível, expressivo e fácil de manter.

# 01

## FILTRO

---

O método `filter()` é usado para selecionar elementos de uma coleção com base em uma condição lógica (predicado).

Ele funciona como um “filtro” que deixa passar apenas os elementos que atendem ao critério definido.

É muito útil quando se deseja trabalhar apenas com uma parte dos dados, sem modificar a coleção original.

# Refinando coleções com precisão

O `filter()` recebe como parâmetro um `Predicate<T>`, ou seja, uma função que testa uma condição e retorna um valor booleano (`true` ou `false`).

Durante a execução, cada elemento da `Stream` é avaliado individualmente.

Apenas os elementos que retornam `true` são mantidos e propagados para as próximas operações.

```
List<Integer> numeros = List.of(1, 5, 8, 12, 15);  
List<Integer> maioresQue10 = numeros.stream()  
    .filter(n -> n > 10)  
    .toList();  
  
System.out.println(maioresQue10); // [12, 15]
```

snappify.com



# 02

## MAPO

---

O método `map()` permite transformar os elementos de uma Stream aplicando uma função a cada um. Ele é uma das ferramentas mais poderosas do paradigma declarativo, pois permite converter tipos, formatar dados ou aplicar cálculos sem modificar a coleção original.

# Convertendo elementos em novas formas

O `map()` recebe uma `Function<T, R>` — uma função que define como cada elemento será transformado. Internamente, cada item da `Stream` é processado e convertido de acordo com a função fornecida, produzindo uma nova `Stream` com os resultados mapeados.

```
List<String> nomes = List.of("java", "kotlin", "python");
List<String> maiusculos = nomes.stream()
    .map(String::toUpperCase)
    .toList();

System.out.println(maiusculos); // [JAVA, KOTLIN, PYTHON]
```

snappify.com

# 03

## COLLECTO

---

Do fluxo ao destino final o método `collect()` transforma uma Stream em uma coleção ou outro tipo de resultado final.

Ele é a operação terminal mais poderosa e flexível das Streams, permitindo agregar, agrupar e resumir dados.

# Reunindo resultados em estruturas

O `collect()` utiliza a interface `Collector`, que define como os elementos serão acumulados. Com o utilitário `Collectors`, é possível criar listas, conjuntos, mapas, ou até estatísticas complexas.

```
List<String> nomes = List.of("Carlos", "Ana", "Beatriz");
List<String> ordenados = nomes.stream()
    .sorted()
    .toList();
System.out.println(ordenados); // [Ana, Beatriz, Carlos]
```

snappify.com

Com `Comparator` personalizado:

```
List<String> inverso = nomes.stream()
    .sorted(Comparator.reverseOrder())
    .toList();
System.out.println(inverso); // [Carlos, Beatriz, Ana]
```

snappify.com



# 04

## DISTINCTO

---

O método `distinct()` remove elementos duplicados de uma Stream, garantindo que cada item apareça apenas uma vez.

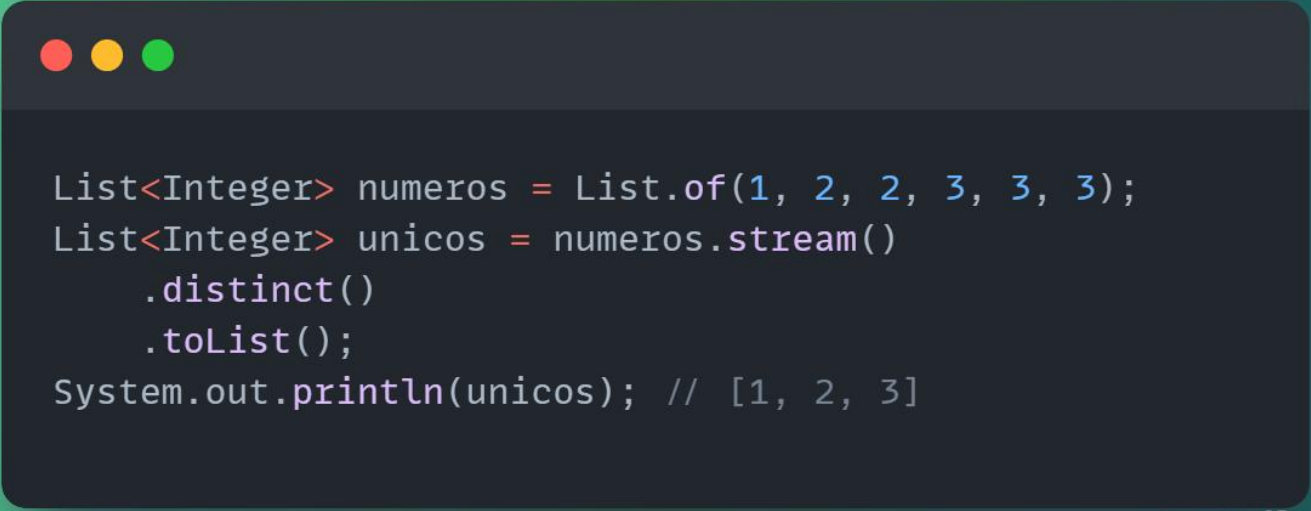
Isso é especialmente útil em cenários de filtragem de dados redundantes, como nomes repetidos ou valores idênticos.

# Eliminando repetições com estilo

O `distinct()` utiliza o método `equals()` e o `hashCode()` dos objetos para determinar a unicidade.

Durante o processamento, a `Stream` cria internamente um conjunto (semelhante a um `HashSet`) para verificar duplicidades.

Portanto, se você estiver trabalhando com objetos personalizados, é essencial sobrescrever `equals()` e `hashCode()` corretamente.



```
List<Integer> numeros = List.of(1, 2, 2, 3, 3, 3);
List<Integer> unicos = numeros.stream()
    .distinct()
    .toList();
System.out.println(unicos); // [1, 2, 3]
```

snappify.com

# 05

## REDUCEO

---

De muitos para um: somas, produtos e além

O método `reduce()` combina todos os elementos de uma Stream em um único resultado acumulado.

Ele é comumente usado para somar números, concatenar strings ou qualquer operação que combine elementos sequencialmente.

# A arte de combinar resultados

O `reduce()` recebe três variações principais:

Um valor inicial (identidade) e uma função acumuladora;

Somente uma função acumuladora, retornando um `Optional`;

Um valor inicial, acumulador e combinador (em `Streams` paralelas).

A função acumuladora é aplicada repetidamente, combinando o valor atual com o próximo elemento.

```
List<Integer> numeros = List.of(1, 2, 3, 4);
int soma = numeros.stream()
    .reduce(0, Integer::sum);
System.out.println(soma); // 10
```

snappify.com

Exemplo de concatenação:

```
List<String> nomes = List.of("Ana", "Bruno", "Carla");
String todos = nomes.stream()
    .reduce("", (a, b) → a + " " + b);
System.out.println(todos.trim()); // Ana Bruno Carla
```

# 06

## COLLECTO

---

O método `collect()` transforma uma Stream em uma coleção ou outro tipo de resultado final. Ele é a operação terminal mais poderosa e flexível das Streams, permitindo agregar, agrupar e resumir dados.



# Reunindo resultados em estruturas

O `collect()` utiliza a interface `Collector`, que define como os elementos serão acumulados. Com o utilitário `Collectors`, é possível criar listas, conjuntos, mapas, ou até estatísticas complexas.

```
List<String> nomes = List.of("Ana", "Bruno", "Carla");
Set<String> conjunto = nomes.stream()
    .collect(Collectors.toSet());
System.out.println(conjunto); // [Ana, Bruno, Carla]
```

snappify.com

Agrupando por tamanho do nome:

```
Map<Integer, List<String>> agrupados = nomes.stream()
    .collect(Collectors.groupingBy(String::length));
System.out.println(agrupados); // {3=[Ana], 5=[Bruno, Carla]}
```

snappify.com

# 07


# FOREACHO

---

O método `forEach()` é usado para percorrer uma Stream e executar uma ação para cada elemento. Ele é normalmente utilizado no fim da cadeia declarativa, quando se deseja exibir, registrar ou processar o resultado final.

# Executando ações finais

O `forEach()` recebe um `Consumer<T>`, ou seja, uma função que consome cada elemento sem retornar valor. Ele é uma operação terminal — ou seja, após sua execução, a `Stream` é encerrada.



```
List<String> nomes = List.of("Ana", "Bruno", "Carla");  
nomes.stream()  
    .forEach(System.out::println);
```

snappify.com

# 08

## FLATMAP

---

Transformando listas dentro de listas em um fluxo contínuo

O método `flatMap()` é essencial quando você trabalha com coleções aninhadas, como listas de listas.

Ele “achata” essas estruturas, transformando cada elemento interno em um fluxo único.

É muito usado para processar dados complexos de forma fluida e sem laços aninhados.

# Expandindo horizontes de dados

O `flatMap()` recebe uma função que transforma cada elemento da `Stream` em outra `Stream`. Esses fluxos internos são então mesclados em um único fluxo contínuo. Ou seja, em vez de gerar uma `Stream` de `Streams`, o resultado é uma única sequência linear de dados.

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

snappify.com

```
List<String> nomes = nomesAninhados.stream()
    .flatMap(List::stream)
    .toList();

System.out.println(nomes); // [Ana, Bruno, Carla, Daniel]
```

snappify.com



# 09

## PEEK

---

Observando o fluxo sem quebrar a cadeia  
O método `peek()` é usado para visualizar o conteúdo de uma Stream durante o processamento, sem modificá-lo. É útil para depuração, logs e análises intermediárias sem interromper a fluidez declarativa do código.

# Inspecione sem interferir

O `peek()` recebe um `Consumer<T>` — uma função que realiza uma ação com cada elemento, mas não altera o fluxo. Ele é uma operação intermediária, ou seja, não executa até que uma operação terminal seja chamada.

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

List<String> resultado = nomes.stream()
    .peek(n → System.out.println("Processando: " + n))
    .map(String::toUpperCase)
    .toList();

System.out.println(resultado); // [ANA, BRUNO, CARLA]
```

snappify.com

# 10

# LIMITO

---

Focando apenas no que é necessário

O método `limit()` é utilizado para restringir o número de elementos processados em uma Stream.

Ele é extremamente útil em situações onde se deseja uma amostra dos dados ou controle de performance em fluxos grandes.

# Controle o tamanho da saída

O `limit(n)` corta o fluxo após `n` elementos, ignorando os restantes.

A operação é curta-circuitada, ou seja, a Stream para de processar assim que o limite é atingido.

```
List<Integer> numeros = List.of(1, 2, 3, 4, 5, 6, 7, 8);  
List<Integer> primeirosTres = numeros.stream()  
    .limit(3)  
    .toList();  
  
System.out.println(primeirosTres); // [1, 2, 3]
```

snappify.com



# SKIPO

---

Pule o início e vá direto ao ponto

O método `skip()` permite ignorar os primeiros elementos de uma `Stream` e continuar a partir de um determinado ponto.

Combinado com `limit()`, ele é excelente para implementar paginação.



# Ignorando o que não importa

O `skip(n)` descarta os `n` primeiros elementos do fluxo e processa o restante normalmente.

Ele também é uma operação curta-circuitada, o que melhora a eficiência em grandes coleções.

```
List<Integer> numeros = List.of(1, 2, 3, 4, 5, 6, 7);  
List<Integer> depoisDosTres = numeros.stream()  
    .skip(3)  
    .toList();
```

```
System.out.println(depoisDosTres); // [4, 5, 6, 7]
```

snappify.com

# 12

## ANYSMATCHO

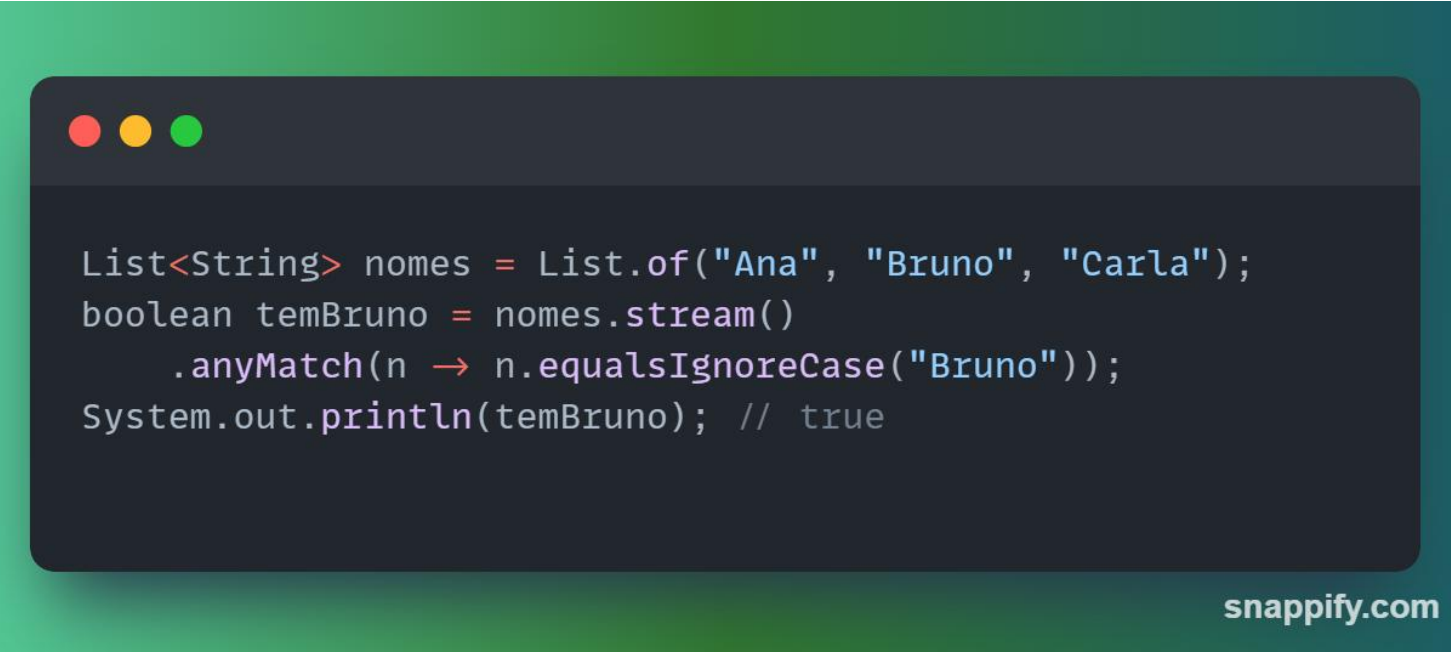
---

Buscando a primeira correspondência verdadeira  
O método `anyMatch()` verifica se pelo menos um elemento da Stream atende a uma condição específica. Ele é ideal para validações rápidas e verificações booleanas.

# Existe pelo menos um que satisfaça?

O `anyMatch()` recebe um `Predicate<T>` e retorna `true` assim que encontra o primeiro elemento que satisfaça a condição.

Por isso, ele é curto-circuitado, interrompendo o fluxo imediatamente quando encontra um resultado positivo.



```
List<String> nomes = List.of("Ana", "Bruno", "Carla");
boolean temBruno = nomes.stream()
    .anyMatch(n -> n.equalsIgnoreCase("Bruno"));
System.out.println(temBruno); // true
```

snappify.com

# 13

## ALLMATCHO

---

Verificando conformidade total

O método `allMatch()` retorna `true` se todos os elementos de uma `Stream` satisfizerem a condição especificada. É amplamente usado para validar listas inteiras.

# Todos atendem à condição?

Assim como `anyMatch()`, ele recebe um `Predicate<T>` e percorre os elementos até encontrar o primeiro que não atenda à condição.

Se todos passarem, o resultado final é `true`.

```
List<Integer> numeros = List.of(2, 4, 6, 8);  
boolean todosPares = numeros.stream()  
    .allMatch(n -> n % 2 == 0);  
System.out.println(todosPares); // true
```

snappify.com



# 14

## NONEMATCHO

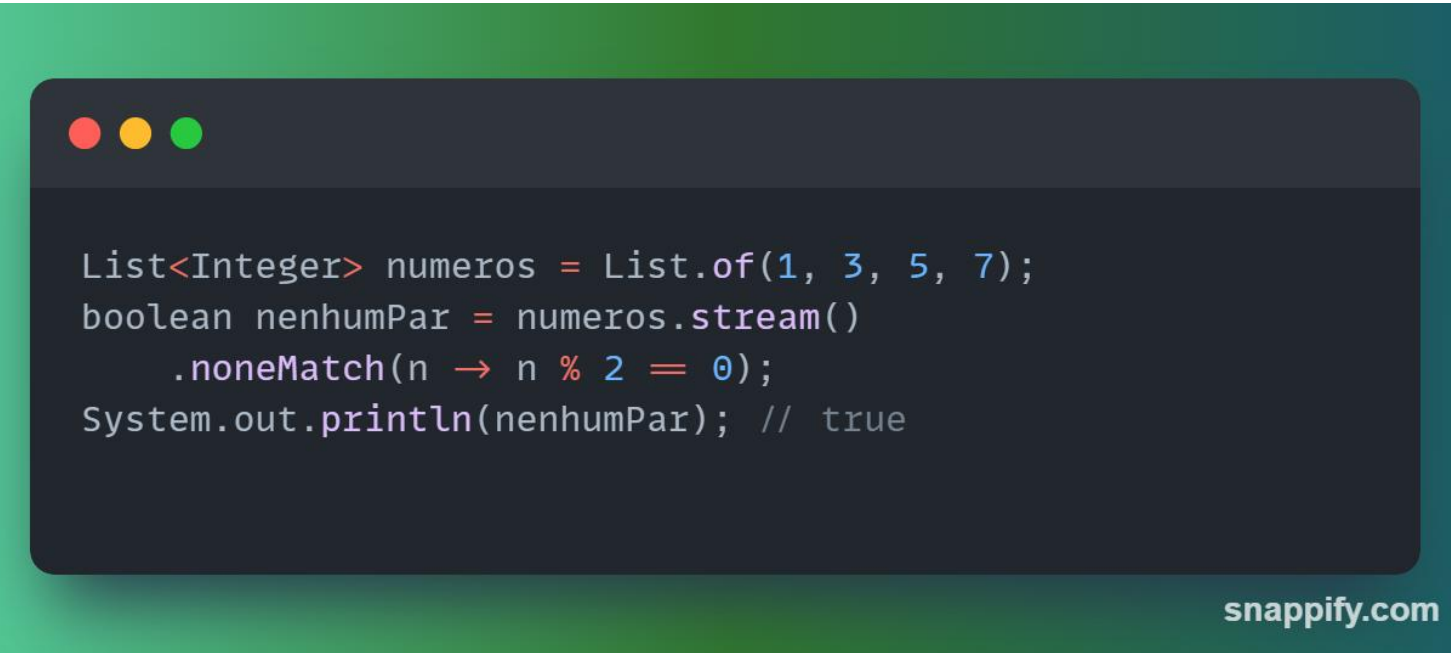
---

A certeza da ausência

O método `noneMatch()` faz o oposto de `anyMatch()`. Ele retorna `true` se nenhum elemento da Stream atender à condição.

# Nenhum corresponde à condição

O `noneMatch()` também recebe um `Predicate<T>` e para a execução assim que encontra o primeiro elemento que viola a regra. É muito útil para garantir que uma coleção está livre de valores indesejados.



```
List<Integer> numeros = List.of(1, 3, 5, 7);  
boolean nenhumPar = numeros.stream()  
    .noneMatch(n → n % 2 == 0);  
System.out.println(nenhumPar); // true
```

snappify.com

# 15

## FINDFIRST

---

Recuperando o primeiro elemento válido

O método `findFirst()` retorna o primeiro elemento presente na Stream, ou o primeiro que aparece após as operações intermediárias.

É comum em filtros e buscas.

# O primeiro da fila

Ele retorna um `Optional<T>` — o que significa que o valor pode estar presente ou não. Essa abordagem evita `NullPointerException` e promove o uso seguro de valores opcionais.

```
List<String> nomes = List.of("Ana", "Bruno", "Carla");
Optional<String> primeiro = nomes.stream()
    .filter(n -> n.startsWith("B"))
    .findFirst();

primeiro.ifPresent(System.out::println); // Bruno
```

snappify.com

# 16

## FINDANYO

---

A busca mais rápida em fluxos paralelos

O método `findAny()` retorna qualquer elemento da Stream.

Ele é especialmente eficiente em Streams paralelas, pois retorna o primeiro resultado encontrado em qualquer thread.

# Qualquer um serve

Assim como `findFirst()`, o retorno é um `Optional<T>`. Em Streams sequenciais, normalmente retorna o primeiro elemento, mas em paralelas o resultado pode variar.

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

qualquer.ifPresent(System.out::println);
```

snappify.com



# 17

## COUNT

---

Quantifique seus resultados sem esforço

O método `count()` retorna o número total de elementos de uma Stream.

Ele é uma das operações terminais mais diretas, útil para relatórios, estatísticas e verificações de tamanho.

# Contando elementos com precisão

O `count()` percorre toda a Stream (ou o que restar após filtros) e retorna um `long` com o total de elementos processados.

```
List<Integer> numeros = List.of(1, 2, 3, 4, 5);  
long quantidade = numeros.stream()  
    .filter(n -> n > 2)  
    .count();  
  
System.out.println(quantidade); // 3
```

snappify.com

# Agradecimento



# Obrigado por ler até aqui

Esse Ebook foi gerado por IA, e diagramado por humano.

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



[LeandroMeca/ebook-ia](https://github.com/LeandroMeca/ebook-ia)