



Inovações da versão 8 a 21

Luís Cunha, 2025



## Bem-vindo/a ao Universo Java Moderno!

Se já programaste em Java, sabes que é uma linguagem robusta, segura e usada em tudo — de aplicações bancárias a videojogos, passando por sistemas empresariais e apps Android. Mas sabias que desde o Java 8, esta linguagem clássica passou por uma revolução silenciosa?

### Este manual é o teu mapa para explorar o Java moderno — da versão 8 até à 21!

Nos últimos anos, o Java evoluiu como nunca: ganhou Expressões Lambda, Streams, a maravilhosa classe `Optional`, referências a métodos e até blocos de texto com aspeto de poesia para o teu código! 😊 E o melhor? Tudo isto veio para tornar o teu código mais **limpo, expressivo e eficiente**.

---

### Porquê este manual?

Porque muitos programadores sabem "Java Clássico", mas não acompanharam a sua evolução. E isso é uma pena, porque hoje podes fazer **muito mais com muito menos código** — e com muito mais elegância! 💡

Este manual é para ti se:

- Já sabes programar em Java, mas queres atualizar-te sem ler 1000 páginas.
  - Adoras exemplos simples, diretos e fáceis de testar.
  - Queres tornar o teu código mais próximo da **programação funcional** (sem perder o toque clássico do Java).
- 

### O que vais encontrar aqui?

Cada capítulo foi pensado como uma mini-aula prática, com:

-  Explicações diretas e sem rodeios.
  -  Exemplos reais e testáveis.
  -  Comparações com a forma antiga de fazer as coisas.
  -  Casos de uso do mundo real.
  -  Resumos visuais para memorizares facilmente os conceitos.
- 

### Pronto para a viagem?

Vamos começar pela porta de entrada para o Java moderno: as **Expressões Lambda** — uma forma elegante de dar vida a funções sem nome. Depois, vais explorar o poder dos

**Streams**, o conforto do `Optional`, e muitas outras funcionalidades que farão o teu código parecer magia! 🌟

🧭 A bússola está pronta. O repositório de exemplos está no GitHub. Vamos juntos nesta jornada do Java clássico ao Java moderno — passo a passo, com estilo e sem complicações.

👉 [Repositório de exemplos no GitHub](#)

## ⌚ Porquê Este Manual?

(Ou: Porque é que o Java mudou tanto... e tu mereces acompanhar essa evolução!) 🚀

O Java é como aquele amigo de infância que continua presente nas nossas vidas — fiável, seguro e sempre pronto para ajudar. Mas atenção: **esse amigo cresceu e modernizou-se!**

💻 Desde o Java 8, a linguagem deu um salto gigantesco. Deixou de ser apenas “orientada a objetos” para abraçar **conceitos funcionais**, melhorar a clareza do código e simplificar o que antes era... bem, demasiado verboso 😅

---

## 🌟 O Problema:

Muitos programadores que já dominam a base do Java não acompanharam esta transformação. O resultado?

- ✗ Continuam a escrever código como se estivessem em 2005.
  - ✓ Quando podiam estar a usar recursos como `Stream`, `Optional`, `var`, `records`, `switch` inteligente e muito mais!
- 

## ⌚ O Objetivo Deste Manual:

Ajudar-te a dar o salto para o **Java moderno** — de forma simples, prática e bem explicada.

Nada de linguagem pomposa.

Nada de teoria seca.

Apenas explicações diretas e exemplos com propósito. 💡

## 🔍 O Que Vais Encontrar?

- ✓ Explicações claras: o "como" e o "porquê" de cada funcionalidade.
  - ✓ Exemplos práticos, prontos a testar e aplicar no teu trabalho real.
  - ✓ Comparações entre o **antes** e o **agora**, para veres a diferença com os teus próprios olhos.
  - ✓ Um repositório no GitHub com todos os exemplos prontos a experimentar:  
👉 <https://github.com/luiscunhacsc/java-features-from-v8-to-now>
- 

## ✳️ Para Quem É Este Manual?

- Para quem aprendeu Java há uns anos... e quer agora dar um refresh.
  - Para quem quer escrever menos e fazer mais.
  - Para quem gosta de aprender por exemplo.
  - Para quem quer preparar-se para os desafios do Java moderno (desde entrevistas técnicas até novos projetos).
- 

💡 Em suma: este manual não é só uma lista de novidades. É um **convite para pensares o Java com olhos novos** — e usá-lo de forma mais elegante, moderna e produtiva.

Pronto para começar a escrever **menos código e com mais poder**?

Vamos a isso! 🚀💻

# 🧠 Capítulo 1 — Expressões Lambda no Java💡

## Uma nova forma de pensar o código!

“Menos código. Mais clareza. E tudo com estilo funcional.”

---

### 🔗 O que são Lambdas?

As **Expressões Lambda** chegaram com o Java 8 para dar uma volta ao estilo de programação tradicional. Imagina poderes escrever funções **sem nome**, diretamente no local onde precisas delas. É como ter superpoderes de concisão!

(a, b) -> a + b

Isto soma dois números. Simples, não é? 😎

Este pequeno truque reduz toneladas de código redundante e torna tudo mais limpo e direto.

---

### 📦 Estrutura de uma Lambda

(parâmetros) -> { corpo da função }

- ◆ **Parâmetros:** os inputs da função.
- ◆ **Seta (->):** separa os parâmetros do que vai ser feito.
- ◆ **Corpo da função:** a lógica — pode ser uma linha ou um bloco completo.

*Exemplos:*

```
// Uma linha  
x -> x * x
```

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

---

### ⌚ Porque é que Lambdas são incríveis?

- ✓ **Menos código:** substituem as antigas classes anónimas.
  - ✓ **Mais legibilidade:** foca-te no que a função faz, não no formato.
  - ✓ **Funciona com Streams:** lambdas + streams = dupla imbatível!
-

## Onde posso usar Lambdas?

Sempre que precisares de uma **interface funcional** (ou seja, com um só método). Exemplos famosos:

- `Runnable`
  - `Comparator<T>`
  - `Predicate<T>`
  - `Function<T, R>`
  - `Consumer<T>`
- 

## Exemplos práticos

### Criar uma thread (antes e depois do Java 8):

```
// Antes:  
Runnable tarefa = new Runnable() {  
    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();
```

### Comparar dois números:

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

### Verificar se um número é par:

```
Predicate<Integer> isPar = n -> n % 2 == 0;
```

---

## Como dominar as Lambdas (em 3 passos):

1. **Identifica uma interface funcional** que se aplica ao teu caso.
  2. **Define o comportamento** que queres — com `->`.
  3. **Usa a expressão lambda** em vez da classe anónima.
- 

## Exemplo prático completo

```
List<String> nomes = Arrays.asList("Ana", "João", "Maria", "Pedro");  
  
nomes.stream()  
    .filter(nome -> nome.startsWith("A")) // só nomes que começam com
```

```
"A"  
.map(nome -> nome.toUpperCase())           // para maiúsculas  
.forEach(System.out::println);             // imprime: ANA
```

---

## Resumo

### Expressões Lambda:

- Tornam o código mais curto e elegante.
  - Funcionam onde houver uma **interface funcional**.
  - São essenciais para tirar partido de APIs modernas como **Streams**.
- 

Pronto para dar o salto do Java clássico para o Java expressivo?  
No próximo capítulo vamos explorar os **Streams**, que se combinam na perfeição com lambdas para processar dados de forma mágica! 

### Vamos nessa?

---

# 💧 Capítulo 2 — A Maravilhosa API de Streams no Java

Trabalhar com coleções... como um verdadeiro mestre Jedi! 

---

## 🦉 Porque é que precisas de Streams?

Imagina que tens uma lista de dados (números, nomes, produtos...) e queres:

- Filtrar só os pares?
- Calcular o quadrado de cada número?
- Ordenar nomes alfabeticamente?
- Somar todos os preços?

Com a API de Streams, tudo isto fica:  Mais simples

 Mais elegante

 Mais expressivo

---

## ▀ O que é um Stream?

Um **Stream** é uma visão fluída sobre uma coleção. Não guarda dados — **processa-os**. Pensa num **pipeline de água**, onde cada gota (elemento) passa por filtros e transformações até chegar ao resultado final 

---

## ⌚ Como funciona?

### 1. Origem dos dados

Ex: lista, array, ficheiro...

### 2. Operações Intermediárias

 Transformam o stream, mas **não o consomem** (ex: filter, map, distinct...)

### 3. Operação Terminal

 Executa o stream e devolve um resultado (ex: collect, forEach, reduce...)

---

## ❖ Exemplo simples

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

List<Integer> quadradosPares = numeros.stream()
    .filter(n -> n % 2 == 0)      // Filtra os pares
    .map(n -> n * n)            // Calcula o quadrado
    .collect(Collectors.toList()); // Converte em lista

System.out.println(quadradosPares); // [4, 16]
```

---

## 🔍 Operações mais comuns

### 👉 Intermidiárias:

Operação	O que faz
filter	Filtrar elementos
map	Transforma elementos
distinct	Remove duplicados
sorted	Ordena os dados
limit	Limita o número de elementos
skip	Salta os primeiros n elementos

### ⌚ Terminais:

Operação	O que faz
forEach	Executa ação para cada elemento
collect	Junta os elementos numa coleção
reduce	Reduz tudo a um único valor
count	Conta os elementos
findFirst	Devolve o primeiro (se existir)

## 💻 Exemplo visual e completo

```
List<String> palavras = Arrays.asList("Java", "Stream", "API", "Lambda",
                                         "Java");

List<String> resultado = palavras.stream()
    .distinct()                  // Remove duplicados
    .map(String::toUpperCase)   // Para maiúsculas
    .sorted()                    // Ordena
    .collect(Collectors.toList());

System.out.println(resultado); // [API, JAVA, LAMBDA, STREAM]
```

---

## Streams são preguiçosos!

As operações só são **executadas de verdade** quando usas uma operação **terminal**. Até lá, estão apenas "em espera".

```
stream().filter(...).map(...) // Nada acontece ainda!
```

Só com `.collect()` ou `.forEach()` é que tudo acontece 

---

## Streams Paralelos (para os pros )

```
numeros.parallelStream()  
    .map(n -> n * 2)  
    .forEach(System.out::println); // ⚠️ Ordem não garantida
```

Usa todos os núcleos da CPU. Ideal para grandes volumes de dados!  
Mas cuidado com efeitos colaterais e dependências de ordem.

---

## Resumo Final

- 🔥 Streams = forma moderna de processar coleções.
  - ⚡ Códigos mais curtos, expressivos e funcionais.
  - ✅ Perfeitos para map/filter/reduce.
  - ↗ Operações intermediárias são encadeáveis.
  - 🙏 Imutável: não altera a lista original.
-

# 🔗 Capítulo 3 — Referências a Métodos e Construtores

A forma mais elegante de escrever código limpo e reutilizável ✨

---

## 💡 O que são?

As **referências a métodos** são como atalhos para dizer ao Java:

“Usa *aquele método ali...* já está tudo feito!”

São uma alternativa ainda mais concisa às Expressões Lambda.

Quando uma lambda apenas chama um método já existente, podemos escrever algo mais limpo:

```
// Lambda tradicional  
nomes.forEach(nome -> System.out.println(nome));  
  
// Com referência a método  
nomes.forEach(System.out::println);
```

📌 **Mais curto. Mais limpo. E com o mesmo poder.**

---

## 🔍 Tipos de Referências

Tipo	Sintaxe	Exemplo	Equivalente Lambda
Método Estático	Classe::método	Math::abs	n -> Math.abs(n)
Método de instância (objeto específico)	objeto::método	sb::append	s -> sb.append(s)
Método de instância (objeto arbitrário)	Classe::método	String::toUpperCase	s -> s.toUpperCase()
Construtor	Classe::new	Produto::new	nome -> new Produto(nome)

---

## 💡 Exemplos Práticos

### 1. 📋 Método Estático

```
List<Integer> numeros = Arrays.asList(-10, -20, 30, 40);  
numeros.stream()  
    .map(Math::abs) // Usa o método estático Math.abs  
    .forEach(System.out::println); // → 10, 20, 30, 40
```

---

## 2. Método de Instância de um Objeto Específico

```
StringBuilder sb = new StringBuilder();
List<String> palavras = Arrays.asList("Java", "Stream", "Lambda");

palavras.forEach(sb::append);
System.out.println(sb.toString()); // JavaStreamLambda
```

---

## 3. Método de Instância de um Objeto Arbitrário

```
List<String> nomes = Arrays.asList("ana", "joão", "maria");

nomes.stream()
    .map(String::toUpperCase) // Cada String chama o seu toUpperCase
    .forEach(System.out::println); // ANA, JOÃO, MARIA
```

---

## 4. Referência a Construtor

```
List<String> nomes = Arrays.asList("Caneta", "Caderno", "Borracha");

List<Produto> produtos = nomes.stream()
    .map(Produto::new) // Cria Produto(nome)
    .toList();

produtos.forEach(p -> System.out.println(p.nome));
```

---

## Quando usar?

- Quando uma lambda apenas chama um método.
  - Quando queres deixar o código mais **legível e direto**.
  - Quando estás a trabalhar com `map`, `forEach`, `filter`, `collect`...
- 

## Dicas Rápidas

- Usa lambdas quando precisas de lógica personalizada.
  - Usa referências a métodos **quando já tens um método pronto** e o queres reaproveitar.
  - Se estiveres na dúvida, **começa com lambda e depois substitui pela referência** se fizer sentido.
-

## ❖ Resumo Final

- **Referências a métodos** são uma forma elegante de simplificar lambdas repetitivas.
  - Ajudam a manter o código mais **limpo, expressivo e conciso**.
  - Existem 4 tipos: método estático, de instância (objeto específico ou arbitrário) e construtor.
  - Funcionam perfeitamente com **Streams e programação funcional** moderna.
- 

👉 No próximo capítulo vamos explorar uma ferramenta indispensável no Java moderno para evitar null e NullPointerException: a poderosa e elegante classe **Optional!** Vem descobrir como ela pode tornar o teu código mais seguro e legível. ❌ 🎉

# Capítulo 4 — `Optional`: Adeus aos NullPointerException!

Uma caixinha segura para os teus valores  

---

## O problema do `null`...

Quantas vezes já te deparaste com o temido erro `NullPointerException`?

```
String nome = null;  
System.out.println(nome.length()); // BOOM! *
```

O Java 8 trouxe uma solução elegante e segura: a classe `Optional`.

Ela funciona como uma **caixa que pode ter um valor... ou estar vazia**. Mas ao contrário de `null`, essa caixa é **segura** e cheia de métodos para lidar com a ausência de valores de forma clara e previsível 

---

## O que é o `Optional`?

`Optional<T>` é um **contentor genérico** que pode conter:

- Um valor do tipo `T`
- Ou... nenhum valor (está vazio)

Com ele, em vez de testares se algo é `null`, pergunta:

“Esta caixa tem algo dentro? Se sim, uso-o. Se não, faço outra coisa.”

---

## Como criar `Optionals`?

```
Optional<String> nome1 = Optional.of("Java");           // Valor presente  
Optional<String> nome2 = Optional.ofNullable(null);    // Pode estar vazio  
Optional<String> nome3 = Optional.empty();             // Explicitamente  
vazio
```

---

## Métodos mais úteis

Método	O que faz
<code>isPresent()</code>	Diz se há um valor dentro da caixa
<code>isEmpty()</code>	Diz se a caixa está vazia (Java 11+)
<code>get()</code>	Devolve o valor (  cuidado: lança exceção se

Método	O que faz
	estiver vazio!)
<code>orElse(valor)</code>	Usa um valor alternativo se estiver vazio
<code>orElseGet(fornecedor)</code>	Gera um valor alternativo com uma função
<code>orElseThrow(exceção)</code>	Lança exceção personalizada se estiver vazio
<code>ifPresent(ação)</code>	Executa algo se o valor existir
<code>ifPresentOrElse(ação, alternativa)</code>	Executa uma de duas ações dependendo da presença do valor
<code>map(...)</code>	Transforma o valor se estiver presente
<code>flatMap(...)</code>	Idem, mas evita <code>Optional&lt;Optional&lt;T&gt;&gt;</code>

---

### 💡 Exemplo visual

```
Optional<String> nome = Optional.of("Joana");

nome.isPresent(valor -> System.out.println("Nome: " + valor));
// Saída: Nome: Joana
```

E se o nome for ausente?

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

System.out.println(nome.orElse("Sem nome"));
// Saída: Sem nome
```

---

### 🔗 Transformar valores com map

```
Optional<String> nome = Optional.of("joão");

Optional<String> maiusculas = nome.map(String::toUpperCase);

System.out.println(maiusculas.orElse("SEM NOME")); // JOÃO
```

---

### 📝 Exemplo prático completo

```
public static Optional<String> encontrarNome(String id) {
    if ("123".equals(id)) return Optional.of("Maria Silva");
    return Optional.empty();
}

Optional<String> resultado = encontrarNome("123");

resultado.ifPresentOrElse(
```

```
    nome -> System.out.println("Nome encontrado: " + nome),  
    () -> System.out.println("Nome não encontrado")  
);
```

---

### Quando usar **Optional**?

- Quando o resultado **pode ou não existir** (ex: busca numa base de dados).
  - Para evitar testes manuais com `null`.
  - Para deixar claro no teu código: **este valor pode não estar presente**.
  - Para encadear chamadas de forma funcional, segura e legível.
- 

### Resumo Final

-  **Optional** é o substituto moderno, seguro e expressivo do uso do `null`.
  -  Permite lidar com ausência de valor sem riscos de exceção.
  -  Tem métodos práticos para transformar, verificar e fornecer alternativas.
  -  Ideal para código mais limpo, funcional e resiliente.
- 

 No próximo capítulo, vamos explorar **as melhorias nas coleções em Java 8** — como `forEach`, `removeIf`, `replaceAll` e criação de coleções imutáveis com apenas uma linha!



# Capítulo 5 — Novidades nas Coleções no Java 8

Trabalhar com listas e mapas ficou (muito) mais fácil! ♦♦

---

## Porquê mudar?

Antes do Java 8, manipular coleções como listas e mapas era funcional... mas, convenhamos, um pouco **verbooso**. Tarefas simples exigiam ciclos `for` e `if` repetitivos. A nova API veio dar um sopro de ar fresco: **métodos mais expressivos, seguros e concisos** para lidar com coleções.

---

## O que há de novo?

O Java 8 adicionou vários **métodos poderosos** diretamente às classes de coleção como `List`, `Set` e `Map`. Vamos ver os principais destaques:

---

### 1. `forEach` — Iteração moderna

Substitui o tradicional `for` com uma lambda expressiva:

```
List<String> nomes = List.of("Ana", "João", "Maria");  
  
nomes.forEach(nome -> System.out.println("Olá, " + nome + "!));
```

 Também funciona com mapas:

```
Map<String, Integer> idades = Map.of("Ana", 25, "João", 30);  
idades.forEach((nome, idade) -> System.out.println(nome + " tem " + idade  
+ " anos."));
```

---

### 2. `removeIf` — Limpeza com estilo

Permite remover elementos com base numa condição (Predicate).

```
List<Integer> numeros = new ArrayList<>(List.of(1, 2, 3, 4, 5));  
  
numeros.removeIf(n -> n % 2 == 0); // Remove os pares  
System.out.println(numeros); // [1, 3, 5]
```

---

## 3. replaceAll — Transformação direta

Altera todos os elementos da lista com base numa função.

```
List<String> nomes = new ArrayList<>(List.of("ana", "joão", "maria"));

nomes.replaceAll(String::toUpperCase);
System.out.println(nomes); // [ANA, JOÃO, MARIA]
```

---

## 4. Métodos novos em Map

`computeIfAbsent` — Cria valores apenas se forem necessários

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

mapa.computeIfAbsent("erro", chave -> new ArrayList<>()).add("Falha de
rede");
System.out.println(mapa); // {erro=[Falha de rede]}
```

Outros métodos úteis:

- `getOrDefault(chave, valorPadrão)`
  - `compute, merge, replaceAll`
- 

## 5. Coleções Imutáveis — Criadas com um só método

Antes do Java 9, era chato criar listas imutáveis. Agora é fácil:

```
List<String> lista = List.of("A", "B", "C");
Set<Integer> conjunto = Set.of(1, 2, 3);
Map<String, String> mapa = Map.of("PT", "Portugal", "ES", "Espanha");
```

 Estas coleções são **imutáveis** — não se pode adicionar ou remover elementos.

---

## Exemplo completo com tudo:

```
List<String> nomes = new ArrayList<>(List.of("ana", "joão", "pedro",
"ana"));

nomes.removeIf(nome -> nome.equalsIgnoreCase("ana"));
nomes.replaceAll(String::toUpperCase);
nomes.forEach(System.out::println); // JOÃO, PEDRO
```

---

## Resumo Final

💡 Java 8 trouxe **métodos novos para coleções** que:

- Tornam o código mais **curto e expressivo**
  - Permitem **filtrar, transformar e iterar** com elegância
  - Facilitam o uso de **lambdas** no dia a dia
  - Introduzem **coleções imutáveis** simples e seguras
-

# 💡 Capítulo 6 — É Mesmo Preciso Fechar o Scanner?

Evita o aviso chato... mas sem exagerar! ⚠️ 💬

---

## 🧠 A dúvida clássica

```
Scanner sc = new Scanner(System.in);  
// ... usa o Scanner ...
```

💡 Deva usar `sc.close()`; no fim?  
Se sim, porquê? Se não, porquê recebo um warning?

---

## ⌚ Contexto

O Scanner é uma ferramenta super útil para ler dados do teclado, ficheiros, etc. Mas, como qualquer recurso que accede ao **sistema operativo** (streams, sockets, ficheiros...), ele **ocupa recursos** e deve ser fechado... em certos casos! 🚫

---

## 🔍 Casos em que deves fechar o Scanner

- ✓ Quando estás a **ler de um ficheiro**:

```
try (Scanner sc = new Scanner(new File("dados.txt"))) {  
    while (sc.hasNextLine()) {  
        System.out.println(sc.nextLine());  
    }  
} // O Scanner é fechado automaticamente aqui!
```

👉 Usa `try-with-resources`! É mais limpo e evita esquecimentos.

---

## ✗ Casos em que **não** deves fechá-lo (apesar do warning)

- ⚠️ Se estiveres a usar `Scanner(System.in)`, **não o feches**!

```
Scanner sc = new Scanner(System.in);  
String nome = sc.nextLine();  
sc.close(); // ⚠️ Pode fechar prematuramente o System.in!
```

🚫 Ao fazer isto, podes **impedir outras partes do programa** de voltar a ler do teclado.

---

## O que o compilador te está a dizer?

O aviso do tipo:

```
Resource leak: 'sc' is never closed
```

Significa: “Estás a criar um recurso que devias libertar... tem cuidado!”

Mas atenção: o compilador **não sabe se estás a ler de System.in ou de um ficheiro**. Por isso avisa por defeito. 

---

## A solução elegante

Se usares `System.in` e quiseres evitar o warning **sem fechar o Scanner**, declara-o como `final` e documenta a intenção:

```
@SuppressWarnings("resource") // Suprime o aviso do compilador
final Scanner sc = new Scanner(System.in);
```

Ou encapsula a lógica toda numa classe utilitária e documenta que o Scanner não deve ser fechado.

---

## Boa prática com try-with-resources

Sempre que leres de ficheiros ou sockets, usa o padrão moderno:

```
try (Scanner sc = new Scanner(new File("dados.txt"))) {
    // Usa o Scanner normalmente
}
```

 O compilador garante o fecho automático, mesmo que haja exceções.

---

## Resumo

- Scanner deve ser fechado se ler de ficheiros ou sockets.
  - **Nunca feches Scanner(System.in)** diretamente — isso fecha o teclado para sempre no programa! 
  - Usa `try-with-resources` sempre que possível.
  - Suprime avisos com `@SuppressWarnings("resource")` quando tiveres boas razões para não fechar.
-

# 💎 Capítulo 7 — O Operador "Diamond" (<>)

O pequeno símbolo que torna o teu código mais limpo e elegante!

---

## 💡 O que é isso do "Diamond"?

O operador <>, introduzido no Java 7 e melhorado no Java 8, é conhecido como o **operador “diamond”** (ou **losango**, em bom português PT). Ele serve para **simplificar a criação de objetos com genéricos**, evitando a repetição desnecessária de tipos.

---

## 📦 Antes do Diamond

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

⚠️ Tanta repetição! Já dissemos à esquerda que o tipo é Map<String, List<Integer>>, então por que repetir tudo à direita?

---

## ✅ Com o Diamond

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

⭐ Muito mais limpo! O compilador **infere os tipos automaticamente** com base no lado esquerdo.

---

## 🔧 Casos comuns de uso

```
List<String> nomes = new ArrayList<>();
Set<Integer> numeros = new HashSet<>();
Map<String, Double> notas = new HashMap<>();
```

Fácil de ler, fácil de manter. E o código continua 100% seguro e fortemente tipado.

---

## 💡 Como funciona a inferência?

O compilador do Java “olha” para o lado esquerdo da declaração e **adivinha (corretamente!)** que tipo deve estar no lado direito. Isto chama-se **inferência de tipo**.

⚠️ Só funciona quando o tipo é claro no contexto.

---

## Restrições do Diamond

O operador `<>` **não pode ser usado** se:

- O compilador **não conseguir inferir** os tipos com clareza.
- Estiveres a usar **classes anónimas**.

Exemplo **inválido**:

```
List<String> lista = new ArrayList<>() {  
    // Classe anónima → não permitido com diamond  
};
```

---

## Melhorias no Java 9+

Desde o Java 9, o operador diamond passou a funcionar também com **interfaces genéricas** (usando `var`, por exemplo):

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

Neste caso, o compilador infere **todos os tipos automaticamente**, até o da variável!

---

## Resumo Final

### O operador `<>`:

- Elimina redundância na criação de objetos genéricos.
- Torna o código mais limpo e legível.
- Funciona quando o compilador consegue **inferir os tipos**.

 Não funciona com **classes anónimas**.

 Usa-o sempre que possível para deixar o teu código mais elegante e moderno!

---

# Capítulo 8 — Do Java 9 ao Java 16: Uma Revolução em Movimento

Novas funcionalidades, mais produtividade e um toque de modernidade 

---

## Introdução

Cada nova versão do Java traz melhorias importantes — algumas subtils, outras transformadoras.

Entre o **Java 9** e o **Java 16**, a linguagem evoluiu de forma **modular, expressiva e poderosa**.

Vamos descobrir as novidades mais marcantes, versão a versão — sem complicações, com exemplos práticos e foco no que realmente interessa a quem programa no dia a dia 

---

## Java 9

### Sistema de Módulos (`module-info.java`)

Permite dividir aplicações em **módulos independentes**, melhorando segurança e manutenção.

```
module minha.app {  
    requires java.sql;  
}
```

Ideal para apps grandes. Para projetos pequenos? Podes continuar a ignorar, sem stress 😊

### jshell: o terminal interativo do Java!

Finalmente! Podemos experimentar código Java **em tempo real**, sem criar ficheiros:

```
jshell> int x = 5 + 3  
x ==> 8
```

Perfeito para testar ideias rapidamente 💡

---

## 10 Java 10

### NEW Inferência de tipo com var

O compilador descobre o tipo automaticamente:

```
var nome = "Java";
var numeros = List.of(1, 2, 3);
```

📌 Mais conciso, mas continua tipado! O tipo é inferido **em tempo de compilação**.

---

## 12 Java 11

### 🔗 Strings com novos métodos:

```
"texto ".strip();      // Remove espaços (melhor que trim)
"".isBlank();          // Verifica se está vazio ou só com espaços
"Olá\nMundo".lines(); // Divide por linhas
```

### 🌐 HTTP Client moderno

Finalmente substitui HttpURLConnection 🎉

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest req =
HttpRequest.newBuilder(URI.create("https://example.com")).build();
HttpResponse<String> res = client.send(req, BodyHandlers.ofString());
```

---

## 12 Java 12

### 💡 switch com preview para múltiplos casos:

```
switch (dia) {
    case MONDAY, FRIDAY -> System.out.println("Quase fim ou início de
semana!");
    case SUNDAY -> System.out.println("Dia de descanso");
}
```

Mais expressivo, menos verboso!

---

## Java 14

### records (modo preview)

Permite criar classes de dados com 1 linha:

```
record Pessoa(String nome, int idade) {}
```

- ◆ Cria automaticamente: construtor, getters, equals, hashCode, toString.
- 

## Java 15

### Classes Seladas (sealed) — Preview

Permite controlar **quem pode herdar** de uma classe.

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

Excelente para APIs mais seguras e lógicas mais controladas 

---

## Java 16

### records tornam-se oficiais!

Acabou o modo preview. Agora é oficial e pronto a usar.

---

## Resumo das versões 9 a 16

Versão	Novidades marcantes
Java 9	Módulos, jshell
Java 10	var para inferência de tipo
Java 11	Novo HTTP Client, novos métodos em String
Java 12	switch melhorado
Java 14	records em preview
Java 15	sealed classes em preview
Java 16	records tornados oficiais

---

👉 No próximo capítulo, vamos explorar as **melhorias das versões 17 a 21 do Java**, incluindo recursos incríveis como pattern matching para `switch`, classes seladas em definitivo, e blocos de texto multiline! Preparado para continuar a viagem? 

---

# Capítulo 9 — As Supernovas do Java 17 a 21

O Java moderno no seu melhor: poderoso, conciso e elegante 

---

## Java 17 – O Novo LTS: estabilidade com estilo

 Versão LTS (Long-Term Support), altamente recomendada para projetos reais!

 *Novidades marcantes:*

- **Pattern Matching para instanceof:**

```
if (obj instanceof String s) {  
    System.out.println(s.toUpperCase());  
}
```

 Declara e converte numa única linha. Sem mais casts!

- **Selar a Herança com sealed:**

```
sealed interface Forma permits Circulo, Quadrado {}  
final class Circulo implements Forma {}  
final class Quadrado implements Forma {}
```

 Garante que só classes autorizadas podem implementar a interface.

---

## Java 18

- **Code Points em Strings:**

```
String texto = "Olá 🙋";  
texto.codePoints().forEach(System.out::println);
```

- **HTTP/3 via incubadora** – preparação para o futuro web 
  - **UTF-8 por defeito** – Fim dos problemas de encoding 
- 

## Java 19

- **Pattern Matching para switch** (preview):

```
static String categoria(Object o) {  
    return switch (o) {  
        case Integer i -> "Número: " + i;  
        case String s -> "Texto: " + s;  
        default -> "Outro tipo";  
    };  
}
```

- **Virtual Threads (preview):**

```
Thread.startVirtualThread(() -> {
    System.out.println("Leve como uma pena 💡");
});
```

➡ Threads ultra-leves! Milhares simultaneamente. Perfeitas para servidores!

---

## ⌚ Java 20

- **Records em Pattern Matching:**

```
record Pessoa(String nome, int idade) {}
Object obj = new Pessoa("Ana", 30);

if (obj instanceof Pessoa(String nome, int idade)) {
    System.out.println(nome + " tem " + idade + " anos.");
}
```

- **Continua a evoluir o suporte a Virtual Threads e Scoped Values.**
- 

## 🌟 Java 21 – A versão que fecha o ciclo moderno

📌 Também LTS! Ideal para adoção empresarial 🚀

### ⌚ Destaques:

1. **Pattern Matching para switch oficializado!**
2. **Virtual Threads estabilizadas!**
3. **Blocos de Texto ("") estabilizados:**

```
String html = """
    <html>
        <body>Olá Mundo</body>
    </html>
""";
```

4. **Records como linguagem “de primeira classe”**
  5. **Scoped Values:** partilha de dados entre threads, com segurança.
- 

## 📋 Resumo das versões 17 a 21

Versão	Destaques
Java 17	Pattern Matching, sealed, LTS
Java 18	UTF-8 por defeito, melhorias em Strings
Java 19	switch com padrões, Virtual Threads (preview)
Java 20	Pattern Matching com record, melhorias em threads

Versão	Destaques
Java 21	Virtual Threads e Pattern Matching estabilizados, LTS

---

## O que muda na prática?

- Código mais expressivo
  - Menos boilerplate
  - Mais desempenho com threads leves
  - Mais segurança com herança controlada
  - Strings e dados estruturados mais fáceis de escrever
- 

 No próximo capítulo, vamos ver como tirar **o melhor partido da inferência de tipos com var**, e quando deves (ou não) usá-lo. Bora continuar? 

---

# 🔍 Capítulo 10 — A Magia do `var`: Inferência de Tipos no Java

Escreve menos, mantém a clareza, e continua seguro!  

---

## 💡 O que é o `var`?

Introduzido no **Java 10**, o `var` permite **inferência de tipo** ao declarar variáveis locais. Ou seja, o compilador **deduz automaticamente o tipo**, com base no valor atribuído.

```
var nome = "Java";           // É uma String
var numeros = List.of(1, 2, 3); // É uma List<Integer>
```

 O código fica mais **conciso**, mas o Java **continua fortemente tipado**. O tipo é verificado **em tempo de compilação**.

---

## ☑️ Onde se pode usar `var`?

 Permitido	 Não permitido
Variáveis <b>locais</b> com inicialização	Atributos de classe ( <code>private var</code> )
Dentro de métodos, ciclos e blocos	Sem valor inicial
Com <code>for</code> , <code>try</code> , <code>catch</code>	Em parâmetros de métodos

---

## 💡 Exemplos práticos

```
var idade = 25;           // int
var mensagem = "Olá, mundo!"; // String
var lista = new ArrayList<String>(); // ArrayList<String>
```

Com ciclos:

```
for (var nome : nomes) {
    System.out.println(nome);
}
```

Com streams:

```
var nomes = List.of("Ana", "João", "Maria");

nomes.stream()
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

---

## ⚠️ Cuidados importantes

Nem sempre var torna o código mais legível.

Evita usá-lo se o tipo não for **óbvio** ou se a inferência tornar o código **ambíguo**.

✗ Exemplo confuso:

```
var x = processarDados(); // Mas... o que devolve?
```

✓ Melhor:

```
List<String> resultado = processarDados();
```

---

## 🌐 Dicas de Boas Práticas

### 1. 📄 Prefere clareza a concisão

Usa var quando o tipo for claro e autoexplicativo.

### 2. 💡 Evita nomes genéricos com var

Nomes como data, obj, valor podem gerar confusão.

### 3. 💚 Combina com testes e IDEs

Ferramentas modernas como IntelliJ e Eclipse mostram o tipo inferido.

### 4. ⚖️ Equilibra entre legibilidade e brevidade

Em streams e coleções, o var brilha!

---

## 📋 Resumo Final

- ✖️ var reduz a verbosidade sem perder a segurança de tipos.
- ✖️ Melhora a leitura quando o tipo é evidente.
- ✖️ Requer **inicialização imediata** — o compilador precisa ver o valor para inferir o tipo.
- ✖️ Não deve ser usado em **declarações de atributos ou parâmetros de método**.