

# Java: Evolução e Inovações (Versões 8 a 21)

## Das Expressões Lambda às Funcionalidades Mais Recentes

Com **exemplos** práticos e didáticos, cuidadosamente comentados e disponíveis no [GitHub](#)



Luís Simões da Cunha, PhD

CC BY-NC 4.0

Edição de 2025



Este trabalho está licenciado sob a Licença Creative Commons Atribuição-NãoComercial 4.0 Internacional (CC BY-NC 4.0).

Você é livre para copiar, distribuir e modificar este documento, desde que:

1. Atribua a autoria original a "Luís Simões da Cunha".
2. Não o utilize para fins comerciais.

Para mais informações sobre esta licença, visite: <https://creativecommons.org/licenses/by-nc/4.0/>

Link para Repositório de [Exemplos no GitHub](#)

(por favor, note que os exemplos são semelhantes, mas não necessariamente idênticos aos apresentados no texto)

# Índice

<b>Introdução .....</b>	<b>1</b>
<b>Porquê Este Manual? .....</b>	<b>1</b>
<b>1. Expressões Lambda no Java .....</b>	<b>2</b>
<b>Estrutura de uma Expressão Lambda.....</b>	<b>2</b>
<b>Porque usar Expressões Lambda? .....</b>	<b>3</b>
<b>Requisitos para Usar Expressões Lambda .....</b>	<b>3</b>
<b>Exemplos de Uso de Expressões Lambda .....</b>	<b>4</b>
<b>Passos para Entender e Usar Lambdas.....</b>	<b>6</b>
<b>Vantagens das Lambdas.....</b>	<b>6</b>
<b>Exemplo Prático Completo .....</b>	<b>7</b>
<b>Resumo.....</b>	<b>7</b>
Exemplos completos de <b>Expressões Lambda</b> em Java: .....	8
<b>Exemplo 1: O que é uma Expressão Lambda? .....</b>	<b>8</b>
<b>Exemplo 2: Lambda com Parâmetros .....</b>	<b>9</b>
<b>Exemplo 3: Lambda com Bloco de Código .....</b>	<b>10</b>
<b>Exemplo 4: Uso de Lambda com Predicate .....</b>	<b>11</b>
<b>Exemplo 5: Lambda com Referência a Métodos.....</b>	<b>12</b>
<b>Resumo.....</b>	<b>12</b>
<b>2. O que é a API de Streams no Java? .....</b>	<b>13</b>
<b>Conceitos Fundamentais .....</b>	<b>13</b>
<b>Porque Usar a API de Streams? .....</b>	<b>13</b>
<b>Exemplo Básico: Como Funciona a API de Streams .....</b>	<b>14</b>
<b>Tipos de Operações em Streams .....</b>	<b>15</b>
<b>Streams Paralelos.....</b>	<b>16</b>
<b>Exemplo Prático: Processar uma Lista de Palavras .....</b>	<b>16</b>
<b>Resumo.....</b>	<b>17</b>
Exemplos completos da <b>API de Streams</b> : .....	18
<b>Exemplo 1: Introdução à API de Streams .....</b>	<b>18</b>
<b>Exemplo 2: Operações Intermediárias.....</b>	<b>19</b>
<b>Exemplo 3: Operações Terminais .....</b>	<b>20</b>
<b>Exemplo 4: Operação reduce .....</b>	<b>21</b>
<b>Exemplo 5: Uso de collect para Criar Coleções .....</b>	<b>22</b>
<b>Exemplo 6: Stream Paralelo.....</b>	<b>23</b>
<b>Resumo.....</b>	<b>24</b>

Tema Suplementar (sobre Streams): Combinação map e reduce .....	25
Como Funciona a Combinação map e reduce? .....	25
Exemplo 1: Soma de Quadrados .....	25
Exemplo 2: Cálculo da Receita Total .....	26
Exemplo 3: Contagem de Palavras em Textos .....	27
Benefícios da Combinação map e reduce .....	27
Resumo .....	28
<b>3. O que são Referências a Métodos e Construtores no Java? .....</b>	<b>29</b>
Tipos de Referências .....	29
Exemplos de Cada Tipo .....	30
Vantagens das Referências a Métodos e Construtores .....	34
Resumo Comparativo .....	34
Dicas Práticas .....	34
Exemplos completos de Referências a Métodos e Construtores: .....	35
Exemplo 1: Referência a um Método Estático .....	35
Exemplo 2: Referência a um Método de Instância de um Objeto Específico .....	36
Exemplo 3: Referência a um Método de Instância de um Objeto Arbitrário .....	37
Exemplo 4: Referência a um Construtor .....	38
Exemplo 5: Comparação entre Referências e Lambdas .....	39
Resumo dos Tipos de Referências .....	40
Dicas Práticas .....	40
<b>4. Classe Optional .....</b>	<b>41</b>
O que é a classe Optional no Java? .....	41
Porque Usar Optional? .....	41
Como Criar um Optional? .....	42
Métodos Comuns do Optional .....	43
Exemplo Prático: Uso de Optional .....	48
Resumo: Quando Usar Optional? .....	49
Exemplos completos sobre o conceito de Optional: .....	50
Exemplo 1: Criação de um Optional .....	50
Exemplo 2: Evitar NullPointerException .....	51
Exemplo 3: Fornecer um Valor Padrão .....	52
Exemplo 4: Transformar Valores com map .....	53
Exemplo 5: Combinar Optional com flatMap .....	54
Exemplo 6: Lançar Exceções com orElseThrow .....	55
Resumo .....	56
<b>5. Melhorias na API de Coleções no Java 8 .....</b>	<b>57</b>

1. Integração com a API de Streams .....	57
2. Novos Métodos em Coleções .....	58
3. Novos Coletores com Collectors .....	61
4. Criação de Coleções Imutáveis .....	62
Resumo das Melhorias .....	63
Exemplos completos sobre melhorias na <b>API de Coleções</b> no Java 8:.....	64
Exemplo 1: Iteração Melhorada com <code>forEach</code> .....	64
Exemplo 2: Remover Elementos com <code>removeIf</code> .....	65
Exemplo 3: Transformar Elementos com <code>replaceAll</code> .....	66
Exemplo 4: Manipulação de Mapas com <code>computeIfAbsent</code> .....	67
Exemplo 5: Integração com Streams .....	68
Exemplo 6: Criação de Coleções Imutáveis .....	69
Resumo das Melhorias .....	70
Conclusão .....	70
6. Sobre o <i>warning</i> relacionado com (não) fecho de Scanner.....	71
Casos em que é necessário fechar o Scanner .....	72
Casos em que o Scanner não precisa de ser fechado explicitamente.....	74
Melhoria no Fechamento com <code>try-with-resources</code> .....	76
Resumo sobre Fecho de Scanner: .....	77
Boa Prática .....	77
7. Notas sobre o operador <code>&lt; &gt;</code> (" <i>diamond</i> " ou "diamante") .....	78
O que é o operador <code>&lt; &gt;</code> no Java?.....	78
Como o Diamond Funciona? .....	78
Casos de Uso Comuns do Operador <code>&lt; &gt;</code> .....	79
Restrições do Diamond .....	80
Melhorias no Java 9 e Posteriores .....	81
Vantagens do Diamond .....	82
Resumo .....	82
8. Java 9 a Java 16: Principais Melhorias .....	83
Java 9 .....	83
Java 10 .....	85
Java 11 .....	85
Java 12 .....	86
Java 14 .....	86
Java 15 .....	86
Java 16 .....	87
Resumo das melhorias da versão 9 à 16: .....	88

<b>9. Melhorias das Versões 17 a 21 da Linguagem Java:</b>	<b>89</b>
Java 17	89
Java 18	91
Java 19	92
Java 20	93
Java 21	94
Resumo das Funcionalidades das Versão 17 a 21:	95
Funcionalidades Estabilizadas a Partir do Java 17:	96
1. Classes Seladas (Sealed Classes)	96
2. Correspondência de Padrões para switch (Pattern Matching for switch)	97
3. Blocos de Texto (Text Blocks)	98
4. Records	99
5. Inferência de Tipos com var	100
6. API de Cliente HTTP	101
7. API de Funções Estrangeiras e Memória (Foreign Function & Memory API)	102
<b>10. Inferência de Tipos com var</b>	<b>103</b>
Exemplos Práticos do Uso de var	103
Considerações Importantes sobre uso de var	105
Vantagens do Uso de var	106
Limitações do uso de var	106
Conclusão	106

## Introdução

Desde o seu surgimento em 1995, a linguagem de programação Java tem desempenhado um papel central no mundo do desenvolvimento de software. Concebida com os objetivos de portabilidade, segurança e simplicidade, o Java rapidamente conquistou uma posição de destaque, sendo amplamente adotada por empresas, instituições acadêmicas e comunidades de programadores. No entanto, é a sua capacidade de evolução e adaptação às novas exigências tecnológicas que a mantém tão relevante décadas após a sua criação.

Este manual é uma celebração dessa evolução, focando-se nas inovações introduzidas desde o Java 8 até ao Java 21. Cada nova versão trouxe consigo funcionalidades que não apenas expandiram as capacidades da linguagem, mas também transformaram a maneira como escrevemos e pensamos sobre código Java. Expressões Lambda, Streams, Classes Optional, referências a métodos e construtores são apenas algumas das ferramentas que redefiniram o paradigma do desenvolvimento em Java, aproximando-o da programação funcional. Mais recentemente, funcionalidades como classes seladas, correspondência de padrões para switch e blocos de texto demonstram que o Java continua a surpreender e inovar.

A motivação para este manual é simples: ajudar programadores que já dominam os fundamentos da linguagem a explorar estas inovações, tornando-as acessíveis através de explicações claras e exemplos práticos. Muitas vezes, quem conhece a sintaxe base de Java não teve oportunidade de acompanhar as evoluções mais recentes, perdendo a oportunidade de tirar partido de construtos que podem tornar o seu código mais expressivo, eficiente e fácil de manter.

Os exemplos fornecidos ao longo do manual foram desenvolvidos com cuidado para serem didáticos, objetivos e diretamente aplicáveis em situações reais. Exemplos semelhantes estão disponíveis [num repositório no GitHub](#), [acessíveis através desta hiperligação](#), permitindo ao leitor experimentar e adaptar o código ao seu próprio contexto.

## Porquê Este Manual?

A linguagem Java combina tradição e inovação, sendo frequentemente usada tanto em aplicações de missão crítica como em projetos pessoais. Contudo, com a constante evolução da linguagem, muitos programadores podem sentir-se desatualizados ou inseguros em relação às novas funcionalidades. Este manual visa preencher essa lacuna, apresentando não apenas as novidades, mas também o contexto e a motivação subjacentes a cada melhoria.

Cada capítulo foi pensado para ser informativo e prático, começando com conceitos introdutórios e evoluindo para exemplos mais avançados. Quer o leitor esteja a descobrir Expressões Lambda pela primeira vez ou queira entender as melhorias introduzidas nas versões mais recentes, encontrará aqui uma abordagem estruturada e acessível.

Acima de tudo, este manual convida-o/a a explorar a linguagem Java de uma forma prática e moderna, reforçando a sua capacidade de desenvolver soluções elegantes e eficazes. Vamos embarcar juntos nesta jornada de descoberta e domínio da evolução do Java!

# 1. Expressões Lambda no Java

As **Expressões Lambda**, introduzidas no **Java 8**, são uma forma concisa de representar métodos anónimos, ou seja, funções que podem ser usadas como valores. Elas permitem que o Java suporte **programação funcional**, tornando o código mais claro e menos verboso.

---

## Estrutura de uma Expressão Lambda

Uma expressão lambda tem a seguinte sintaxe básica:

(parametros) -> { corpo da função }

*Componentes:*

### 1. Parâmetros:

- Representam os argumentos que a função recebe.
- Podem ser omitidos os parênteses se houver apenas um parâmetro.
- Exemplo:  $x \rightarrow x * x$  (recebe um número  $x$  e devolve o seu quadrado).

### 2. Seta (->):

- Separa os parâmetros do corpo da função.

### 3. Corpo da Função:

- Contém a lógica que será executada.
- Pode ser uma única expressão ou um bloco de código (entre { }).

*Exemplos:*

- **Lambda de uma única linha:**

(a, b) -> a + b

Soma dois números.

- **Lambda com bloco de código:**

```
(a, b) -> {  
    int resultado = a * b;  
    return resultado;  
}
```



---

## Porque usar Expressões Lambda?

1. **Redução de Verbosidade:**
    - Substituem classes anónimas para implementar interfaces funcionais, simplificando o código.
  2. **Aproximação à Programação Funcional:**
    - Permitem tratar comportamentos como valores, passá-los como argumentos ou retorná-los de métodos.
  3. **Melhor Legibilidade:**
    - Torna o código mais limpo e intuitivo.
- 

## Requisitos para Usar Expressões Lambda

1. **Interface Funcional:**
  - As lambdas só podem ser usadas onde se espera uma **interface funcional** (interface com apenas um método abstrato).
  - Exemplos de interfaces funcionais no Java:
    - `Runnable` (do pacote `java.lang`).
    - `Comparator<T>` (do pacote `java.util`).
    - Interfaces no pacote `java.util.function` como `Predicate`, `Function`, `Consumer`, etc.

---

## Exemplos de Uso de Expressões Lambda

### 1. *Uso com Runnable*

Antes do Java 8 (com classe anônima):

```
Runnable tarefa = new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("Tarefa em execução!");  
    }  
};  
new Thread(tarefa).start();
```

Com Lambda:

```
Runnable tarefa = () -> System.out.println("Tarefa em execução!");  
new Thread(tarefa).start();
```

---

### 2. *Uso com Comparator*

Antes do Java 8 (com classe anônima):

```
Comparator<Integer> comparador = new Comparator<Integer>() {  
    @Override  
    public int compare(Integer a, Integer b) {  
        return a - b;  
    }  
};
```

Com Lambda:

```
Comparator<Integer> comparador = (a, b) -> a - b;
```

---

### 3. *Uso com Predicate*

O `Predicate<T>` é uma interface funcional que avalia uma condição e retorna `true` ou `false`.

#### **Exemplo:**

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

// Lambda para verificar se o número é par
Predicate<Integer> isPar = num -> num % 2 == 0;

// Filtra números pares
numeros.stream()
    .filter(isPar)
    .forEach(System.out::println); // Imprime: 2, 4
```

---

### 4. *Uso com Function*

O `Function<T, R>` é uma interface funcional que transforma um valor de tipo `T` em um de tipo `R`.

#### **Exemplo:**

```
Function<String, Integer> tamanho = str -> str.length();
System.out.println(tamanho.apply("Java")); // Imprime: 4
```

---

## Passos para Entender e Usar Lambdas

### 1. Identificar uma Interface Funcional:

- Certifique-se de que o contexto suporta uma interface funcional.
- Exemplo: Comparator, Predicate, Runnable, ou qualquer interface funcional personalizada.

### 2. Definir o Comportamento:

- Use a expressão lambda para definir o comportamento desejado.

### 3. Utilizar a Lambda:

- Atribua a lambda a uma variável, passe-a como argumento ou use-a diretamente.
- 

## Vantagens das Lambdas

### 1. Código Mais Conciso:

- Substituem classes anónimas extensas com poucas linhas de código.

### 2. Melhor Legibilidade:

- O foco é no comportamento, não na estrutura da classe.

### 3. Combinação com Streams:

- Funcionam de forma excelente com a API de Streams, permitindo operações como mapear, filtrar e reduzir coleções de forma declarativa.

---

## Exemplo Prático Completo

Este exemplo combina o uso de Streams e Expressões Lambda para processar uma lista de nomes:

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

public class ExemploLambda {
    public static void main(String[] args) {
        // Lista de nomes
        List<String> nomes = Arrays.asList("Ana", "João", "Maria", "Pedro");

        // Filtrar nomes que começam com "A" e transformar em maiúsculas
        nomes.stream()
            .filter(nome -> nome.startsWith("A")) // Filtra nomes que começam com "A"
            .map(nome -> nome.toUpperCase())       // Converte para maiúsculas
            .forEach(System.out::println);         // Imprime o resultado
    }
}
```

**Saída:**

ANA

---

## Resumo

1. **Expressões Lambda** são métodos anónimos que tornam o código mais conciso e funcional.
2. São usadas em **interfaces funcionais** (uma interface com um único método abstrato).
3. Facilitam a programação funcional e o processamento declarativo de dados, especialmente em combinação com **Streams**.

## Exemplos completos de **Expressões Lambda** em Java:

---

### Exemplo 1: O que é uma Expressão Lambda?

Este exemplo mostra como usar uma expressão lambda para substituir uma classe anônima.

```
// Exemplo básico com Runnable
public class ExemploLambda1 {
    public static void main(String[] args) {
        // Antes do Java 8: Classe anônima
        Runnable tarefaAntes = new Runnable() {
            @Override
            public void run() {
                System.out.println("Tarefa em execução (antes do Java 8)!");
            }
        };
        tarefaAntes.run(); // Executa a tarefa
        // Com Java 8: Expressão Lambda
        Runnable tarefaLambda = () -> System.out.println("Tarefa em execução (com lambda)!");
        tarefaLambda.run(); // Executa a tarefa
    }
}
```

#### Comentários:

- A expressão lambda `() -> System.out.println("Tarefa em execução (com lambda)!")` substitui a implementação de uma classe anônima para a interface funcional `Runnable`.
- A seta `->` separa os **parâmetros** (entre parênteses) do **corpo da função**.
- Neste caso, não há parâmetros, por isso os parênteses estão vazios.

---

## Exemplo 2: Lambda com Parâmetros

Uma expressão lambda pode receber parâmetros. Vamos criar um exemplo simples de uma operação matemática.

```
import java.util.function.BiFunction;
public class ExemploLambda2 {
    public static void main(String[] args) {
        // Lambda para somar dois números
        BiFunction<Integer, Integer, Integer> somar = (a, b) -> a + b;
        // Lambda para multiplicar dois números
        BiFunction<Integer, Integer, Integer> multiplicar = (a, b) -> a * b;
        // Testar as lambdas
        System.out.println("Soma: " + somar.apply(5, 3)); // Imprime: Soma: 8
        System.out.println("Multiplicação: " + multiplicar.apply(5, 3)); // Imprime:
Multiplicação: 15
    }
}
```

### Comentários:

- O **BiFunction<T, U, R>** é uma interface funcional onde:
  - T e U são os tipos dos dois parâmetros da função.
  - R é o tipo do valor de retorno.
- As lambdas **somar** e **multiplicar** implementam o método **apply(T t, U u)** da interface **BiFunction**.

---

### Exemplo 3: Lambda com Bloco de Código

Quando o corpo da função é mais complexo, pode-se usar um bloco de código com { }.

```
import java.util.function.Function;
public class ExemploLambda3 {
    public static void main(String[] args) {
        // Lambda com bloco de código para calcular o fatorial de um número
        Function<Integer, Integer> fatorial = n -> {
            int resultado = 1;
            for (int i = 1; i <= n; i++) {
                resultado *= i; // Multiplica os números de 1 a n
            }
            return resultado; // Devolve o fatorial calculado
        };
        // Testar a lambda
        System.out.println("Fatorial de 5: " + fatorial.apply(5)); // Imprime: Fatorial
de 5: 120
    }
}
```

#### Comentários:

- As { } são usadas para escrever um bloco de código completo.
- A palavra-chave return é necessária quando o corpo da função tem múltiplas linhas.



---

## Exemplo 4: Uso de Lambda com Predicate

O **Predicate<T>** é uma interface funcional que avalia uma condição e retorna um boolean.

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;
public class ExemploLambda4 {
    public static void main(String[] args) {
        // Lista de números
        List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);
        // Lambda para verificar se o número é par
        Predicate<Integer> isPar = num -> num % 2 == 0;
        // Filtrar e imprimir apenas os números pares
        System.out.println("Números pares:");
        numeros.stream() // Cria um Stream a partir da lista
                .filter(isPar) // Aplica o filtro (lambda)
                .forEach(System.out::println); // Imprime cada número par
    }
}
```

### Comentários:

- A lambda `num -> num % 2 == 0` implementa o método `test(T t)` da interface `Predicate`.
- O `Stream` processa a lista e filtra apenas os números que passam no teste.

---

## Exemplo 5: Lambda com Referência a Métodos

As lambdas podem ser substituídas por **referências a métodos**, quando a lógica da função já existe como um método.

```
import java.util.Arrays;
import java.util.List;
public class ExemploLambda5 {
    public static void main(String[] args) {
        // Lista de nomes
        List<String> nomes = Arrays.asList("Ana", "João", "Maria");
        // Usar referência a método para imprimir os nomes
        System.out.println("Nomes:");
        nomes.forEach(System.out::println); // Referência ao método println
    }
}
```

### Comentários:

- `System.out::println` é uma referência ao método `println` da classe `System.out`.
- Substitui a lambda `(nome) -> System.out.println(nome)`.

---

## Resumo

### 1. O que são lambdas?

- Uma forma concisa de implementar interfaces funcionais, reduzindo a verbosidade.
- Podem substituir classes anônimas em muitos casos.

### 2. Sintaxe:

- `(parametros) -> { corpo da função }`
- Se o corpo for uma única linha, as `{ }` e o `return` podem ser omitidos.

### 3. Casos de Uso:

- Uso com **interfaces funcionais** como `Runnable`, `Comparator`, `Predicate`, `Function`, etc.
- Operações com **Streams** para filtrar, mapear e processar coleções.

## 2. O que é a API de Streams no Java?

A **API de Streams**, introduzida no **Java 8**, é uma ferramenta poderosa para processar coleções de dados de forma **declarativa**, ou seja, especificando *o que fazer* (e não *como fazer*). É amplamente utilizada para realizar operações como **filtrar**, **mapear**, **ordenar** e **reduzir** coleções ou outros conjuntos de dados de forma eficiente e legível.

---

### Conceitos Fundamentais

#### 1. Stream:

- Um fluxo sequencial de dados que suporta operações como filtro, transformação e agregação.
- Não é uma estrutura de dados; funciona como uma **visão** sobre a coleção.
- As operações num Stream não alteram os dados da coleção original.

#### 2. Pipeline de Operações:

- Um **Stream** é processado através de uma sequência de operações:
    - **Intermediárias**: Transformam o Stream (e.g., `filter`, `map`).
    - **Terminais**: Consomem o Stream e produzem um resultado (e.g., `forEach`, `collect`).
- 

### Porque Usar a API de Streams?

#### 1. Código Declarativo:

- Substitui loops e condições com operações simples e legíveis.
- Exemplo: Filtrar números pares de uma lista.

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

#### 2. Processamento Eficiente:

- Suporta **processamento paralelo** com o uso de `parallelStream()`, otimizando o desempenho em CPUs multi-core.

#### 3. Imutabilidade:

- O Stream não altera a coleção original, garantindo a segurança dos dados.

---

## Exemplo Básico: Como Funciona a API de Streams

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

public class ExemploStream {
    public static void main(String[] args) {
        // Lista de números
        List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);

        // Filtrar números pares, calcular o quadrado e coletar os resultados numa nova lista
        List<Integer> quadradosPares = numeros.stream() // Cria um Stream a partir da lista
            .filter(n -> n % 2 == 0) // Filtra números pares
            .map(n -> n * n) // Calcula o quadrado de cada número
            .collect(Collectors.toList()); // Coleta o resultado numa nova lista

        // Imprime o resultado
        System.out.println(quadradosPares); // Saída: [4, 16]
    }
}
```

### Explicação:

1. **stream()**: Cria um Stream a partir da lista de números.
2. **filter**: Operação intermediária que filtra números pares.
3. **map**: Operação intermediária que transforma cada número no seu quadrado.
4. **collect**: Operação terminal que coleta os resultados numa lista.

---

## Tipos de Operações em Streams

### 1. Operações Intermediárias

- Transformam ou filtram dados no Stream, mas não o consomem.
- São **preguiçosas**: só são executadas quando uma operação terminal é chamada.

Operação	Descrição
<b>filter</b>	Filtra elementos com base numa condição (retorna um subconjunto).
<b>map</b>	Transforma os elementos (e.g., converte de um tipo para outro).
<b>distinct</b>	Remove elementos duplicados.
<b>sorted</b>	Ordena os elementos de forma natural ou com base num comparador.
<b>limit</b>	Limita o número de elementos no Stream.
<b>skip</b>	Ignora os primeiros n elementos no Stream.

#### Exemplo:

```
List<String> nomes = Arrays.asList("Ana", "João", "Maria", "Ana");
List<String> nomesUnicos = nomes.stream()
    .distinct() // Remove duplicados
    .sorted()   // Ordena alfabeticamente
    .collect(Collectors.toList());
System.out.println(nomesUnicos); // Saída: [Ana, João, Maria]
```

---

### 2. Operações Terminais

- Consomem o Stream, encerrando o pipeline de operações.

Operação	Descrição
<b>forEach</b>	Executa uma ação para cada elemento.
<b>collect</b>	Coleta os resultados num List, Set, ou outra estrutura de dados.
<b>reduce</b>	Combina os elementos para produzir um único valor.
<b>count</b>	Conta o número de elementos no Stream.
<b>findFirst</b>	Devolve o primeiro elemento no Stream (opcional).

#### Exemplo:

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);
int soma = numeros.stream()
    .reduce(0, Integer::sum); // Soma todos os números
System.out.println("Soma: " + soma); // Saída: Soma: 15
```

---

## Streams Paralelos

A API de Streams suporta processamento paralelo, distribuindo as tarefas entre múltiplos núcleos da CPU.

### Exemplo:

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Processamento paralelo para calcular o dobro de cada número
numeros.parallelStream()
    .map(n -> n * 2)
    .forEach(System.out::println);
```

### Nota:

- Os Streams paralelos podem melhorar o desempenho em grandes volumes de dados, mas não garantem a ordem de execução.

---

## Exemplo Prático: Processar uma Lista de Palavras

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

public class ExemploStreamCompleto {
    public static void main(String[] args) {
        // Lista de palavras
        List<String> palavras = Arrays.asList("Java", "Stream", "API", "Lambda", "Java");

        // Processar a lista: remover duplicados, converter para maiúsculas e ordenar
        List<String> resultado = palavras.stream()
            .distinct()                // Remove duplicados
            .map(String::toUpperCase) // Converte para maiúsculas
            .sorted()                  // Ordena alfabeticamente
            .collect(Collectors.toList()); // Coleta o resultado

        // Imprime o resultado
        System.out.println(resultado); // Saída: [API, JAVA, LAMBDA, STREAM]
    }
}
```

---

## Resumo

### 1. O que é a API de Streams?

- A API de Streams processa coleções de dados de forma declarativa, facilitando operações como filtragem, transformação e agregação.

### 2. Como funciona?

- Baseia-se em **pipelines de operações**:
  - Operações **intermediárias** (e.g., `filter`, `map`) para transformar o Stream.
  - Operações **terminais** (e.g., `forEach`, `collect`) para consumir os dados.

### 3. Benefícios:

- Código mais legível e expressivo.
- Processamento eficiente, com suporte a **paralelismo**.

### 4. Exemplo Real:

- Processar listas, transformar dados, filtrar resultados, etc.

## Exemplos completos da **API de Streams**:

---

### Exemplo 1: Introdução à API de Streams

A API de Streams permite trabalhar de forma declarativa com coleções de dados. Neste exemplo, vamos usar um **Stream** para filtrar, transformar e processar dados.

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

public class IntroducaoStream {
    public static void main(String[] args) {
        // Lista de números
        List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6);

        // Filtrar números pares, calcular o quadrado de cada número e coletar os resultados numa nova lista
        List<Integer> quadradosPares = numeros.stream()           // Cria um Stream a partir da lista
            .filter(n -> n % 2 == 0)                             // Filtra números pares
            .map(n -> n * n)                                       // Calcula o quadrado de cada número
            .collect(Collectors.toList());                         // Coleta o resultado numa nova lista

        // Imprime os resultados
        System.out.println("Quadrados dos números pares: " + quadradosPares); // Saída: [4, 16, 36]
    }
}
```

#### Explicação:

1. **stream()**: Cria um Stream a partir da lista.
2. **filter**: Filtra os números pares (`n -> n % 2 == 0`).
3. **map**: Transforma cada número no seu quadrado.
4. **collect**: Coleta os resultados numa nova lista.



---

## Exemplo 2: Operações Intermediárias

As **operações intermediárias** transformam ou filtram dados num Stream e são "preguiçosas", ou seja, só são executadas quando existe uma operação terminal.

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

public class OperacoesIntermediarias {
    public static void main(String[] args) {
        // Lista de nomes
        List<String> nomes = Arrays.asList("Ana", "João", "Pedro", "Ana", "Maria");

        // Transformar nomes: remover duplicados, converter para maiúsculas e ordenar
        nomes.stream()
            .distinct()           // Remove duplicados
            .map(String::toUpperCase) // Converte para maiúsculas
            .sorted()             // Ordena alfabeticamente
            .forEach(System.out::println); // Imprime cada nome transformado
    }
}
```

### Saída:

```
ANA
JOÃO
MARIA
PEDRO
```

### Explicação:

- **distinct()**: Remove duplicados.
- **map(String::toUpperCase)**: Converte cada elemento para maiúsculas.
- **sorted()**: Ordena os elementos.

---

### Exemplo 3: Operações Terminais

As **operações terminais** encerram o Stream e produzem um resultado ou efeito colateral.

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

public class OperacoesTerminais {
    public static void main(String[] args) {
        // Lista de números
        List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);

        // Contar números pares
        long countPares = numeros.stream()
            .filter(n -> n % 2 == 0) // Filtra números pares
            .count();                // Conta o número de elementos no Stream

        System.out.println("Números pares: " + countPares); // Saída: Números pares: 2
    }
}
```

#### Explicação:

- **filter**: Filtra os números pares.
- **count**: Operação terminal que devolve o número de elementos.

---

## Exemplo 4: Operação reduce

A operação reduce combina os elementos do Stream num único valor.

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

public class ExemploReduce {
    public static void main(String[] args) {
        // Lista de números
        List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);

        // Soma de todos os números
        int soma = numeros.stream()
            .reduce(0, (a, b) -> a + b); // Combina os elementos com a soma

        System.out.println("Soma dos números: " + soma); // Saída: Soma dos números: 15
    }
}
```

### Explicação:

- **reduce(0, (a, b) -> a + b)**: O valor inicial é 0, e cada elemento do Stream é somado ao acumulador.

---

## Exemplo 5: Uso de `collect` para Criar Coleções

A operação `collect` é frequentemente usada para coletar os resultados num `List`, `Set` ou outro tipo de coleção.

```
import java.util.Arrays;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class ExemploCollect {
    public static void main(String[] args) {
        // Lista de nomes
        List<String> nomes = Arrays.asList("Ana", "João", "Maria", "Ana");

        // Criar um conjunto (Set) com nomes únicos
        Set<String> nomesUnicos = nomes.stream()
            .collect(Collectors.toSet()); // Coleta os elementos num Set (remove
duplicados)

        System.out.println("Nomes únicos: " + nomesUnicos); // Saída: [Ana, João, Maria]
    }
}
```

### Explicação:

- **`collect(Collectors.toSet())`**: Coleta os elementos num `Set`, eliminando duplicados automaticamente.

---

## Exemplo 6: Stream Paralelo

Os **Streams paralelos** permitem processar os dados em múltiplos núcleos da CPU, melhorando o desempenho.

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

public class StreamParalelo {
    public static void main(String[] args) {
        // Lista de números
        List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Processar números em paralelo e calcular o dobro
        numeros.parallelStream() // Cria um Stream paralelo
            .map(n -> n * 2) // Calcula o dobro de cada número
            .forEach(System.out::println); // Imprime os resultados (ordem não
garantida)
    }
}
```

### Nota:

- **parallelStream()**: Divide os dados para serem processados em paralelo.
- **Ordem**: Streams paralelos não garantem a ordem de processamento.

---

## Resumo

### *Conceitos-Chave da API de Streams:*

#### 1. **Stream:**

- Um fluxo sequencial ou paralelo de dados.
- Não altera a coleção original.

#### 2. **Operações Intermediárias:**

- Transformam o Stream (e.g., `filter`, `map`, `distinct`).
- São preguiçosas (executadas apenas quando existe uma operação terminal).

#### 3. **Operações Terminais:**

- Consomem o Stream (e.g., `forEach`, `collect`, `reduce`).

#### 4. **Streams Paralelos:**

- Permitem processar grandes volumes de dados de forma eficiente.

## Tema Suplementar (sobre Streams): Combinação `map` e `reduce`

A combinação **`map` e `reduce`** é amplamente utilizada em programação funcional e é muito comum na **API de Streams do Java**. Esta combinação permite transformar os elementos de uma coleção (com `map`) e depois agregá-los para produzir um único valor (com `reduce`).

### Como Funciona a Combinação `map` e `reduce`?

#### 1. `map`:

- Aplica uma transformação a cada elemento do Stream.
- Exemplo: Converter uma lista de preços em euros para dólares.

#### 2. `reduce`:

- Combina os elementos transformados numa única saída.
  - Exemplo: Somar os preços convertidos.
- 

### Exemplo 1: Soma de Quadrados

Neste exemplo, usamos `map` para calcular o quadrado de cada número e `reduce` para somar os quadrados.

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

public class SomaQuadrados {
    public static void main(String[] args) {
        // Lista de números
        List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);

        // Calcular a soma dos quadrados
        int somaQuadrados = numeros.stream()
            .map(n -> n * n)           // Transforma cada número no seu quadrado
            .reduce(0, Integer::sum); // Soma os quadrados

        System.out.println("Soma dos quadrados: " + somaQuadrados); // Saída: Soma dos
quadrados: 55
    }
}
```

#### Explicação:

- `map(n -> n * n)`: Transforma cada número no seu quadrado.
- `reduce(0, Integer::sum)`: Soma todos os quadrados, começando do valor inicial 0.

---

## Exemplo 2: Cálculo da Receita Total

Suponha que temos uma lista de produtos com os respectivos preços e quantidades vendidas. Queremos calcular a receita total.

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

class Produto {
    String nome;
    double preco;
    int quantidade;

    Produto(String nome, double preco, int quantidade) {
        this.nome = nome;
        this.preco = preco;
        this.quantidade = quantidade;
    }
}

public class ReceitaTotal {
    public static void main(String[] args) {
        // Lista de produtos
        List<Produto> produtos = Arrays.asList(
            new Produto("Caneta", 1.50, 100),
            new Produto("Caderno", 5.00, 50),
            new Produto("Borracha", 0.50, 200)
        );

        // Calcular a receita total
        double receitaTotal = produtos.stream()
            .map(produto -> produto.preco * produto.quantidade) // Calcula a receita de cada
produto
            .reduce(0.0, Double::sum); // Soma as receitas

        System.out.println("Receita total: " + receitaTotal); // Saída: Receita total: 425.0
    }
}
```

### Explicação:

- **map(produto -> produto.preco \* produto.quantidade)**: Calcula a receita individual de cada produto.
- **reduce(0.0, Double::sum)**: Soma todas as receitas para obter o total.



---

### Exemplo 3: Contagem de Palavras em Textos

Este exemplo demonstra como usar `map` e `reduce` para contar o número total de palavras numa lista de frases.

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

public class ContarPalavras {
    public static void main(String[] args) {
        // Lista de frases
        List<String> frases = Arrays.asList(
            "Java é uma linguagem poderosa",
            "Streams facilitam o processamento de dados",
            "Expressões lambda tornam o código mais conciso"
        );

        // Calcular o número total de palavras
        int totalPalavras = frases.stream()
            .map(frase -> frase.split(" ").length) // Conta as palavras em cada frase
            .reduce(0, Integer::sum);               // Soma as contagens

        System.out.println("Número total de palavras: " + totalPalavras); // Saída:
        Número total de palavras: 15
    }
}
```

#### Explicação:

- `map(frase -> frase.split(" ").length)`: Transforma cada frase no número de palavras que contém.
- `reduce(0, Integer::sum)`: Soma o número de palavras de todas as frases.

---

### Benefícios da Combinação `map` e `reduce`

#### 1. Legibilidade:

- O código reflete claramente a sequência de transformação e agregação.

#### 2. Imutabilidade:

- A coleção original não é alterada.

#### 3. Paralelismo:

- Com `parallelStream()`, a combinação `map/reduce` pode ser paralelizada, aumentando o desempenho em grandes volumes de dados.

## Resumo

A combinação map/reduce é muito útil para:

1. **Transformar elementos** (com map).
2. **Agregá-los** num único valor (com reduce).

Esta abordagem é comum em tarefas como:

- Somar valores após transformação.
- Contar ocorrências.
- Calcular médias ou outros agregados.

### 3. O que são Referências a Métodos e Construtores no Java?

As **referências a métodos e construtores**, introduzidas no **Java 8**, são uma forma concisa de expressar **lambdas** que simplesmente chamam um método ou criam um objeto. Elas tornam o código mais limpo e legível, permitindo reutilizar métodos existentes diretamente em operações como `map`, `filter` e `forEach`.

---

#### Tipos de Referências

Existem **quatro tipos principais** de referências a métodos e construtores:

**1. Referência a um método estático**

- Usa a sintaxe: `Classe::metodoEstatico`.
- Exemplo: `Math::abs`.

**2. Referência a um método de instância de um objeto específico**

- Usa a sintaxe: `instancia::metodo`.
- Exemplo: `meuObjeto::toString`.

**3. Referência a um método de instância de um objeto arbitrário de um tipo específico**

- Usa a sintaxe: `Classe::metodoDeInstancia`.
- Exemplo: `String::toUpperCase`.

**4. Referência a um construtor**

- Usa a sintaxe: `Classe::new`.
- Exemplo: `ArrayList::new`.

---

## Exemplos de Cada Tipo

### *1. Referência a um Método Estático*

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

public class MetodoEstatico {
    public static void main(String[] args) {
        // Lista de números
        List<Integer> numeros = Arrays.asList(-10, -20, 30, 40);

        // Usar Math::abs para calcular o valor absoluto
        numeros.stream()
            .map(Math::abs) // Referência ao método estático Math.abs
            .forEach(System.out::println); // Imprime: 10, 20, 30, 40
    }
}
```

#### **Explicação:**

- `Math::abs` é uma referência ao método estático `abs` da classe `Math`.
- Substitui a lambda `(n -> Math.abs(n))`.

---

## 2. Referência a um Método de Instância de um Objeto Específico

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

public class MetodoInstanciaEspecifico {
    public static void main(String[] args) {
        // Lista de palavras
        List<String> palavras = Arrays.asList("Java", "Stream", "Lambda");

        // Criar um StringBuilder
        StringBuilder sb = new StringBuilder();

        // Usar o método append do StringBuilder
        palavras.forEach(sb::append); // Referência ao método append
        System.out.println(sb.toString()); // Imprime: JavaStreamLambda
    }
}
```

### Explicação:

- `sb::append` refere-se ao método `append` do objeto específico `sb` de tipo `StringBuilder`.
- Substitui a lambda (`palavra -> sb.append(palavra)`).

---

### 3. Referência a um Método de Instância de um Objeto Arbitrário

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

public class MetodoInstanciaArbitrario {
    public static void main(String[] args) {
        // Lista de nomes
        List<String> nomes = Arrays.asList("ana", "joão", "maria");

        // Usar String::toUpperCase para converter para maiúsculas
        nomes.stream()
            .map(String::toUpperCase) // Referência ao método toUpperCase de String
            .forEach(System.out::println); // Imprime: ANA, JOÃO, MARIA
    }
}
```

#### Explicação:

- `String::toUpperCase` refere-se ao método `toUpperCase` de qualquer objeto do tipo `String`.
- Substitui a lambda `(s -> s.toUpperCase())`.

---

#### 4. Referência a um Construtor

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

class Produto {
    String nome;

    Produto(String nome) {
        this.nome = nome;
    }
}

public class ReferenciaConstrutor {
    public static void main(String[] args) {
        // Lista de nomes de produtos
        List<String> nomes = Arrays.asList("Caneta", "Caderno", "Borracha");

        // Usar Produto::new para criar objetos Produto
        List<Produto> produtos = nomes.stream()
            .map(Produto::new) // Referência ao construtor
            .toList();

        // Imprimir os nomes dos produtos
        produtos.forEach(produto -> System.out.println(produto.nome));
        // Saída: Caneta, Caderno, Borracha
    }
}
```

##### Explicação:

- `Produto::new` refere-se ao construtor da classe `Produto`.
- Substitui a lambda `(nome -> new Produto(nome))`.

---

## Vantagens das Referências a Métodos e Construtores

### 1. Código Mais Conciso:

- Eliminam a necessidade de criar lambdas que apenas chamam métodos.

### 2. Melhor Legibilidade:

- O foco é na ação (como `String::toUpperCase`) em vez de na estrutura da lambda.

### 3. Reutilização:

- Métodos existentes podem ser reutilizados diretamente.
- 

## Resumo Comparativo

Tipo	Sintaxe	Exemplo	Equivalente a Lambda
Método Estático	<code>Classe::metodoEstatico</code>	<code>Math::abs</code>	<code>n -&gt; Math.abs(n)</code>
Método de Instância de um Objeto Específico	<code>instancia::metodo</code>	<code>sb::append</code>	<code>s -&gt; sb.append(s)</code>
Método de Instância de um Objeto Arbitrário	<code>Classe::metodoInstancia</code>	<code>String::toUpperCase</code>	<code>s -&gt; s.toUpperCase()</code>
Construtor	<code>Classe::new</code>	<code>Produto::new</code>	<code>nome -&gt; new Produto(nome)</code>

---

## Dicas Práticas

### 1. Escolha Simplicidade:

- Use referências a métodos quando estas tornarem o código mais legível.

### 2. Teste Alternativas:

- Se a referência não for clara no contexto, use lambdas explicitamente.

### 3. Identifique o Tipo:

- Considere se está a referenciar um método estático, um método de instância ou um construtor.



## Exemplos completos de Referências a Métodos e Construtores:

---

### Exemplo 1: Referência a um Método Estático

Este tipo de referência permite usar métodos estáticos diretamente no pipeline de Streams ou em outras situações.

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

public class MetodoEstatico {
    public static void main(String[] args) {
        // Lista de números inteiros
        List<Integer> numeros = Arrays.asList(-10, -20, 30, 40);

        // Usar uma referência ao método estático Math.abs para calcular valores absolutos
        numeros.stream()
            .map(Math::abs) // Math::abs refere-se ao método estático abs da classe Math
            .forEach(System.out::println); // Imprime os valores absolutos: 10, 20, 30, 40
    }
}
```

#### Comentários no Código:

- `Math::abs` refere-se ao método **estático** `abs` da classe `Math`.
- Substitui a lambda equivalente: `(n -> Math.abs(n))`.

---

## Exemplo 2: Referência a um Método de Instância de um Objeto Específico

Se já tens uma instância de um objeto, podes referir-te diretamente a um dos seus métodos.

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

public class MetodoInstanciaEspecifico {
    public static void main(String[] args) {
        // Lista de palavras
        List<String> palavras = Arrays.asList("Java", "Streams", "Lambda");

        // Criar uma instância de StringBuilder
        StringBuilder sb = new StringBuilder();

        // Usar a referência ao método append do StringBuilder
        palavras.forEach(sb::append); // sb::append refere-se ao método de instância
append
        System.out.println(sb.toString()); // Imprime: JavaStreamsLambda
    }
}
```

### Comentários no Código:

- `sb::append` refere-se ao método `append` do objeto `sb` (uma instância de `StringBuilder`).
- Substitui a lambda equivalente: `(palavra -> sb.append(palavra))`.

---

### Exemplo 3: Referência a um Método de Instância de um Objeto Arbitrário

Este tipo de referência aplica-se a objetos arbitrários de um tipo específico.

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

public class MetodoInstanciaArbitrario {
    public static void main(String[] args) {
        // Lista de nomes
        List<String> nomes = Arrays.asList("ana", "joão", "maria");

        // Converter todos os nomes para maiúsculas
        nomes.stream()
            .map(String::toUpperCase) // String::toUpperCase refere-se ao método
toUpperCase de String
            .forEach(System.out::println); // Imprime: ANA, JOÃO, MARIA
    }
}
```

#### Comentários no Código:

- `String::toUpperCase` refere-se ao método de instância `toUpperCase` aplicado a cada elemento do `Stream`.
- Substitui a lambda equivalente: `(s -> s.toUpperCase())`.

---

## Exemplo 4: Referência a um Construtor

Este tipo de referência permite criar objetos diretamente no pipeline de Streams.

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

class Produto {
    String nome;

    // Construtor da classe Produto
    Produto(String nome) {
        this.nome = nome;
    }
}

public class ReferenciaConstrutor {
    public static void main(String[] args) {
        // Lista de nomes de produtos
        List<String> nomes = Arrays.asList("Caneta", "Caderno", "Borracha");

        // Criar uma lista de objetos Produto usando uma referência ao construtor
        Produto::new
        List<Produto> produtos = nomes.stream()
                                    .map(Produto::new) // Produto::new refere-se ao
construtor
                                    .collect(Collectors.toList());

        // Imprimir os nomes dos produtos
        produtos.forEach(produto -> System.out.println(produto.nome)); // Imprime:
Caneta, Caderno, Borracha
    }
}
```

### Comentários no Código:

- `Produto::new` refere-se ao construtor da classe `Produto`, que aceita um parâmetro `String`.
- Substitui a lambda equivalente: `(nome -> new Produto(nome))`.

---

## Exemplo 5: Comparação entre Referências e Lambdas

Para clarificar a diferença entre usar lambdas e referências, vejamos o mesmo exemplo escrito das duas formas.

### *Com Lambda:*

```
List<String> nomes = Arrays.asList("ana", "joão", "maria");
nomes.stream()
    .map(nome -> nome.toUpperCase()) // Lambda para converter para maiúsculas
    .forEach(nome -> System.out.println(nome)); // Lambda para imprimir
```

### *Com Referências:*

```
List<String> nomes = Arrays.asList("ana", "joão", "maria");
nomes.stream()
    .map(String::toUpperCase) // Referência ao método de instância toUpperCase
    .forEach(System.out::println); // Referência ao método println
```

### **Vantagem das Referências:**

- O código é mais conciso e legível, uma vez que evita a repetição de lógica.

---

## Resumo dos Tipos de Referências

Tipo	Sintaxe	Exemplo	Equivalente Lambda
Método Estático	Classe::metodoEstatico	Math::abs	n -> Math.abs(n)
Método de Instância de Objeto Específico	objeto::metodo	sb::append	s -> sb.append(s)
Método de Instância de Objeto Arbitrário	Classe::metodoDeInstancia	String::toUpperCase	s -> s.toUpperCase()
Construtor	Classe::new	Produto::new	nome -> new Produto(nome)

---

## Dicas Práticas

1. Use **referências a métodos** quando estas tornarem o código mais legível.
2. Escolha lambdas explícitas se as referências não forem claras no contexto.
3. Lembre-se que referências a métodos são apenas uma forma concisa de escrever lambdas que chamam métodos existentes.

## 4. Classe Optional

### O que é a classe Optional no Java?

A classe **Optional**, introduzida no **Java 8**, é uma classe do pacote `java.util` que representa um valor que pode estar **presente** ou **ausente**. O objetivo principal de **Optional** é evitar o uso descontrolado de `null`, que frequentemente resulta em **NullPointerException** (um dos erros mais comuns em Java).

---

### Porque Usar Optional?

#### 1. Evitar NullPointerException:

- Em vez de verificar manualmente se um valor é `null`, o **Optional** fornece métodos seguros para lidar com a ausência de valor.

#### 2. Código Mais Claro:

- Torna explícito no código que um valor pode estar ausente, melhorando a legibilidade e a manutenção.

#### 3. Encapsulamento da Lógica de Verificação:

- Reduz a necessidade de fazer verificações manuais como `if (obj != null)`.

---

## Como Criar um Optional?

O `Optional` pode ser criado de várias formas, dependendo da situação:

### 1. `Optional.of(T valor)`:

- Cria um `Optional` com o valor fornecido (o valor não pode ser `null`).
- Exemplo:

```
Optional<String> opcional = Optional.of("Java");
```

### 2. `Optional.ofNullable(T valor)`:

- Cria um `Optional` que pode conter um valor ou estar vazio (`Optional.empty()`) se o valor for `null`.
- Exemplo:

```
Optional<String> opcional = Optional.ofNullable(null);
```

### 3. `Optional.empty()`:

- Cria um `Optional` vazio.
- Exemplo:

```
Optional<String> vazio = Optional.empty();
```



---

## Métodos Comuns do Optional

### *1. Verificar a Presença de Valor*

- **isPresent():**

- Retorna true se o valor estiver presente.
- Exemplo:

```
Optional<String> opcional = Optional.of("Java");  
System.out.println(opcional.isPresent()); // true
```

- **isEmpty()** (Introduzido no Java 11):

- Retorna true se o valor estiver ausente.
- Exemplo:

```
Optional<String> vazio = Optional.empty();  
System.out.println(vazio.isEmpty()); // true
```

---

## 2. Obter o Valor

- **get():**
  - Retorna o valor, mas lança uma exceção se estiver ausente.
  - Exemplo:

```
Optional<String> opcional = Optional.of("Java");  
System.out.println(opcional.get()); // "Java"
```
- **Evite usar get() diretamente**, porque pode lançar exceção. Prefira métodos mais seguros como `orElse`.

---

### 3. Fornecer um Valor Padrão

- **orElse(T outroValor):**

- Retorna o valor contido no `Optional` ou um valor padrão se estiver vazio.
- Exemplo:

```
Optional<String> opcional = Optional.ofNullable(null);  
System.out.println(opcional.orElse("Valor Padrão")); // "Valor Padrão"
```

- **orElseGet(Supplier<? extends T> fornecedor):**

- Semelhante a `orElse`, mas o valor padrão é gerado por um fornecedor.
- Exemplo:

```
Optional<String> opcional = Optional.ofNullable(null);  
System.out.println(opcional.orElseGet(() -> "Valor Gerado")); // "Valor Gerado"
```

- **orElseThrow(Supplier<? extends Throwable> fornecedor):**

- Lança uma exceção personalizada se o valor estiver ausente.
- Exemplo:

```
Optional<String> opcional = Optional.ofNullable(null);  
opcional.orElseThrow(() -> new IllegalArgumentException("Valor ausente!"));
```

---

#### 4. Executar Ações Condicionais

- **ifPresent(Consumer<? super T> consumidor):**

- Executa uma ação se o valor estiver presente.
- Exemplo:

```
Optional<String> opcional = Optional.of("Java");
opcional.ifPresent(valor -> System.out.println("Valor: " + valor)); //
"Valor: Java"
```

- **ifPresentOrElse(Consumer<? super T> consumidor, Runnable runnable)** (Introduzido no Java 9):

- Executa uma ação se o valor estiver presente, ou outra ação se estiver ausente.
- Exemplo:

```
Optional<String> opcional = Optional.ofNullable(null);
opcional.ifPresentOrElse(
    valor -> System.out.println("Valor: " + valor),
    () -> System.out.println("Valor ausente")
);
// Saída: "Valor ausente"
```

---

## 5. Transformar o Valor

- **map(Function<? super T, ? extends U> mapper):**

- Transforma o valor contido num Optional e retorna um novo Optional.
- Exemplo:

```
Optional<String> opcional = Optional.of("Java");  
Optional<Integer> tamanho = opcional.map(String::length);  
System.out.println(tamanho.orElse(0)); // 4
```

- **flatMap(Function<? super T, Optional<U>> mapper):**

- Semelhante a map, mas evita aninhamento de Optional<Optional<U>>.
- Exemplo:

```
Optional<String> opcional = Optional.of("Java");  
Optional<String> resultado = opcional.flatMap(valor ->  
Optional.of(valor.toUpperCase()));  
System.out.println(resultado.orElse("Sem valor")); // "JAVA"
```

---

## Exemplo Prático: Uso de Optional

Aqui está um exemplo que demonstra como o Optional pode ser usado para evitar verificações manuais de null.

```
import java.util.Optional;

public class ExemploOptional {
    public static void main(String[] args) {
        // Método que pode devolver um valor ou null
        Optional<String> nome = encontrarNome("João");

        // Forma segura de lidar com o valor
        nome.ifPresentOrElse(
            valor -> System.out.println("Nome encontrado: " + valor), // Caso o valor exista
            () -> System.out.println("Nome não encontrado")           // Caso o valor esteja ausente
        );
    }

    // Simula a busca de um nome numa base de dados
    public static Optional<String> encontrarNome(String nome) {
        if ("João".equals(nome)) {
            return Optional.of("João Silva");
        }
        return Optional.empty();
    }
}
```

### Saída:

Nome encontrado: João Silva

---

## Resumo: Quando Usar `Optional`?

### 1. Representar Valores Opcionais:

- Use `Optional` para métodos que podem devolver um valor ou nada (ex.: buscar dados numa base de dados).

### 2. Evitar Erros de `NullPointerException`:

- Substitua verificações manuais de `null` pelo uso de métodos como `orElse` ou `ifPresent`.

### 3. Melhorar a Legibilidade:

- Torne claro no código que um valor pode estar ausente.

## Exemplos completos sobre o conceito de **Optional**:

---

### Exemplo 1: Criação de um Optional

Demonstra as formas de criar e trabalhar com um Optional.

```
import java.util.Optional;

public class ExemploOptional1 {
    public static void main(String[] args) {
        // Criar um Optional com valor presente
        Optional<String> opcionalPresente = Optional.of("Java");
        System.out.println("Valor presente: " + opcionalPresente.get()); // Saída: Java

        // Criar um Optional vazio
        Optional<String> opcionalVazio = Optional.empty();
        System.out.println("Está vazio? " + opcionalVazio.isEmpty()); // Saída: true

        // Criar um Optional com valor que pode ser null
        Optional<String> opcionalNullable = Optional.ofNullable(null);
        System.out.println("Está presente? " + opcionalNullable.isPresent()); // Saída: false
    }
}
```

#### Comentários no Código:

1. **Optional.of(valor):**
  - Cria um Optional com o valor fornecido.
  - Lança uma exceção se o valor for null.
2. **Optional.empty():**
  - Cria um Optional vazio.
3. **Optional.ofNullable(valor):**
  - Permite criar um Optional que pode conter null.



---

## Exemplo 2: Evitar NullPointerException

Demonstra como evitar o uso de verificações manuais de null.

```
import java.util.Optional;

public class ExemploOptional2 {
    public static void main(String[] args) {
        // Método que pode devolver null (simulação de base de dados)
        Optional<String> nome = buscarNome("João");

        // Verificar se o valor está presente e agir de acordo
        nome.ifPresentOrElse(
            valor -> System.out.println("Nome encontrado: " + valor), // Caso esteja presente
            () -> System.out.println("Nome não encontrado")           // Caso esteja ausente
        );
    }

    // Simula um método que busca um nome e pode devolver null
    public static Optional<String> buscarNome(String nome) {
        if ("João".equals(nome)) {
            return Optional.of("João Silva");
        }
        return Optional.empty();
    }
}
```

### Comentários no Código:

- **ifPresentOrElse:**
  - Executa uma ação caso o valor esteja presente (ifPresent) ou outra ação caso esteja ausente (orElse).

---

## Exemplo 3: Fornecer um Valor Padrão

Utiliza métodos como `orElse` e `orElseGet` para fornecer valores alternativos.

```
import java.util.Optional;

public class ExemploOptional3 {
    public static void main(String[] args) {
        // Método que pode devolver um Optional vazio
        Optional<String> opcional = buscarNome("Maria");

        // Usar um valor padrão caso o Optional esteja vazio
        String nome = opcional.orElse("Valor Padrão");
        System.out.println("Nome: " + nome); // Saída: Nome: Valor Padrão

        // Usar um valor gerado dinamicamente
        String nomeGerado = opcional.orElseGet(() -> "Nome Gerado Dinamicamente");
        System.out.println("Nome: " + nomeGerado); // Saída: Nome: Nome Gerado Dinamicamente
    }

    public static Optional<String> buscarNome(String nome) {
        if ("João".equals(nome)) {
            return Optional.of("João Silva");
        }
        return Optional.empty();
    }
}
```

### Comentários no Código:

- **orElse(valorPadrão):**
  - Retorna o valor contido no `Optional` ou o valor padrão se estiver vazio.
- **orElseGet(fornecedor):**
  - Semelhante a `orElse`, mas o valor padrão é gerado dinamicamente.

---

## Exemplo 4: Transformar Valores com map

Demonstra como transformar o valor contido no Optional.

```
import java.util.Optional;

public class ExemploOptional4 {
    public static void main(String[] args) {
        // Método que devolve um nome
        Optional<String> nome = buscarNome("João");

        // Transformar o nome para maiúsculas
        Optional<String> nomeMaiusculas = nome.map(String::toUpperCase);

        // Imprimir o resultado
        System.out.println(nomeMaiusculas.orElse("Sem valor")); // Saída: JOÃO SILVA
    }

    public static Optional<String> buscarNome(String nome) {
        if ("João".equals(nome)) {
            return Optional.of("João Silva");
        }
        return Optional.empty();
    }
}
```

### Comentários no Código:

- **map:**
  - Transforma o valor contido no Optional e devolve um novo Optional.

---

## Exemplo 5: Combinar Optional com flatMap

Evita o aninhamento de Optional ao lidar com valores opcionais.

```
import java.util.Optional;

public class ExemploOptional5 {
    public static void main(String[] args) {
        // Método que devolve um Optional aninhado
        Optional<Optional<String>> nome = Optional.of(Optional.of("João Silva"));

        // Usar flatMap para evitar o aninhamento
        Optional<String> nomePlano = nome.flatMap(valor -> valor);

        // Imprimir o resultado
        System.out.println(nomePlano.orElse("Sem valor")); // Saída: JOÃO SILVA
    }
}
```

### Comentários no Código:

- **flatMap:**
  - Semelhante a map, mas evita o aninhamento de Optional<Optional<T>>.

---

## Exemplo 6: Lançar Exceções com `orElseThrow`

Demonstra como lançar uma exceção personalizada caso o valor esteja ausente.

```
import java.util.Optional;

public class ExemploOptional6 {
    public static void main(String[] args) {
        // Método que pode devolver um Optional vazio
        Optional<String> nome = buscarNome("Maria");

        // Lançar uma exceção se o valor estiver ausente
        String resultado = nome.orElseThrow(() -> new IllegalArgumentException("Nome não encontrado!"));
        System.out.println(resultado);
    }

    public static Optional<String> buscarNome(String nome) {
        if ("João".equals(nome)) {
            return Optional.of("João Silva");
        }
        return Optional.empty();
    }
}
```

### Comentários no Código:

- **`orElseThrow`:**
  - Lança uma exceção personalizada caso o `Optional` esteja vazio.

---

## Resumo

1. **Optional** representa um valor que pode estar presente ou ausente.
2. **Métodos Importantes:**
  - `isPresent`, `orElse`, `orElseGet`, `orElseThrow`, `map`, `flatMap`, `ifPresent`, `ifPresentOrElse`.
3. **Vantagens:**
  - Reduz o uso de `null` e evita erros como `NullPointerException`.
  - Torna o código mais claro e seguro.

## 5. Melhorias na API de Coleções no Java 8

Com a introdução do **Java 8**, a **API de Coleções** foi significativamente melhorada para tornar o trabalho com coleções mais eficiente e intuitivo. Estas melhorias incluem a integração com a **API de Streams**, novos métodos funcionais e funcionalidades adicionais para manipulação direta das coleções.

---

### 1. Integração com a API de Streams

As coleções no Java 8 foram integradas com a API de Streams, permitindo operações como **filtragem**, **transformação** e **redução** de dados de forma declarativa.

#### *Exemplo: Filtrar e Processar uma Lista*

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

public class ExemploStreamsComColecoes {
    public static void main(String[] args) {
        // Lista de números
        List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

        // Filtrar números pares e calcular o dobro
        numeros.stream()
            .filter(n -> n % 2 == 0) // Filtra números pares
            .map(n -> n * 2)          // Calcula o dobro de cada número
            .forEach(System.out::println); // Imprime os resultados: 4, 8, 12, 16
    }
}
```

#### **Explicação:**

- **stream()**: Cria um Stream a partir da coleção.
- **filter** e **map**: Operações declarativas sobre os elementos da lista.
- **forEach**: Consome os resultados e executa uma ação para cada elemento.

---

## 2. Novos Métodos em Coleções

### a) *forEach*

Adicionado à interface `Iterable`, permite executar uma ação para cada elemento da coleção.

#### Exemplo:

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

public class ExemploForEach {
    public static void main(String[] args) {
        List<String> nomes = Arrays.asList("Ana", "João", "Maria");

        // Usar forEach para imprimir cada nome
        nomes.forEach(nome -> System.out.println("Nome: " + nome));
    }
}
```

#### Explicação:

- **forEach**: Substitui o loop tradicional para iterar sobre uma coleção.

---

### b) *removeIf*

Permite remover elementos da coleção com base numa condição.

#### Exemplo:

```
import java.util.ArrayList;
import java.util.List;

public class ExemploRemoveIf {
    public static void main(String[] args) {
        List<Integer> numeros = new ArrayList<>(List.of(1, 2, 3, 4, 5, 6));

        // Remover números ímpares
        numeros.removeIf(n -> n % 2 != 0);

        System.out.println("Apenas números pares: " + numeros); // Saída: [2, 4, 6]
    }
}
```

#### Explicação:

- **removeIf**: Remove todos os elementos que satisfazem a condição fornecida.



### c) *replaceAll*

Permite substituir cada elemento da lista com base numa função.

#### **Exemplo:**

```
import java.util.ArrayList;
import java.util.List;

public class ExemploReplaceAll {
    public static void main(String[] args) {
        List<Integer> numeros = new ArrayList<>(List.of(1, 2, 3, 4, 5));

        // Multiplicar cada número por 2
        numeros.replaceAll(n -> n * 2);

        System.out.println("Números duplicados: " + numeros); // Saída: [2, 4, 6, 8, 10]
    }
}
```

#### **Explicação:**

- **replaceAll**: Aplica a função fornecida a cada elemento, substituindo-o pelo resultado.

---

#### *d) compute, computeIfAbsent e computeIfPresent*

Estes métodos foram adicionados à interface Map para simplificar a manipulação de valores associados a chaves.

##### **Exemplo: computeIfAbsent**

```
import java.util.HashMap;
import java.util.Map;

public class ExemploComputeIfAbsent {
    public static void main(String[] args) {
        Map<String, Integer> mapa = new HashMap<>();
        mapa.put("Ana", 10);

        // Adicionar valor se a chave não existir
        mapa.computeIfAbsent("João", k -> 20);
        mapa.computeIfAbsent("Ana", k -> 30); // "Ana" já existe, não será alterado

        System.out.println(mapa); // Saída: {Ana=10, João=20}
    }
}
```

##### **Explicação:**

- **computeIfAbsent:** Calcula e adiciona um valor apenas se a chave não existir no mapa.

---

### 3. Novos Coletores com Collectors

O pacote `java.util.stream.Collectors` oferece métodos utilitários para manipular Streams e recolher resultados em coleções.

#### *Exemplo: Transformar uma Lista num Map*

```
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class ExemploCollectors {
    public static void main(String[] args) {
        List<String> nomes = List.of("Ana", "João", "Maria");

        // Transformar a lista num mapa onde a chave é o nome e o valor é o tamanho do
nome
        Map<String, Integer> mapa = nomes.stream()
                                        .collect(Collectors.toMap(nome -> nome, nome ->
nome.length()));

        System.out.println(mapa); // Saída: {Ana=3, João=4, Maria=5}
    }
}
```

#### **Explicação:**

- **toMap:** Cria um Map a partir de um Stream, usando funções para definir as chaves e os valores.

---

## 4. Criação de Coleções Imutáveis

No Java 8, métodos como `List.of`, `Set.of` e `Map.of` permitem criar coleções imutáveis de forma mais concisa.

### Exemplo:

```
import java.util.List;

public class ExemploColecoesImutaveis {
    public static void main(String[] args) {
        List<String> listaImutavel = List.of("Ana", "João", "Maria");

        System.out.println(listaImutavel); // Saída: [Ana, João, Maria]

        // listaImutavel.add("Pedro"); // Lança UnsupportedOperationException
    }
}
```

### Explicação:

- **List.of**: Cria uma lista imutável.
- A lista não pode ser modificada após a criação.

---

## Resumo das Melhorias

Funcionalidade	Método	Descrição
Iteração Melhorada	forEach	Executa uma ação para cada elemento.
Filtragem	removeIf	Remove elementos com base numa condição.
Transformação	replaceAll	Substitui elementos com base numa função.
Manipulação de Mapas	compute*	Simplifica atualizações baseadas em chaves.
Integração com Streams	stream()	Permite transformar, filtrar e reduzir coleções declarativamente.
Criação de Coleções Imutáveis	List.of, Set.of	Cria coleções imutáveis de forma simples e segura.

## Exemplos completos sobre melhorias na **API de Coleções** no Java 8:

---

### Exemplo 1: Iteração Melhorada com `forEach`

O método `forEach`, adicionado à interface `Iterable`, permite executar uma ação para cada elemento da coleção de forma mais concisa.

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

public class ExemploForEach {
    public static void main(String[] args) {
        // Lista de nomes
        List<String> nomes = Arrays.asList("Ana", "João", "Maria");

        // Iteração com forEach
        nomes.forEach(nome -> System.out.println("Nome: " + nome));

        // Alternativa com referência a método
        nomes.forEach(System.out::println); // Mais conciso
    }
}
```

#### Comentários no Código:

##### 1. `forEach`:

- Substitui o loop tradicional para iterar sobre os elementos da coleção.
- Aceita uma **expressão lambda** ou uma **referência a método**.

---

## Exemplo 2: Remover Elementos com `removeIf`

O método `removeIf` remove elementos da coleção com base numa condição.

```
import java.util.ArrayList;
import java.util.List;

public class ExemploRemoveIf {
    public static void main(String[] args) {
        // Lista de números
        List<Integer> numeros = new ArrayList<>(List.of(1, 2, 3, 4, 5, 6, 7, 8, 9));

        // Remover números pares
        numeros.removeIf(n -> n % 2 == 0);

        System.out.println("Apenas números ímpares: " + numeros); // Saída: [1, 3, 5, 7,
9]
    }
}
```

### Comentários no Código:

- **`removeIf`:**
  - Remove todos os elementos que satisfazem a condição fornecida (números pares, neste caso).

---

### Exemplo 3: Transformar Elementos com `replaceAll`

O método `replaceAll` permite modificar os elementos de uma lista com base numa função.

```
import java.util.ArrayList;
import java.util.List;

public class ExemploReplaceAll {
    public static void main(String[] args) {
        // Lista de números
        List<Integer> numeros = new ArrayList<>(List.of(1, 2, 3, 4, 5));

        // Multiplicar cada número por 10
        numeros.replaceAll(n -> n * 10);

        System.out.println("Números multiplicados por 10: " + numeros); // Saída: [10,
20, 30, 40, 50]
    }
}
```

#### Comentários no Código:

- **`replaceAll`:**
  - Aplica uma função a cada elemento da lista e substitui o elemento pelo resultado.



---

## Exemplo 4: Manipulação de Mapas com `computeIfAbsent`

Os métodos `computeIfAbsent`, `computeIfPresent` e `compute` facilitam a manipulação de valores em mapas.

```
import java.util.HashMap;
import java.util.Map;

public class ExemploComputeIfAbsent {
    public static void main(String[] args) {
        // Mapa com nomes e pontuações
        Map<String, Integer> pontuacoes = new HashMap<>();
        pontuacoes.put("Ana", 10);

        // Adicionar uma pontuação apenas se a chave não existir
        pontuacoes.computeIfAbsent("João", nome -> 20);
        pontuacoes.computeIfAbsent("Ana", nome -> 30); // Não altera porque "Ana" já
existe

        System.out.println("Pontuações: " + pontuacoes); // Saída: {Ana=10, João=20}
    }
}
```

### Comentários no Código:

- **`computeIfAbsent`:**
  - Adiciona um valor calculado apenas se a chave não existir no mapa.

---

## Exemplo 5: Integração com Streams

As coleções no Java 8 foram integradas com a API de Streams, permitindo operações como filtrar, mapear e reduzir elementos.

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

public class ExemploStreamsComColecoes {
    public static void main(String[] args) {
        // Lista de nomes
        List<String> nomes = Arrays.asList("Ana", "João", "Maria", "Joana");

        // Filtrar nomes que começam com "J" e convertê-los para maiúsculas
        nomes.stream()
            .filter(nome -> nome.startsWith("J")) // Filtra nomes que começam com "J"
            .map(String::toUpperCase)             // Converte os nomes para maiúsculas
            .forEach(System.out::println);        // Imprime: JOÃO, JOANA
    }
}
```

### Comentários no Código:

1. **stream():**
  - Cria um Stream a partir da coleção.
2. **filter e map:**
  - Operações intermediárias para filtrar e transformar os elementos.
3. **forEach:**
  - Operação terminal que consome os resultados.

---

## Exemplo 6: Criação de Coleções Imutáveis

No Java 8, os métodos `List.of`, `Set.of` e `Map.of` permitem criar coleções imutáveis.

```
import java.util.List;
import java.util.Set;

public class ExemploColecoesImutaveis {
    public static void main(String[] args) {
        // Criar uma lista imutável
        List<String> listaImutavel = List.of("Ana", "João", "Maria");

        // Criar um conjunto imutável
        Set<String> conjuntoImutavel = Set.of("Java", "Streams", "Lambda");

        System.out.println("Lista: " + listaImutavel);          // Saída: [Ana, João, Maria]
        System.out.println("Conjunto: " + conjuntoImutavel);   // Saída: [Java, Streams,
Lambda]

        // listaImutavel.add("Pedro"); // Lança UnsupportedOperationException
    }
}
```

### Comentários no Código:

- **List.of e Set.of:**
  - Criam coleções imutáveis, que não podem ser alteradas após a criação.

## Resumo das Melhorias

Método	Descrição
<b>forEach</b>	Executa uma ação para cada elemento da coleção.
<b>removeIf</b>	Remove elementos da coleção com base numa condição.
<b>replaceAll</b>	Substitui elementos numa lista com base numa função.
<b>compute*</b>	Simplifica manipulações de valores em mapas.
<b>stream</b>	Permite trabalhar com coleções de forma declarativa usando a API de Streams.
<b>List.of</b> e <b>Set.of</b>	Cria coleções imutáveis de forma concisa.

---

## Conclusão

Estas melhorias na API de Coleções tornam o trabalho com coleções:

1. **Mais Declarativo:**
  - Métodos como `forEach` e `stream` promovem um estilo funcional de programação.
2. **Mais Flexível:**
  - Métodos como `removeIf` e `replaceAll` simplificam operações comuns.
3. **Mais Seguro:**
  - Coleções imutáveis ajudam a evitar modificações acidentais.

## 6. Sobre o *warning* relacionado com (não) fecho de Scanner

Ao programar em Java, é comum utilizar a classe `Scanner` para ler dados de diferentes fontes, como ficheiros, fluxos de entrada (`System.in`) ou strings. No entanto, ao usar o `Scanner`, especialmente com o fluxo de entrada padrão (`System.in`), o Visual Studio Code (VS Code) frequentemente apresenta um aviso indicando que o objeto `Scanner` deve ser fechado. Este aviso tem como objetivo alertar os programadores para a importância de gerir corretamente os recursos utilizados pela aplicação, prevenindo fugas de memória ou outros problemas relacionados com o uso indevido de recursos.

Nesta secção, será explorado quando e por que razão deve fechar o `Scanner`, com especial foco nos casos em que é necessário, como no acesso a ficheiros, e nas implicações práticas de ignorar este passo.

O método **`close()`** do **`Scanner`** é usado para libertar os recursos associados ao objeto (como ficheiros ou fluxos de entrada) quando este já não é necessário. A necessidade de usar **`close()`** depende do tipo de recurso que o `Scanner` está a usar e da versão do **Java**.

---

## Casos em que é necessário fechar o Scanner

### 1. Scanner associado a um ficheiro

Se o Scanner estiver associado a um ficheiro ou a outro recurso que implemente **Closeable** (como streams), é essencial chamar **close()** para evitar fugas de recursos.

#### Exemplo:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ScannerComFicheiro {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner scanner = new Scanner(new File("exemplo.txt"));

        while (scanner.hasNextLine()) {
            System.out.println(scanner.nextLine());
        }

        // É importante fechar o Scanner
        scanner.close();
    }
}
```

#### Porquê fechar?

- O Scanner usa um recurso externo (neste caso, um ficheiro) e deve ser fechado explicitamente para libertar o recurso.

---

## 2. *Scanner associado a um InputStream personalizado*

Se o Scanner estiver associado a um **InputStream** (como `FileInputStream` ou `BufferedInputStream`), também é necessário fechá-lo para libertar os recursos.

### **Exemplo:**

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ScannerComInputStream {
    public static void main(String[] args) throws FileNotFoundException {
        FileInputStream fis = new FileInputStream("exemplo.txt");
        Scanner scanner = new Scanner(fis);

        while (scanner.hasNextLine()) {
            System.out.println(scanner.nextLine());
        }

        // Fechar o Scanner (e implicitamente o FileInputStream)
        scanner.close();
    }
}
```

---

## Casos em que o Scanner não precisa de ser fechado explicitamente

### 1. Scanner associado a System.in

Quando o Scanner está ligado ao **System.in** (entrada padrão do teclado), **não é recomendado fechá-lo**: fechar o Scanner também fecha o fluxo subjacente (**System.in**), tornando-o inutilizável em outras partes do programa.

#### Exemplo (não fechar o Scanner):

```
import java.util.Scanner;

public class ScannerComTeclado {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Insira o seu nome: ");
        String nome = scanner.nextLine();
        System.out.println("Olá, " + nome);

        // Não é necessário fechar o Scanner ligado a System.in
        // scanner.close(); // Pode causar problemas se outros métodos tentarem usar System.in
    }
}
```

#### Nota:

- A partir do momento em que se fecha o fluxo **System.in**, ele não pode ser reaberto.



---

## 2. Scanner associado a uma String

Quando o Scanner é associado a uma String, não é obrigatório fechar o objeto porque o recurso associado (a String) está na memória e não depende de um fluxo externo. No entanto, não há problema em chamar o método `close()`, pois este não terá impacto negativo.

### Exemplo com String

```
import java.util.Scanner;

public class ScannerComString {
    public static void main(String[] args) {
        String dados = "1,2,3,4,5";
        Scanner scanner = new Scanner(dados);

        // Configurar o delimitador como uma vírgula
        scanner.useDelimiter(",");

        // Ler e imprimir os números da string
        while (scanner.hasNextInt()) {
            System.out.println(scanner.nextInt());
        }

        // Opcional: Fechar o Scanner
        scanner.close();
    }
}
```

### Porquê fechar?

Neste caso específico, fechar o Scanner não é obrigatório, mas também não prejudica o funcionamento do programa. A decisão de o fazer ou não é mais uma questão de boa prática e consistência no código.

---

## Melhoria no Fechamento com try-with-resources

A partir do **Java 7**, pode-se usar **try-with-resources** para garantir que o Scanner é fechado automaticamente quando associado a recursos externos.

### Exemplo com ficheiro:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ScannerTryWithResources {
    public static void main(String[] args) throws FileNotFoundException {
        try (Scanner scanner = new Scanner(new File("exemplo.txt"))) {
            while (scanner.hasNextLine()) {
                System.out.println(scanner.nextLine());
            }
        } // O Scanner é fechado automaticamente aqui
    }
}
```

### Vantagens:

- Evita esquecer de fechar o Scanner.
- Torna o código mais limpo e seguro.

## Resumo sobre Fecho de Scanner:

Uso do Scanner	É necessário fechar?	Porquê?
Ligado a <b>System.in</b>	Não recomendado	Fechar o Scanner também fecha <code>System.in</code> , inutilizando-o.
Ligado a um <b>ficheiro</b>	Sim	Liberta o recurso associado ao ficheiro.
Ligado a um <b>InputStream</b>	Sim	Liberta o fluxo de entrada associado.
Ligado a uma <b>String</b>	Não necessário	Não envolve recursos externos.

## Boa Prática

- Use **try-with-resources** (a partir do Java 7) para fechar automaticamente o Scanner quando associado a ficheiros ou streams.

## 7. Notas sobre o operador <> (“*diamond*” ou “diamante”)

### O que é o operador <> no Java?

O operador **diamond** (<>) foi introduzido no **Java 7** como parte das melhorias no sistema de tipos genéricos. Ele permite que o compilador infira o tipo genérico de um objeto a partir do contexto, reduzindo a necessidade de especificar explicitamente o tipo repetidas vezes.

---

### Como o Diamond Funciona?

Antes do Java 7, ao criar um objeto genérico, era necessário especificar o tipo tanto no lado esquerdo quanto no lado direito da declaração:

```
List<String> lista = new ArrayList<String>();
```

Com o **diamond operator** no Java 7 e versões posteriores, o compilador infere automaticamente o tipo no lado direito:

```
List<String> lista = new ArrayList<>();
```

#### Vantagens:

1. Reduz a verbosidade do código.
2. Torna o código mais legível.
3. Evita redundâncias.

---

## Casos de Uso Comuns do Operador < >

### *1. Listas e Conjuntos*

#### **Antes do Java 7:**

```
List<String> nomes = new ArrayList<String>();  
Set<Integer> numeros = new HashSet<Integer>();
```

#### **Com Diamond (Java 7+):**

```
List<String> nomes = new ArrayList<>();  
Set<Integer> numeros = new HashSet<>();
```

### *2. Mapas*

#### **Antes do Java 7:**

```
Map<String, Integer> mapa = new HashMap<String, Integer>();
```

#### **Com Diamond (Java 7+):**

```
Map<String, Integer> mapa = new HashMap<>();
```

---

## Restrições do Diamond

Embora o operador diamond seja poderoso, ele tem algumas restrições importantes:

### *1. Em Variáveis Locais*

O operador diamond funciona perfeitamente para variáveis locais:

```
List<String> lista = new ArrayList<>();
```

### *2. Em Declarações com Subclasses*

Quando se usa uma classe anônima, o operador diamond **não pode ser usado**, porque o compilador não consegue inferir completamente o tipo.

#### **Exemplo (inválido):**

```
List<String> lista = new ArrayList<>() {  
    // Implementação personalizada  
};
```

#### **Forma correta (com tipo explícito):**

```
List<String> lista = new ArrayList<String>() {  
    // Implementação personalizada  
};
```

---

## Melhorias no Java 9 e Posteriores

Com o **Java 9**, o diamond foi estendido para funcionar com tipos genéricos em **classes anónimas**, desde que o tipo possa ser inferido do contexto.

### Exemplo (válido no Java 9+):

```
List<String> lista = new ArrayList<>() {  
    @Override  
    public boolean add(String s) {  
        System.out.println("Adicionado: " + s);  
        return super.add(s);  
    }  
};
```

### Porquê funciona no Java 9?

- O compilador consegue inferir o tipo `String` a partir do lado esquerdo (`List<String>`).

---

## Vantagens do Diamond

### 1. Redução da Verbosidade:

- Facilita a criação de objetos genéricos sem repetir explicitamente os tipos.
- Exemplo:

```
Map<String, List<Integer>> mapa = new HashMap<>();
```

### 2. Menos Propenso a Erros:

- Evita inconsistências entre os tipos do lado esquerdo e direito da declaração.

### 3. Maior Legibilidade:

- O código fica mais limpo e fácil de entender.
- 

## Resumo

Introduzido em	Versão	Funcionalidade
Java 7	Diamond < > para variáveis locais	
Java 9	Suporte para diamond em classes anônimas com tipos inferidos	

### Quando Usar o Diamond?

- Use o operador sempre que possível para reduzir a verbosidade, exceto em casos de **classes anônimas** em versões anteriores ao **Java 9**.



## 8. Java 9 a Java 16: Principais Melhorias

A partir da **versão 9** do Java, a linguagem e a plataforma receberam uma série de melhorias e novas funcionalidades significativas. Aqui estão as **principais funcionalidades introduzidas no Java 9** e em versões subsequentes, organizadas de forma prática e explicada:

---

### Java 9

#### 1. Sistema de Módulos (*Project Jigsaw*)

- Introduziu um sistema de módulos para organizar melhor grandes aplicações e a própria JDK.
- Permite criar aplicações mais leves ao incluir apenas os módulos necessários.

#### Exemplo: Definição de um módulo no ficheiro `module-info.java`

```
module com.exemplo.app {
    requires java.base; // Inclui apenas os módulos necessários
    exports com.exemplo.util; // Torna os pacotes visíveis para outros módulos
}
```

#### Benefícios:

- Melhor encapsulamento.
  - Redução do tamanho da aplicação.
  - Maior controlo sobre dependências.
- 

#### 2. Métodos Privados em Interfaces

- Introduz a possibilidade de ter **métodos privados** em interfaces, para reutilizar código em métodos `default` e `static`.

#### Exemplo:

```
public interface Exemplo {
    default void metodoPublico() {
        metodoPrivado(); // Reutiliza o método privado
    }

    private void metodoPrivado() {
        System.out.println("Método privado na interface.");
    }
}
```

### 3. Coleções *Imutáveis* (*Convenient Factory Methods*)

- Adicionados métodos para criar coleções imutáveis de forma mais simples com `List.of`, `Set.of` e `Map.of`.

#### Exemplo:

```
List<String> lista = List.of("Java", "Python", "C++");
Set<Integer> conjunto = Set.of(1, 2, 3);
Map<String, Integer> mapa = Map.of("A", 1, "B", 2);
```

#### Benefícios:

- Código mais conciso e legível.
  - Garantia de imutabilidade.
- 

### 4. API de Fluxos Aprimorada

- Novos métodos como `takeWhile`, `dropWhile`, e `ofNullable` foram adicionados à API de Streams.

#### Exemplo: `takeWhile`

```
List<Integer> numeros = List.of(1, 2, 3, 4, 5);
numeros.stream()
    .takeWhile(n -> n < 4) // Pára ao encontrar um elemento que não satisfaz a
    condição
    .forEach(System.out::println); // Imprime: 1, 2, 3
```

---

### 5. Process API Melhorada

- Novos métodos para obter informações sobre processos do sistema operativo.

#### Exemplo:

```
ProcessHandle currentProcess = ProcessHandle.current();
System.out.println("PID: " + currentProcess.pid());
System.out.println("Nome do processo: " + currentProcess.info().command().orElse("N/A"));
```

---

### 6. REPL (JShell)

- Ferramenta interativa para testar código Java diretamente no terminal.

#### Exemplo (JShell no terminal):

```
jshell> int x = 5;
jshell> System.out.println(x * 2);
10
```

---

## Java 10

### 1. Inferência de Tipos com var

- Introduzido o uso de var para permitir que o compilador infira o tipo das variáveis locais.

#### Exemplo:

```
var lista = List.of("Java", "Kotlin"); // O tipo é inferido como List<String>
for (var nome : lista) {
    System.out.println(nome);
}
```

---

## Java 11

### 1. Strings Melhoradas

- Novos métodos como lines, strip, repeat, entre outros.

#### Exemplo:

```
String texto = "  Olá, Mundo!  ";
System.out.println(texto.strip()); // Remove espaços: "Olá, Mundo!"
System.out.println("Java".repeat(3)); // Repete a string: "JavaJavaJava"
```

---

### 2. HTTP Client API

- Introdução de uma nova API para fazer chamadas HTTP, mais moderna e fácil de usar.

#### Exemplo:

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://api.example.com"))
    .GET()
    .build();
HttpResponse<String> response = client.send(request,
    HttpResponse.BodyHandlers.ofString());
System.out.println(response.body());
```

---

## Java 12

### 1. *switch Melhorado (Preview)*

- O switch tornou-se mais expressivo e pode ser usado como uma expressão.

#### Exemplo:

```
int dia = 5;
String resultado = switch (dia) {
    case 1, 2, 3, 4, 5 -> "Dia útil";
    case 6, 7 -> "Fim de semana";
    default -> "Desconhecido";
};
System.out.println(resultado); // Saída: Dia útil
```

---

## Java 14

### 1. *Record (Preview)*

- Introdução dos **record**, uma forma concisa de definir classes que contêm apenas dados.

#### Exemplo:

```
public record Ponto(int x, int y) {}
Ponto ponto = new Ponto(10, 20);
System.out.println(ponto.x()); // 10
System.out.println(ponto.y()); // 20
```

---

## Java 15

### 1. *Text Blocks*

- Facilita o trabalho com strings multilinhas.

#### Exemplo:

```
String html = """
    <html>
        <body>
            <h1>Olá, Mundo!</h1>
        </body>
    </html>
    """;
System.out.println(html);
```

---

## Java 16

### *1. Melhorias nos `Stream.toList`*

- Adicionada a funcionalidade para criar listas imutáveis diretamente a partir de Streams.

#### **Exemplo:**

```
List<String> lista = List.of("A", "B", "C").stream()  
    .map(String::toLowerCase)  
    .toList();
```

---

## Resumo das melhorias da versão 9 à 16:

Versão	Funcionalidades Importantes
Java 9	Sistema de módulos, coleções imutáveis, métodos privados em interfaces.
Java 10	Inferência de tipos com <code>var</code> .
Java 11	Melhorias em Strings, HTTP Client API, remoção de código legado.
Java 12	<code>switch</code> melhorado (preview).
Java 14	Introdução dos record (preview).
Java 15	Suporte a blocos de texto ( <code>text blocks</code> ).
Java 16	Melhorias nos Streams.

## 9. Melhorias das Versões 17 a 21 da Linguagem Java:

### Java 17

Lançado em setembro de 2021, o Java 17 é uma versão de Suporte de Longo Prazo (LTS), garantindo atualizações e suporte estendido. As principais funcionalidades introduzidas incluem:

#### 1. Classes Seladas (Sealed Classes)

As **classes seladas** permitem restringir quais outras classes ou interfaces podem estender ou implementar uma classe ou interface específica, proporcionando maior controle sobre hierarquias de classes.

##### Exemplo:

```
public sealed class Forma permits Circulo, Quadrado, Retangulo {}

public final class Circulo extends Forma {}
public final class Quadrado extends Forma {}
public final class Retangulo extends Forma {}
```

Neste exemplo, apenas Circulo, Quadrado e Retangulo podem estender Forma.

---

#### 2. Correspondência de Padrões para switch (Preview)

Esta funcionalidade permite o uso de correspondência de padrões no operador switch, tornando o código mais conciso e legível.

##### Exemplo:

```
static String formatar(Object obj) {
    return switch (obj) {
        case Integer i -> String.format("Número inteiro: %d", i);
        case String s -> String.format("Texto: %s", s);
        default -> "Tipo desconhecido";
    };
}
```

Aqui, o switch avalia o tipo do objeto e executa a ação correspondente.

---

#### 3. Encapsulamento Forte dos Internos do JDK

O Java 17 reforça o encapsulamento das APIs internas do JDK, melhorando a segurança e a manutenção do código.

---

#### ***4. Remoção da Ativação RMI***

O mecanismo de ativação RMI foi removido, simplificando a plataforma e eliminando funcionalidades obsoletas.

---

#### ***5. Filtros de Desserialização Específicos de Contexto***

Introduzidos para melhorar a segurança durante o processo de desserialização de objetos, permitindo maior controlo sobre quais classes podem ser desserializadas.



---

## Java 18

Lançado em março de 2022, o Java 18 trouxe melhorias incrementais, incluindo:

### *1. API de Funções Estrangeiras e Memória (Incubadora)*

Esta API permite que programas Java interajam com código e memória fora da JVM de forma segura e eficiente, facilitando a integração com bibliotecas nativas.

---

### *2. Servidor Web Simples*

Introduz um servidor HTTP minimalista, útil para testes e desenvolvimento rápido sem a necessidade de configurações complexas.

#### **Exemplo:**

```
var servidor = HttpServer.create(new InetSocketAddress(8080), 0);
servidor.createContext("/saudacao", troca -> {
    String resposta = "Olá, Mundo!";
    troca.sendResponseHeaders(200, resposta.getBytes().length);
    try (OutputStream os = troca.getResponseBody()) {
        os.write(resposta.getBytes());
    }
});
servidor.start();
```

---

## Java 19

Lançado em setembro de 2022, o Java 19 continuou a introduzir melhorias, como:

### *1. Threads Virtuais (Preview)*

As **threads virtuais** facilitam a criação e gestão de threads leves, melhorando a escalabilidade de aplicações simultâneas.

#### **Exemplo:**

```
try (var escopo = new StructuredTaskScope.ShutdownOnFailure()) {
    Future<String> resultado = escopo.fork(() -> {
        // Tarefa intensiva
        return "Resultado";
    });
    escopo.join();
    System.out.println(resultado.resultNow());
}
```

---

### *2. API de Funções Estrangeiras e Memória (Segunda Incubadora)*

Continuação do desenvolvimento da API para interagir com código e memória fora da JVM, com melhorias baseadas em feedback.

---

## Java 20

Lançado em março de 2023, o Java 20 trouxe:

### *1. Correspondência de Padrões para switch (Segunda Preview)*

Aprimoramentos na correspondência de padrões, permitindo casos mais complexos e maior expressividade no uso do switch.

---

### *2. Threads Virtuais (Segunda Preview)*

Continuação dos testes e refinamentos das threads virtuais antes de sua estabilização.

---

## Java 21

Lançado em setembro de 2023, o Java 21 introduziu:

### *1. Correspondência de Padrões para `switch` (Feature Completa)*

A funcionalidade foi estabilizada, permitindo seu uso em produção.

---

### *2. Threads Virtuais (Feature Completa)*

As threads virtuais foram finalizadas e estão prontas para uso em produção, facilitando a construção de aplicações simultâneas escaláveis.

---

## Resumo das Funcionalidades das Versão 17 a 21:

Versão	Funcionalidades Principais
<b>Java 17</b>	Classes Seladas, Correspondência de Padrões para <code>switch</code> (Preview), Encapsulamento Forte dos Internos do JDK
<b>Java 18</b>	API de Funções Estrangeiras e Memória (Incubadora), Servidor Web Simples
<b>Java 19</b>	Threads Virtuais (Preview), API de Funções Estrangeiras e Memória (Segunda Incubadora)
<b>Java 20</b>	Correspondência de Padrões para <code>switch</code> (Segunda Preview), Threads Virtuais (Segunda Preview)
<b>Java 21</b>	Correspondência de Padrões para <code>switch</code> (Feature Completa), Threads Virtuais (Feature Completa)

Estas atualizações refletem o compromisso contínuo da comunidade Java em melhorar a linguagem, tornando-a mais expressiva, eficiente e alinhada com as necessidades modernas de desenvolvimento de software.

## Funcionalidades Estabilizadas a Partir do Java 17:

A partir do **Java 17**, várias funcionalidades que estavam em fase de pré-visualização ou incubação foram finalizadas e estão agora disponíveis para uso em produção. Aqui estão algumas das principais funcionalidades que se tornaram estáveis, juntamente com exemplos práticos:

---

### 1. Classes Seladas (Sealed Classes)

As **classes seladas** permitem restringir quais classes podem estender ou implementar uma determinada classe ou interface, proporcionando um maior controlo sobre a hierarquia de classes.

**Exemplo:**

```
public sealed class Forma permits Circulo, Quadrado {  
    // Membros da classe  
}  
  
public final class Circulo extends Forma {  
    // Implementação específica de Circulo  
}  
  
public final class Quadrado extends Forma {  
    // Implementação específica de Quadrado  
}
```

Neste exemplo, apenas `Circulo` e `Quadrado` podem estender a classe `Forma`.

---

## 2. Correspondência de Padrões para switch (Pattern Matching for switch)

Esta funcionalidade permite que o switch avalie não apenas valores exatos, mas também padrões, tornando o código mais conciso e expressivo.

### Exemplo:

```
public String processarValor(Object valor) {  
    return switch (valor) {  
        case Integer i -> "Número inteiro: " + i;  
        case String s -> "Texto: " + s;  
        case null -> "Valor nulo";  
        default -> "Tipo desconhecido";  
    };  
}
```

Aqui, o switch adapta-se ao tipo do objeto valor e executa a lógica correspondente.

---

### 3. Blocos de Texto (Text Blocks)

Os **blocos de texto** facilitam a definição de strings multilinha, melhorando a legibilidade e manutenção do código.

#### Exemplo:

```
String json = """
    {
        "nome": "João",
        "idade": 30,
        "cidade": "Lisboa"
    }
    """;
```

Este bloco de texto define uma string JSON de forma clara e legível.



---

## 4. Records

Os **records** fornecem uma forma concisa de declarar classes imutáveis que são principalmente portadoras de dados.

### Exemplo:

```
public record Ponto(int x, int y) {  
    // Métodos adicionais podem ser definidos, se necessário  
}
```

```
Ponto ponto = new Ponto(5, 10);  
System.out.println(ponto.x()); // Saída: 5  
System.out.println(ponto.y()); // Saída: 10
```

Aqui, Ponto é um record que encapsula x e y como campos imutáveis.

---

## 5. Inferência de Tipos com var

A palavra-chave `var` permite que o compilador infira o tipo da variável com base no valor atribuído, reduzindo a verbosidade do código.

### Exemplo:

```
var lista = List.of("Maçã", "Banana", "Laranja");  
for (var fruta : lista) {  
    System.out.println(fruta);  
}
```

O compilador infere que `lista` é do tipo `List<String>`.

---

## 6. API de Cliente HTTP

A nova API de cliente HTTP facilita a realização de solicitações HTTP de forma síncrona ou assíncrona.

### Exemplo:

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://api.exemplo.com/dados"))
    .build();
```

```
HttpResponse<String> response = client.send(request,
    HttpResponse.BodyHandlers.ofString());
System.out.println(response.body());
```

Este código realiza uma solicitação GET para a URL especificada e imprime a resposta.

---

## 7. API de Funções Estrangeiras e Memória (Foreign Function & Memory API)

Esta API permite que programas Java interajam de forma segura e eficiente com código e memória fora da JVM.

### Exemplo:

```
try (var segmento = MemorySegment.allocateNative(100)) {  
    MemoryAccess.setIntAtOffset(segmento, 0, 42);  
    int valor = MemoryAccess.getIntAtOffset(segmento, 0);  
    System.out.println(valor); // Saída: 42  
}
```

Este exemplo aloca um segmento de memória nativa, define um valor inteiro e, em seguida, lê esse valor.

## 10. Inferência de Tipos com var

A partir do **Java 10**, a palavra-chave **var** foi introduzida para permitir a **inferência de tipos** em variáveis locais, tornando o código mais conciso e legível. O compilador infere o tipo da variável com base no valor atribuído durante a sua inicialização.

### Exemplos Práticos do Uso de var

#### 1. Declaração Simples de Variáveis

Em vez de declarar explicitamente o tipo de uma variável, pode-se utilizar `var` para que o compilador infira o tipo:

```
var mensagem = "Olá, Mundo!"; // Inferido como String
var numero = 42;                // Inferido como int
var decimal = 3.14;             // Inferido como double
var ativo = true;               // Inferido como boolean
```

Neste exemplo, o compilador determina automaticamente os tipos `String`, `int`, `double` e `boolean` com base nos valores atribuídos.

#### 2. Uso em Estruturas de Controle

O `var` pode ser utilizado em loops, facilitando a leitura do código:

```
var numeros = List.of(1, 2, 3, 4, 5);

for (var numero : numeros) {
    System.out.println(numero);
}
```

Aqui, `numeros` é inferido como `List<Integer>`, e `numero` como `Integer`.

### 3. *Uso com Mapas*

Ao lidar com mapas, o uso de `var` pode simplificar a sintaxe:

```
var capitais = Map.of(
    "Portugal", "Lisboa",
    "Espanha", "Madrid",
    "França", "Paris"
);

for (var entrada : capitais.entrySet()) {
    var pais = entrada.getKey();
    var cidade = entrada.getValue();
    System.out.println(pais + " - " + cidade);
}
```

Neste caso, `capitais` é inferido como `Map<String, String>`, `entrada` como `Map.Entry<String, String>`, `pais` e `cidade` como `String`.

### 4. *Operações com Streams*

O `var` também pode ser útil ao trabalhar com a API de Streams:

```
var nomes = List.of("Ana", "Bruno", "Carlos");

var nomesFiltrados = nomes.stream()
    .filter(nome -> nome.startsWith("B"))
    .collect(Collectors.toList());

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

Aqui, `nomesFiltrados` é inferido como `List<String>`.

## Considerações Importantes sobre uso de var

- **Inicialização Obrigatória:** A variável declarada com var deve ser inicializada no momento da declaração para que o compilador possa inferir o tipo.

```
var contador; // Erro: variável não inicializada
```

- **Uso Restrito a Variáveis Locais:** O var só pode ser utilizado para variáveis locais dentro de métodos, construtores ou blocos. Não é permitido em variáveis de instância, variáveis de classe ou parâmetros de métodos.

```
public class Exemplo {  
    var atributo = "Valor"; // Erro: não permitido fora de um método  
  
    public void metodo(var parametro) { // Erro: não permitido em parâmetros  
        // ...  
    }  
}
```

- **Clareza do Código:** Embora o var torne o código mais conciso, deve-se garantir que o tipo da variável seja evidente a partir do contexto para manter a legibilidade.

```
var resultado = calcular(); // Que tipo é 'resultado'? Pode não ser claro
```

## Vantagens do Uso de var

- **Redução de Verbosidade:** Elimina a necessidade de declarações de tipos longas e repetitivas.

```
Map<String, List<Integer>> mapa = new HashMap<>(); // Sem 'var'
var mapa = new HashMap<String, List<Integer>>();   // Com 'var'
```

- **Facilidade de Manutenção:** Alterações no tipo de uma variável requerem mudanças em menos locais do código.

## Limitações do uso de var

- **Não Pode Ser Usado sem Inicialização:** O var requer uma expressão de inicialização para que o tipo possa ser inferido.

```
var valor = null; // Erro: não é possível inferir o tipo a partir de 'null'
```

- **Não Pode Ser Usado para Tipos Anônimos:** O var não pode ser utilizado para declarar variáveis de tipos anônimos diretamente.

```
var objeto = new Object() {
    String nome = "Exemplo";
}; // Erro: não é possível inferir o tipo de um tipo anônimo
```

## Conclusão

O uso de var no Java moderno permite escrever código mais limpo e conciso, aproveitando a inferência de tipos do compilador. No entanto, é fundamental utilizá-lo de forma consciente, garantindo que a legibilidade e a clareza do código sejam mantidas.