

# Introdução à Programação em Java



Luís Simões da Cunha

# Índice

 <b>Introdução</b>	8
 <b>Capítulo 1 – Introdução à Programação e à Linguagem Java</b>	10
 1.1 O que é programar?	10
 1.2 Porquê Java?	10
 1.3 Como funciona um programa Java?	10
 1.4 Ferramentas básicas	10
 1.5 O teu primeiro programa Java	11
 1.6 Aprender Java com Objetos desde o início	11
 1.7 Conceitos-chave deste capítulo	12
 1.8 Mini-laboratório	12
 1.9 Desafio	12
 <b>Capítulo 2 – Primeiros Passos com Java: Variáveis, Tipos e Operadores</b>	13
 2.1 O que são variáveis?	13
 2.2 Tipos de dados	13
 2.3 Operadores: Como fazer cálculos em Java	14
 2.4 Como declarar variáveis em Java?	15
 2.5 Como usar variáveis em cálculos?	15
 2.6 Exemplo prático	15
 2.7 Desafio	16
 2.8 Resumo dos conceitos	16
 <b>Capítulo 3 – Controle de Fluxo: Tomando Decisões</b>	17
 3.1 O que é "controle de fluxo"?	17
 3.2 O condicional if	17
 if-else: escolhe entre dois caminhos	17
 3.3 if-else-if: vários casos diferentes	18
 3.4 Repetição com ciclos	18
 3.5 O comando switch	19
 3.6 break e continue	19
 3.7 Exemplo prático: classificador de notas	20

3.8 Mini-laboratório .....	20
3.9 Desafio .....	20
3.10 Resumo dos conceitos.....	21
<b>Capítulo 4 – Introdução a Classes, Objetos e Métodos .....</b>	<b>22</b>
4.1 Chegámos à parte mais importante: a POO.....	22
4.2 O que é afinal uma classe? .....	22
4.3 O que é um objeto? .....	22
4.4 Como criar a tua primeira classe .....	23
4.5 Criar e usar um objeto .....	23
4.6 Separar bem: uma classe por ficheiro .....	23
4.7 Organização mental: classe ≠ programa completo .....	24
4.8 Reusar métodos: evitar código repetido .....	24
4.9 Resumo dos termos (com exemplos!) .....	24
4.10 Mini-laboratório.....	25
4.11 Desafio .....	25
4.12 Resumo final .....	25
<b>Capítulo 5 – Construtores: Dar Vida aos Objetos com Valores Iniciais ...</b>	<b>26</b>
5.1 O que é um construtor? .....	26
5.2 Criar um construtor personalizado .....	26
5.3 A palavra-chave this .....	27
5.4 Construtor por defeito .....	27
5.5 Sobrecarga de construtores .....	27
5.6 Atenção à ordem dos parâmetros.....	28
5.7 Vantagens de usar construtores .....	28
5.8 Mini-laboratório .....	28
5.9 Desafio .....	29
5.10 Resumo final .....	29
<b>Capítulo 6 – Encapsulamento: Esconder para Proteger .....</b>	<b>30</b>
6.1 O que é o encapsulamento?.....	30
6.2 Como se faz encapsulamento em Java?.....	30

🔒	6.3 Esconder os atributos (boa prática) .....	30
🎤	6.4 O papel dos métodos get e set .....	31
警示教育图标	6.5 Quando <i>não</i> usar set .....	31
⚠️	6.6 “Mas posso usar public nos atributos, não?” .....	32
🎯	6.7 Vantagens do encapsulamento .....	32
📝	6.8 Mini-laboratório .....	32
🎯	6.9 Desafio .....	33
🧠	6.10 Resumo final .....	33
📘	<b>Capítulo 7 – Herança: Reutilizar para Crescer</b> .....	34
🧠	7.1 O que é herança? .....	34
🧬	7.2 Como se define herança em Java?.....	34
📋	7.3 O que herda uma subclasse? .....	34
🔄	7.4 Sobrescrever métodos (@Override).....	35
💬	7.5 A palavra-chave super .....	35
🧱	7.6 Hierarquias de classes .....	36
👥	7.7 Polimorfismo com herança .....	36
📝	7.8 Mini-laboratório .....	36
🎯	7.9 Desafio .....	36
❗	7.10 Cuidados com a herança .....	37
🧠	7.11 Resumo final .....	37
📘	<b>Capítulo 8 – Polimorfismo: Um Nome, Muitos Comportamentos</b> .....	38
🧠	8.1 O que é o polimorfismo? .....	38
🧬	8.2 Polimorfismo com herança .....	38
📝	8.3 Usar o polimorfismo na prática .....	39
🔄	8.4 Vantagem: código genérico e extensível.....	39
🖨️	8.5 Polimorfismo com interfaces .....	39
🧱	8.6 Exemplo completo com lista de objetos.....	40
📝	8.7 Mini-laboratório .....	41
🎯	8.8 Desafio .....	41
❗	8.9 Polimorfismo ≠ sobrecarga .....	41

	8.10 Resumo final .....	42
	<b>Capítulo 9 – Composição: Quando um Objeto Tem Outro .....</b>	43
	9.1 O que é composição? .....	43
	9.2 Composição em Java.....	43
	9.3 Composição vs Herança.....	43
	9.4 Encapsulamento dentro da composição.....	44
	9.5 Composição com listas (coleções de objetos) .....	44
	9.6 Mini-laboratório .....	45
	9.7 Desafio .....	45
	9.8 Composição excessiva? Nem sempre. ....	45
	9.9 Resumo final.....	46
	<b>Capítulo 10 – Interfaces: Contratos de Comportamento.....</b>	47
	10.1 O que é uma interface? .....	47
	10.2 Como se define uma interface? .....	47
	10.3 Como se implementa uma interface? .....	47
	10.4 Interface ≠ Herança .....	48
	10.5 Interfaces no mundo real .....	48
	10.6 Exemplo prático com várias classes .....	48
	10.7 Desafio .....	49
	10.8 Interface com múltiplas implementações .....	49
	10.9 Interface como forma de independência .....	50
	10.10 Resumo final .....	50
	<b>Capítulo 11 – Classes Abstratas: Quando Só Faz Sentido Completar Depois .....</b>	51
	11.1 O que é uma classe abstrata? .....	51
	11.2 Quando usar uma classe abstrata? .....	51
	11.3 Como se declara uma classe abstrata? .....	51
	11.4 Uma classe abstrata não pode ser instanciada.....	52
	11.5 Implementar uma subclasse concreta .....	52
	11.6 Diferença entre classe abstrata e interface .....	52
	11.7 Exemplo completo.....	53

🎯 11.8 Desafio .....	54
🧠 11.9 Resumo final .....	54
<b>📘 Capítulo 12 – Coleções e Estruturas de Dados: Guardar e Organizar Objetos.....</b>	<b>55</b>
🧠 12.1 O que são coleções? .....	55
📦 12.2 Tipos de coleções mais usados em Java.....	55
📋 12.3 A classe ArrayList (lista dinâmica).....	55
⌚ 12.4 Percorrer listas com ciclos for - each .....	56
🔄 12.5 Remover e modificar elementos .....	56
🏢 12.6 Trabalhar com objetos em listas .....	56
🌐 12.7 O Set: coleções sem repetições.....	56
🔑 12.8 O Map: pares chave → valor .....	57
🧪 12.9 Mini-laboratório.....	57
🎯 12.10 Desafio .....	57
🧠 12.11 Resumo final .....	57
<b>📘 Capítulo 13 – Tratamento de Exceções: Lidar com Erros de Forma Segura .....</b>	<b>59</b>
🧠 13.1 O que são exceções?.....	59
💥 13.2 Erros em tempo de execução .....	59
🌐 13.3 Usar try - catch para proteger o programa .....	59
🎈 13.4 Bloco finally: sempre executado.....	60
✍ 13.5 Exceções com entrada de dados .....	60
⚠ 13.6 Exceções mais comuns .....	60
👤 13.7 Criar exceções personalizadas .....	61
🧪 13.8 Mini-laboratório.....	61
🎯 13.9 Desafio .....	61
🧠 13.10 Resumo final .....	62
<b>📘 Capítulo 14 – Interface Gráfica com Swing (GUI).....</b>	<b>63</b>
🧠 14.1 O que é o Swing? .....	63
🚀 14.2 O essencial para começar.....	63
✳ 14.3 Adicionar botões e interações .....	64

✓ Componentes comuns:.....	64
14.4 Layouts: organizar os componentes.....	64
14.5 Formulário simples com campos e botão.....	65
14.6 Como desenhar um gráfico simples.....	65
14.7 Desafio .....	66
14.8 Resumo final .....	67
<b>Capítulo 15 – Boas Práticas e Estruturação de Projetos .....</b>	68
15.1 Porque é que estrutura e boas práticas são tão importantes?.....	68
15.2 Separar bem as classes .....	68
15.3 Encapsular tudo o que puder.....	68
15.4 Usar construtores com lógica.....	69
15.5 Relacionar classes com sentido .....	69
15.6 Usar listas e mapas com significado .....	69
15.7 Comentar com utilidade .....	69
15.8 Escrever código limpo.....	70
15.9 Mini-laboratório.....	70
15.10 Desafio .....	71
15.11 Resumo final .....	71

## Introdução

Este documento foi concebido como um **guião prático e acessível** para apoiar os estudantes que estão a **dar os primeiros passos na Programação Orientada por Objetos (POO)**.

Trata-se de um recurso complementar, criado pelo docente da unidade curricular com o mesmo nome, com o objetivo de:

-  **Acompanhar os conteúdos lecionados nas aulas**
  -  **Oferecer exemplos claros e funcionais em Java**
  -  **Ajudar a compreender os conceitos mais abstratos da POO**
  -  **Fornecer um ponto de partida para o desenvolvimento de projetos**
- 

### Para quem é este manual?

Este manual destina-se a **estudantes do 1.º ano do curso de Informática** (ou áreas afins), que estejam a iniciar o contacto com os princípios fundamentais da programação por objetos:

- Classes e Objetos
  - Herança e Polimorfismo
  - Encapsulamento e Abstração
  - Interfaces e Classes Abstratas
  - Estruturas de dados e manipulação de coleções
  - Interface Gráfica com Swing
  - Tratamento de erros e boas práticas
- 

### Como usar este documento?

O manual está dividido por **capítulos curtos, pedagógicos e progressivos**, com:

- Explicações passo a passo
  - Exemplos comentados
  - Mini-laboratórios e desafios
  - Códigos testáveis diretamente no Visual Studio Code
  - Um projeto completo que pode servir de referência para avaliação
- 

#### Um entre vários instrumentos

Este manual é apenas **um dos vários instrumentos desenvolvidos pelo docente** da UC de POO, integrando uma estratégia pedagógica mais ampla que inclui:

- Aulas práticas com acompanhamento individual
  - Recursos interativos no Moodle
  - Banco de questões para autoavaliação
  - Projetos orientados e guiões de defesa
- 

#### Desejo de sucesso

Espero que este recurso te ajude a:

- Aprender com prazer
- Explorar com curiosidade
- Programar com confiança

Porque programar bem **não é decorar comandos**, é compreender ideias. E este manual está aqui para te ajudar a fazê-lo.

---

#### O docente

Luís Simões da Cunha



(2025)

## Capítulo 1 – Introdução à Programação e à Linguagem Java

---

### 1.1 O que é programar?

Programar é **dar instruções a um computador** para que ele resolva problemas por nós. Mas atenção: o computador **só faz o que lhe mandamos fazer**, e não o que "achamos" que ele vai entender.

 **Analogia simples:** imagina que estás a dar instruções a um robô que não percebe nada de bom senso. Se disseres "faz o jantar", ele vai ficar parado. Mas se deres ordens passo a passo, tipo "pega num tacho", "enche com água", "acende o fogão", ele vai conseguir!

---

### 1.2 Porquê Java?

Java é uma das linguagens de programação mais usadas no mundo, por várias razões:

- Funciona em quase todos os computadores e sistemas (Windows, macOS, Linux...)
  - É usada em aplicações móveis (como Android), web e até em bancos!
  - É baseada em **POO – Programação Orientada por Objetos**, o que ajuda a organizar bem os projetos
  - Tem uma comunidade enorme e muito material de apoio
- 

### 1.3 Como funciona um programa Java?

Ao contrário de linguagens como Python, que são interpretadas, o Java precisa de **duas etapas** para funcionar:

1. **Compilar** o código (converter para *bytecode*)
2. **Executar** o programa na máquina virtual Java (JVM)

### Compilação + Execução = Portabilidade

Escreves uma vez, corre em todo o lado!

---

### 1.4 Ferramentas básicas

Para começares a programar em Java, precisas de:

- **JDK** – Java Development Kit (é o motor)
  - **Editor de texto** (como o VS Code ou IntelliJ)
  - **Terminal/linha de comandos**
- Usa o VS Code com a extensão “Extension Pack for Java” — é leve, gratuito e excelente para quem começa.
- 

## 1.5 O teu primeiro programa Java

Vamos ver o programa mais famoso do mundo: o "**Olá, Mundo!**" 😊

```
public class OlaMundo {  
    public static void main(String[] args) {  
        System.out.println("Olá, Mundo!");  
    }  
}
```

Explicação passo a passo:

- `public class OlaMundo`: define uma **classe** chamada `OlaMundo`
- `public static void main(...)`: é o **ponto de entrada** do programa
- `System.out.println(...)`: escreve uma mensagem no **ecrã**

Para compilar e correr:

```
javac OlaMundo.java  # compilar  
java OlaMundo        # executar
```

---

## 1.6 Aprender Java com Objetos desde o início

Java foi desenhado com **POO no coração**. Isso quer dizer que tudo gira à volta de **classes e objetos**, como vais ver nos próximos capítulos.

**Resumo em modo "fast forward":**

- Crias **classes** → como receitas
  - A partir delas, crias **objetos** → como bolos reais
  - Cada objeto tem **atributos** (estado) e **métodos** (ações)
-

## 1.7 Conceitos-chave deste capítulo

Conceito	Explicação simples
Programa	Conjunto de instruções para o computador
Java	Linguagem multiplataforma, orientada a objetos
Compilar	Converter código para linguagem que a máquina entende
Classe	Modelo ou receita para criar objetos
Objeto	Entidade criada a partir de uma classe
Método main	Onde o programa Java começa a correr

---

## 1.8 Mini-laboratório

Cria um programa chamado `Apresentacao.java` que escreva o teu nome, idade e curso no ecrã:

```
public class Apresentacao {  
    public static void main(String[] args) {  
        System.out.println("O meu nome é João.");  
        System.out.println("Tenho 18 anos.");  
        System.out.println("Estou no curso de Informática.");  
    }  
}
```

-  Experimenta mudar as mensagens.
  -  Acrescenta mais linhas com curiosidades sobre ti.
- 

## 1.9 Desafio

Cria um programa chamado `CalculadoraSimples` que mostre o resultado de:

- uma soma
- uma subtração
- uma multiplicação
- uma divisão

Usa só instruções `System.out.println()` com contas simples dentro.

---

## Capítulo 2 – Primeiros Passos com Java: Variáveis, Tipos e Operadores

---

### 2.1 O que são variáveis?

Uma **variável** é como uma "caixinha" onde guardamos dados. Essa caixinha tem um **nome** e um **tipo** de dado que pode armazenar, como números ou texto.

 **Analogia simples:** Imagina que tens várias gavetas no teu escritório, e cada gaveta tem um nome. Dentro de cada gaveta, guardas algo específico: uma gaveta para canetas, outra para documentos importantes, e assim por diante.

Em Java, as variáveis funcionam de forma parecida. Por exemplo, uma variável **idade** pode guardar o número de anos de uma pessoa, enquanto uma variável **nome** pode guardar o nome de alguém.

---

### 2.2 Tipos de dados

Em Java, existem dois tipos principais de dados: **tipos primitivos** e **tipos referenciados**.

*Tipos primitivos:*

- **int:** Números inteiros (ex: 10, -7, 0)
- **double:** Números decimais (ex: 3.14, -0.5)
- **char:** Um único caractere (ex: 'a', '1', '#')
- **boolean:** Valores verdadeiro ou falso (ex: true, false)

 **Exemplo:**

```
int idade = 20; // Armazena um número inteiro
double peso = 70.5; // Armazena um número decimal
char letra = 'A'; // Armazena um único caractere
boolean estaChovendo = false; // Armazena verdadeiro ou falso
```

*Tipos referenciados:*

- **String:** Sequência de caracteres (ex: "Olá Mundo")
- **Arrays:** Lista de dados do mesmo tipo (ex: uma lista de números)

 **Exemplo:**

```
String nome = "João"; // Armazena uma sequência de caracteres
int[] idades = {20, 21, 22}; // Armazena uma lista de idades
```

---

## ➊ 2.3 Operadores: Como fazer cálculos em Java

Os **operadores** são usados para realizar operações com variáveis ou valores. Em Java, temos vários tipos de operadores.

### 2.3.1 Operadores aritméticos (para cálculos)

- `+`: soma (ex: `5 + 3`)
- `-`: subtração (ex: `5 - 3`)
- `*`: multiplicação (ex: `5 * 3`)
- `/`: divisão (ex: `6 / 2`)
- `%`: resto da divisão (ex: `7 % 3` dá 1)

#### 💡 Exemplo:

```
int a = 5;
int b = 3;
int soma = a + b; // soma = 8
int resto = a % b; // resto = 2
```

### 2.3.2 Operadores de comparação (para comparar valores)

- `==`: igual (ex: `a == b`)
- `!=`: diferente (ex: `a != b`)
- `>`: maior que (ex: `a > b`)
- `<`: menor que (ex: `a < b`)
- `>=`: maior ou igual (ex: `a >= b`)
- `<=`: menor ou igual (ex: `a <= b`)

#### 💡 Exemplo:

```
int a = 5;
int b = 3;
boolean igual = (a == b); // igual = false
boolean maior = (a > b); // maior = true
```

### 2.3.3 Operadores lógicos (para tomar decisões)

- `&&`: e (ex: `a > b && b > 0`)
- `||`: ou (ex: `a > b || b > 0`)

- `!:` não (ex: `!(a > b)`)

 **Exemplo:**

```
boolean resultado = (a > b) && (b > 0); // resultado = true
```

---

 **2.4 Como declarar variáveis em Java?**

Para declarar uma variável, precisas de **especificar o tipo e dar-lhe um nome**. Se quiseres atribuir um valor logo de seguida, basta usar o sinal `=`.

 **Exemplo:**

```
int idade = 18; // Declara e atribui um valor  
String nome = "Maria"; // Declara e atribui um valor
```

Além disso, se quiseres **não inicializar uma variável**, basta declarar apenas o tipo e nome:

```
int altura; // Declara, mas ainda não atribui valor
```

 **Boa prática:** É sempre melhor **inicializar a variável** logo quando a declares, para evitar erros.

---

 **2.5 Como usar variáveis em cálculos?**

Agora que já sabes como declarar e atribuir valores a variáveis, vamos ver como usá-las em cálculos.

 **Exemplo:**

```
int a = 10;  
int b = 5;  
int resultado = a + b; // soma: resultado = 15  
System.out.println(resultado);
```

---

 **2.6 Exemplo prático**

Vamos criar um programa simples para calcular a **média** de três números que o utilizador introduz:

```

import java.util.Scanner;

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

        System.out.print("Digite o primeiro número: ");
        double num1 = scanner.nextDouble();
        System.out.print("Digite o segundo número: ");
        double num2 = scanner.nextDouble();
        System.out.print("Digite o terceiro número: ");
        double num3 = scanner.nextDouble();

        double media = (num1 + num2 + num3) / 3;
        System.out.println("A média é: " + media);
    }
}

```

→ Este programa usa a classe `Scanner` para ler números do utilizador e depois calcula a média deles.

---

## 2.7 Desafio

Cria um programa que calcule o **imposto sobre um produto**. O programa deve perguntar pelo preço e pela taxa de imposto, e mostrar o valor final com imposto.

---

## 2.8 Resumo dos conceitos

Conceito	Explicação simples
Variáveis	Caixinhas onde guardamos dados (números, textos, etc.)
Tipos de dados	Especifica que tipo de dado uma variável pode armazenar (int, double, String, etc.)
Operadores aritméticos	Ferramentas para fazer cálculos (+, -, *, /, %)
Operadores de comparação	Ferramentas para comparar valores (==, !=, >, <)
Operadores lógicos	Ferramentas para tomar decisões (&&,

---

## Capítulo 3 – Controle de Fluxo: Tomando Decisões

---

### 3.1 O que é "controle de fluxo"?

Quando programas, nem sempre queres que tudo aconteça numa ordem fixa. Às vezes queres que o programa **tome decisões**, ou **repita ações** várias vezes.

 O **controle de fluxo** permite que o teu código:

- Tome decisões com **condições** (`if`, `else`)
- Faça escolhas entre vários caminhos (`switch`)
- Repita blocos de código (`while`, `for`, `do-while`)

 **Analogia simples:** imagina que estás a seguir uma receita. Se tens ovos, fazes um bolo. Se não tens, fazes panquecas. Essa escolha — com base numa condição — é controle de fluxo.

---

### 3.2 O condicional `if`

O `if` é a forma mais simples de tomar decisões em Java.

*Exemplo:*

```
int idade = 18;

if (idade >= 18) {
    System.out.println("És maior de idade.");
}
```

 **if-else:** escolhe entre dois caminhos

```
if (idade >= 18) {
    System.out.println("És maior de idade.");
} else {
    System.out.println("És menor de idade.");
}
```

---

### 3.3 if-else-if: vários casos diferentes

```
int nota = 15;

if (nota >= 18) {
    System.out.println("Excelente!");
} else if (nota >= 10) {
    System.out.println("Aprovado");
} else {
    System.out.println("Reprovado");
}
```

 O Java avalia as condições **de cima para baixo**. Assim que encontrar uma condição verdadeira, **ignora as restantes**.

---

### 3.4 Repetição com ciclos

Quando precisas de repetir um bloco de código várias vezes, usas **ciclos**.

*while: repete enquanto a condição for verdadeira*

```
int i = 1;
while (i <= 5) {
    System.out.println("Contar: " + i);
    i++;
}
```

---

*do-while: repete pelo menos uma vez*

```
int i = 1;
do {
    System.out.println("Contar: " + i);
    i++;
} while (i <= 5);
```

 A principal diferença é que o do-while **executa o bloco pelo menos uma vez**, mesmo que a condição seja falsa logo no início.

---

*for: ideal quando sabes quantas vezes vais repetir*

```
for (int i = 1; i <= 5; i++) {  
    System.out.println("Contar: " + i);  
}
```

💡 O **for** é dividido em três partes:

- `int i = 1`: inicia a variável
  - `i <= 5`: condição de continuação
  - `i++`: incrementa no final de cada repetição
- 

## 🔗 3.5 O comando **switch**

O **switch** permite **testar várias opções de um valor** — como um menu.

```
int dia = 3;  
  
switch (dia) {  
    case 1:  
        System.out.println("Segunda-feira");  
        break;  
    case 2:  
        System.out.println("Terça-feira");  
        break;  
    case 3:  
        System.out.println("Quarta-feira");  
        break;  
    default:  
        System.out.println("Dia inválido");  
}
```

⚠ **Importante:** sem o `break`, o Java continua a executar os casos seguintes (efeito chamado *fall-through*).

---

## ─ 3.6 **break** e **continue**

- **break**: sai imediatamente do ciclo ou **switch**
- **continue**: salta o resto do ciclo atual e passa para a próxima repetição

*Exemplo com continue:*

```
for (int i = 1; i <= 5; i++) {  
    if (i == 3) continue;
```

```
        System.out.println(i);
    }
// Resultado: 1, 2, 4, 5 (3 foi ignorado)
```

---

### 3.7 Exemplo prático: classificador de notas

```
import java.util.Scanner;

public class Classificador {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Insere a nota (0-20): ");
        int nota = scanner.nextInt();

        if (nota >= 18) {
            System.out.println("Excelente!");
        } else if (nota >= 10) {
            System.out.println("Aprovado.");
        } else {
            System.out.println("Reprovado.");
        }
    }
}
```

---

### 3.8 Mini-laboratório

Cria um programa Contador que:

- Mostre os números de 1 a 10
  - Mas salte os múltiplos de 3 (usa `continue!`)
- 

### 3.9 Desafio

Cria um programa que:

- Pergunta ao utilizador que operação quer fazer (+, -, \*, /)
  - Lê dois números
  - Usa `switch` para aplicar a operação correta
-

## 3.10 Resumo dos conceitos

Conceito	Explicação
<code>if / else</code>	Escolher entre dois ou mais caminhos
<code>while, for, do-while</code>	Repetição de blocos de código
<code>switch</code>	Alternativa ao <code>if-else-if</code> para muitos casos
<code>break</code>	Sai de um ciclo ou bloco
<code>continue</code>	Salta uma iteração e continua o ciclo

---

## Capítulo 4 – Introdução a Classes, Objetos e Métodos

---

### 4.1 Chegámos à parte mais importante: a POO

 **Tudo o que aprendemos até agora (variáveis, operadores, ciclos)** foram ferramentas fundamentais...

Mas atenção: **ainda não programámos “à moda do Java”!** 

 A partir deste capítulo, **entras na Programação Orientada por Objetos (POO)** — a forma como os projetos Java **devem ser construídos**.

---

### 4.2 O que é afinal uma classe?

Uma **classe** é uma **receita**. Define como algo deve ser:

- Que **atributos** (dados) tem
- Que **métodos** (ações) pode executar

 **Analogia simples:**

Pensa numa forma de bolachas. A forma define o aspeto das bolachas, mas **não é uma bolacha**.

As bolachas reais são **os objetos** que saem dessa forma.

---

### 4.3 O que é um objeto?

Um **objeto** é uma **instância concreta** de uma classe.

Exemplo do mundo real:

- **Classe:** Carro
- **Objeto:** O meu carro vermelho da marca X com matrícula Y

Em código:

```
Carro meuCarro = new Carro();
```

---

## 4.4 Como criar a tua primeira classe

```
public class Pessoa {  
    String nome;  
    int idade;  
  
    void apresentar() {  
        System.out.println("Olá, o meu nome é " + nome + " e tenho " +  
idade + " anos.");  
    }  
}
```

- `nome` e `idade` → são **atributos** (dados)
  - `apresentar()` → é um **método** (ação)
- 

## 4.5 Criar e usar um objeto

```
public class TestePessoa {  
    public static void main(String[] args) {  
        Pessoa p1 = new Pessoa(); // Criar objeto  
        p1.nome = "Maria";  
        p1.idade = 20;  
        p1.apresentar(); // Chamar método  
    }  
}
```

 O que está a acontecer?

- `new Pessoa()` → cria uma nova pessoa na memória
  - `p1.nome = "Maria"` → atribui o nome
  - `p1.apresentar()` → executa a ação de apresentar
- 

## 4.6 Separar bem: uma classe por ficheiro

Regra de ouro:

 Cada classe pública deve estar no **seu próprio ficheiro** com o mesmo nome.

Exemplo:

- Classe `Pessoa` → ficheiro `Pessoa.java`
- Classe `TestePessoa` → ficheiro `TestePessoa.java`

---

## 4.7 Organização mental: classe ≠ programa completo

Uma **classe sozinha não corre nada**.

Precisas de uma classe com o método `main()` para iniciar a execução.

Por isso, tens:

- **Modelos (classes)** → Pessoa, Carro, Livro
  - **Executores (testes)** → Main, TesteCarro, AppBiblioteca
- 

## 4.8 Reusar métodos: evitar código repetido

Podes criar métodos que **manipulam os dados** da própria classe.

Isto melhora a **organização, reutilização** e clareza.

```
public class Calculadora {  
    int somar(int a, int b) {  
        return a + b;  
    }  
}
```

Usar no `main()`:

```
Calculadora calc = new Calculadora();  
int resultado = calc.somar(3, 5);  
System.out.println("Resultado: " + resultado);
```

---

## 4.9 Resumo dos termos (com exemplos!)

Conceito	Definição	Exemplo
Classe	Receita ou modelo	<code>class Pessoa { ... }</code>
Objeto	Exemplar da classe	<code>Pessoa p1 = new Pessoa();</code>
Atributo	Dado que pertence à classe	<code>String nome;</code>
Método	Ação que a classe executa	<code>void apresentar() { ... }</code>
Instanciar	Criar um novo objeto	<code>new Pessoa()</code>

---

## 4.10 Mini-laboratório

Cria uma classe `Livro` com:

- Atributos: `título`, `autor`, `ano`
- Um método `mostrarInfo()` que imprime os dados

Depois, no `main()`, cria dois livros e mostra a informação de cada um.

---

## 4.11 Desafio

Cria uma classe `Cão` com os seguintes elementos:

- Atributos: `nome`, `raça`
- Métodos: `latir()` (imprime "Au au!") e `apresentar()`

No `main()`, cria dois cães e chama os métodos.

---

## 4.12 Resumo final

 Este foi o **verdadeiro ponto de partida da POO**. Até aqui, estivemos só a aprender a linguagem do Java. Agora estamos a começar a **pensar como programadores orientados a objetos**.

 Lembrar:

- Uma **classe** define o que os objetos **são e fazem**
- Um **objeto** é uma instância concreta
- Podemos criar métodos para **agir sobre os atributos**

---

## Capítulo 5 – Construtores: Dar Vida aos Objetos com Valores Iniciais

---

### 5.1 O que é um construtor?

Um **construtor** é um método especial que é **chamado automaticamente** quando criamos um objeto com `new`.

O seu papel é simples mas essencial: **dar valores iniciais aos atributos** do objeto.

#### Analogia simples:

Quando montas um móvel, segues um manual (a classe), e o resultado (o objeto) já vem com os parafusos apertados e as peças no sítio certo. O **construtor** é o momento de montagem.

---

### 5.2 Criar um construtor personalizado

```
public class Pessoa {  
    String nome;  
    int idade;  
  
    // Construtor  
    public Pessoa(String nomeInicial, int idadeInicial) {  
        nome = nomeInicial;  
        idade = idadeInicial;  
    }  
  
    void apresentar() {  
        System.out.println("Olá, sou " + nome + " e tenho " + idade + " anos.");  
    }  
}
```

Agora, no `main()`:

```
Pessoa p = new Pessoa("João", 25);  
p.apresentar();
```

 O construtor recebe dois parâmetros e usa-os para preencher os atributos do objeto.

---

## 5.3 A palavra-chave `this`

Quando o nome do parâmetro é igual ao do atributo, usamos `this` para **diferenciar**.

```
public Pessoa(String nome, int idade) {  
    this.nome = nome;  
    this.idade = idade;  
}
```

- ☞ `this.nome` refere-se ao **atributo do objeto**
  - ☞ `nome` (sem `this`) refere-se ao **parâmetro recebido**
- 

## 5.4 Construtor por defeito

Se **não escreveres nenhum construtor**, o Java cria um **construtor por defeito** (sem parâmetros).

```
Pessoa p = new Pessoa(); // Só funciona se não tiveres definido nenhum  
construtor
```

! Mas se definires um construtor com parâmetros, o construtor por defeito **deixa de existir automaticamente**. Se quiseres tê-lo, tens de o criar:

```
public Pessoa() {  
    nome = "Desconhecido";  
    idade = 0;  
}
```

---

## 5.5 Sobrecarga de construtores

Podes ter **vários construtores na mesma classe**, com assinaturas diferentes. Isto dá flexibilidade!

```
public class Pessoa {  
    String nome;  
    int idade;  
  
    public Pessoa(String nome) {  
        this.nome = nome;  
        this.idade = 0; // valor por defeito  
    }
```

```
public Pessoa(String nome, int idade) {  
    this.nome = nome;  
    this.idade = idade;  
}  
}
```

➡ Agora podes criar:

```
Pessoa p1 = new Pessoa("Ana");  
Pessoa p2 = new Pessoa("Carlos", 30);
```

---

## ⚠ 5.6 Atenção à ordem dos parâmetros

Os construtores são diferenciados pela **quantidade e ordem dos tipos** dos parâmetros.  
Por isso:

```
Pessoa(String nome, int idade) // é diferente de  
Pessoa(int idade, String nome)
```

---

## 📋 5.7 Vantagens de usar construtores

- ✓ Torna a criação de objetos **mais clara e concisa**
  - ✓ Evita esquecer atributos importantes
  - ✓ Garante que os objetos começam **num estado válido**
- 

## ✍ 5.8 Mini-laboratório

Cria uma classe Aluno com:

- Atributos: nome, curso, ano
  - Dois construtores:
    - Um que recebe todos os dados
    - Outro que só recebe o nome e preenche os restantes com valores por defeito
-

## ⌚ 5.9 Desafio

Cria uma classe `Produto` com os atributos `nome`, `preco`, `stock`.

- Cria três construtores:
  - Um com todos os parâmetros
  - Um só com nome e preço
  - Um por defeito, com valores base

No `main()`, cria três produtos diferentes e imprime a informação de cada um.

---

## ⌚ 5.10 Resumo final

Conceito	Explicação
Construtor	Método especial que inicializa o objeto
<code>this</code>	Refere-se ao próprio objeto (útil para diferenciar nomes)
Construtor por defeito	Criado automaticamente se nenhum for declarado
Sobrecarga	Ter vários construtores com parâmetros diferentes

---

## Capítulo 6 – Encapsulamento: Esconder para Proteger

---

### 6.1 O que é o encapsulamento?

**Encapsulamento** é um dos pilares da Programação Orientada por Objetos. Significa **esconder os detalhes internos de uma classe**, deixando que outros accedam apenas ao que é necessário.

#### Analogia simples:

Um micro-ondas tem botões para aquecer, parar e abrir a porta.

Tu **usas os botões**, mas **não precisas de ver os fios lá dentro**.

Isso é encapsulamento: usas sem conhecer os detalhes técnicos.

---

### 6.2 Como se faz encapsulamento em Java?

Usamos **modificadores de acesso** para controlar quem pode ver ou alterar o quê:

Modificador	Visível para...
<code>private</code>	Só dentro da própria classe
<code>public</code>	Visível para todas as classes
<code>protected</code>	Visível na própria classe e nas suas subclasses
<i>(sem nada)</i>	Visível no mesmo pacote

---

### 6.3 Esconder os atributos (boa prática)

A boa prática em POO é:

 **Tornar os atributos `private`** e aceder a eles através de métodos públicos.

```
public class Conta {  
    private double saldo;  
  
    public double getSaldo() {  
        return saldo;  
    }  
  
    public void depositar(double valor) {
```

```

        if (valor > 0) {
            saldo += valor;
        }
    }
}

```

📌 Quem usa a classe não pode alterar o `saldo` diretamente, apenas através dos métodos permitidos.

---

## 🎙 6.4 O papel dos métodos `get` e `set`

- `getX()` → devolve o valor de um atributo
- `setX(valor)` → altera o valor de um atributo (se for permitido)

```

public class Pessoa {
    private String nome;

    public String getName() {
        return nome;
    }

    public void setName(String novoNome) {
        if (!novoNome.isEmpty()) {
            nome = novoNome;
        }
    }
}

```

⌚ O método `setName()` garante que o nome **não fica vazio**, ou seja, mantém o objeto num estado válido.

---

## ⌚ 6.5 Quando *não* usar `set`

Nem todos os atributos devem ser modificáveis!

Podes ter atributos **só de leitura** (`get`, mas sem `set`) ou **só de escrita**.

💡 Exemplo:

```

public class Sensor {
    private final String id;

    public Sensor(String id) {

```

```
        this.id = id;
    }

    public String getId() {
        return id;
    }
}
```

📌 O `id` nunca muda, por isso **não existe `setId()`**.

---

## ⚠ 6.6 “Mas posso usar `public` nos atributos, não?”

Poder, podes... mas **não deves**.

Quando tornas um atributo `public`, qualquer parte do programa pode fazer isto:

```
conta.saldo = -1000;
```

📌 Resultado: o objeto entra num **estado inválido** e perigoso. O encapsulamento existe **precisamente para evitar isso**.

---

## ⌚ 6.7 Vantagens do encapsulamento

- ✓ Protege os dados
  - ✓ Permite aplicar regras (ex: só aceitar valores válidos)
  - ✓ Facilita a manutenção e evolução da classe
  - ✓ Esconde detalhes que não interessam a quem usa o objeto
- 

## ✍ 6.8 Mini-laboratório

Cria uma classe `Temperatura` com:

- Um atributo privado `celsius`
  - Métodos:
    - `getCelsius()`
    - `setCelsius(double valor)` (só aceita valores entre -100 e 100)
    - `getFahrenheit()` (devolve a conversão para Fahrenheit)
-

## 6.9 Desafio

Cria uma classe `Produto` com:

- Atributos privados: `nome`, `preco`, `quantidadeEmStock`
  - Getters e setters com validação:
    - `preco` não pode ser negativo
    - `quantidadeEmStock` não pode ser inferior a 0
- 

## 6.10 Resumo final

Conceito	Explicação
Encapsulamento	Esconder o interior de uma classe, expondo só o necessário
<code>private</code>	Protege os atributos da classe
Getters e Setters	Métodos públicos para aceder ou alterar atributos
Validação	Garante que os dados mantêm-se válidos

---

## Capítulo 7 – Herança: Reutilizar para Crescer

---

### 7.1 O que é herança?

Herança é um mecanismo que permite que uma **classe herde atributos e métodos de outra classe**.

A ideia é simples: se várias classes partilham comportamento comum, podemos **definir esse comportamento numa superclasse** e deixá-las **herdar**.

 **Analogia simples:**

Imagina que há uma classe **Veiculo**.

Carro, Motociclo e Camião são todos veículos. Em vez de repetir o código em cada um, coloca o comum em **Veiculo** e **reutilizas**.

---

### 7.2 Como se define herança em Java?

```
public class Veiculo {  
    int velocidade;  
  
    void acelerar() {  
        velocidade += 10;  
        System.out.println("A acelerar: " + velocidade + " km/h");  
    }  
}  
  
public class Carro extends Veiculo {  
    void buzinar() {  
        System.out.println("Buzina! 🚗");  
    }  
}
```

 A palavra-chave **extends** indica que **Carro herda de Veiculo**.

---

### 7.3 O que herda uma subclasse?

Uma subclasse **herda tudo o que for public ou protected** na superclasse:

- Atributos
- Métodos

Mas:

- **Não herda os construtores**
  - **Não acede a membros private** diretamente
- 

#### 7.4 Sobrescrever métodos (@Override)

Uma subclasse pode **alterar o comportamento** de um método herdado, escrevendo a sua própria versão.

```
public class Cão {  
    void fazerSom() {  
        System.out.println("Som genérico");  
    }  
}  
  
public class PastorAlemao extends Cão {  
    @Override  
    void fazerSom() {  
        System.out.println("Au au!");  
    }  
}
```

---

#### 7.5 A palavra-chave super

A palavra **super** serve para:

1. Chamar **métodos da superclasse** que foram sobrescritos
2. Invocar **construtores da superclasse**

```
public class Animal {  
    Animal(String nome) {  
        System.out.println("Animal: " + nome);  
    }  
}  
  
public class Gato extends Animal {  
    Gato() {  
        super("Felix"); // chama o construtor da superclasse
```

```
    }  
}
```

---

## 7.6 Hierarquias de classes

Podes construir cadeias de herança:

```
class Animal { ... }  
class Mamifero extends Animal { ... }  
class Humano extends Mamifero { ... }
```

- Cada classe herda da anterior, acumulando comportamentos e podendo acrescentar ou alterar funcionalidades.
- 

## 7.7 Polimorfismo com herança

Quando usas herança, podes **tratar objetos da subclasse como se fossem da superclasse**.

```
Animal a = new Gato();  
a.fazerSom(); // chama a versão de Gato, se estiver sobreposta
```

- Isto é **polimorfismo**, e vamos explorá-lo mais no próximo capítulo.
- 

## 7.8 Mini-laboratório

Cria:

- Uma superclasse Funcionario com nome e método mostrarDados()
  - Uma subclasse Professor que acrescenta disciplina e **sobre**escreve mostrarDados()
  - Um main() que cria um professor e mostra os dados
- 

## 7.9 Desafio

Cria uma hierarquia:

- Veiculo com marca, modelo, acelerar()
- Carro que herda de Veiculo e tem buzinar()

- Camião que herda de Veiculo e tem carregarCarga()

Testa os métodos no `main()` com dois veículos diferentes.

---

## ! 7.10 Cuidados com a herança

Embora seja poderosa, a herança deve ser usada **quando há uma verdadeira relação "é-um"**:

- ✓ Um **Gato** é um **Animal** → ok
- ✗ Um **Carro** é um **Motor** → não (aqui é composição: um carro **tem** um motor)

 **Dica de ouro:** se a frase "X é um Y" fizer sentido, talvez devas usar herança.

---

## ⌚ 7.11 Resumo final

Conceito	Explicação
Herança	Permite que uma classe herde de outra
<code>extends</code>	Palavra-chave para herdar
<code>super</code>	Acede a membros ou construtores da superclasse
<code>@Override</code>	Indica que um método foi redefinido
Hierarquia	Cadeia de herança com várias camadas

---

## Capítulo 8 – Polimorfismo: Um Nome, Muitos Comportamentos

---

### 8.1 O que é o polimorfismo?

**Polimorfismo** vem do grego *poly* (muitos) + *morphos* (formas).

Em programação orientada por objetos, significa que **um mesmo método pode ter comportamentos diferentes dependendo do objeto que o executa.**

#### Analogia simples:

Pede a várias pessoas para "tocar música".

Um pianista toca piano, um guitarrista toca guitarra — **todos respondem ao mesmo comando**, mas de formas diferentes.

---

### 8.2 Polimorfismo com herança

Quando tens uma **superclasse** e várias **subclasses** que **sobrescrevem o mesmo método**, podes usar uma variável da superclasse para chamar o método — e o Java escolhe **automaticamente** a versão correta.

*Exemplo:*

```
class Animal {  
    void fazerSom() {  
        System.out.println("Som genérico");  
    }  
}  
  
class Cão extends Animal {  
    @Override  
    void fazerSom() {  
        System.out.println("Au au");  
    }  
}  
  
class Gato extends Animal {  
    @Override  
    void fazerSom() {  
        System.out.println("Miau");  
    }  
}
```

```
    }  
}
```

---

### 8.3 Usar o polimorfismo na prática

```
public class Teste {  
    public static void main(String[] args) {  
        Animal a1 = new Cão();  
        Animal a2 = new Gato();  
  
        a1.fazerSom(); // Au au  
        a2.fazerSom(); // Miau  
    }  
}
```

→ Mesmo usando a referência **Animal**, o Java **escolhe o método certo em tempo de execução**.

---

### 8.4 Vantagem: código genérico e extensível

Com polimorfismo, podemos escrever código que **funciona com qualquer tipo de objeto**, desde que respeite a estrutura comum (ou seja, **herde ou implemente a mesma interface**).

```
public class Zoo {  
    public static void emitirSom(Animal a) {  
        a.fazerSom();  
    }  
  
    public static void main(String[] args) {  
        emitirSom(new Cão());  
        emitirSom(new Gato());  
    }  
}
```

---

### 8.5 Polimorfismo com interfaces

Polimorfismo não depende só da herança!

Também acontece com **interfaces** — veremos isso mais a fundo no Capítulo 10.

```

interface Imprimivel {
    void imprimir();
}

class Documento implements Imprimivel {
    public void imprimir() {
        System.out.println("Imprimir documento...");
    }
}

class Imagem implements Imprimivel {
    public void imprimir() {
        System.out.println("Imprimir imagem...");
    }
}

public class Teste {
    public static void main(String[] args) {
        Imprimivel i1 = new Documento();
        Imprimivel i2 = new Imagem();

        i1.imprimir(); // Imprimir documento...
        i2.imprimir(); // Imprimir imagem...
    }
}

```

---

## 8.6 Exemplo completo com lista de objetos

```

public class Main {
    public static void main(String[] args) {
        Animal[] animais = new Animal[3];
        animais[0] = new Cão();
        animais[1] = new Gato();
        animais[2] = new Cão();

        for (Animal a : animais) {
            a.fazerSom();
        }
    }
}

```

- ➡ Mesmo com diferentes tipos de animais, todos podem ser tratados de forma **uniforme**, graças ao polimorfismo.
- 

## 8.7 Mini-laboratório

Cria:

- Superclasse `Funcionario` com método `calcularSalario()`
- Subclasse `Professor` com salário fixo
- Subclasse `Tecnico` com salário por horas

Cria um array de `Funcionario` e calcula os salários de vários trabalhadores.

---

## 8.8 Desafio

Define uma interface `Desenhavel` com método `desenhar()`.

Depois cria duas classes:

- `Circulo` que imprime "Desenhar círculo"
- `Quadrado` que imprime "Desenhar quadrado"

No `main()`, cria uma lista de objetos `Desenhavel` e desenha todos com um ciclo.

---

## 8.9 Polimorfismo ≠ sobrecarga

⚠ Atenção para não confundir **polimorfismo** com **sobrecarga de métodos**!

- **Polimorfismo**: várias implementações do **mesmo método em classes diferentes**
- **Sobrecarga**: vários métodos com o **mesmo nome** mas **assinaturas diferentes** (veremos mais à frente)

---

## 8.10 Resumo final

Conceito	Explicação
Polimorfismo	Permite usar objetos diferentes de forma genérica
@Override	Redefine um método herdado
Interface	Permite polimorfismo sem herança
Código extensível	Podes adicionar novas classes sem alterar o código principal

---

## Capítulo 9 – Composição: Quando um Objeto Tem Outro

---

### 9.1 O que é composição?

**Composição** é um princípio da POO onde uma classe é construída **a partir de outras classes**, que funcionam como **partes internas**.

Dizemos que há uma relação "**tem-um**" entre objetos.

#### Analogia simples:

Um carro **tem um** motor.

Uma escola **tem** alunos.

Um telemóvel **tem** um ecrã.

→ Estas partes não são subclasses. São **componentes**.

---

### 9.2 Composição em Java

```
public class Motor {  
    void ligar() {  
        System.out.println("Motor ligado!");  
    }  
}  
  
public class Carro {  
    private Motor motor = new Motor(); // composição  
  
    void ligar() {  
        motor.ligar(); // delega a ação ao motor  
    }  
}
```

 O Carro **contém** um Motor. Não herda dele, mas **usa-o como parte**.

---

### 9.3 Composição vs Herança

Herança (“é um”)	Composição (“tem um”)
Cria hierarquias	Cria relações internas
Forte acoplamento	Flexível e modular

Herança (“é um”)	Composição (“tem um”)
Ex: Gato é um Animal	Ex: Carro tem um Motor

💡 **Composição é geralmente preferível**, especialmente quando:

- As classes **não partilham o mesmo tipo**
  - Queres manter o sistema **modular e reutilizável**
- 

## 📦 9.4 Encapsulamento dentro da composição

Podes **encapsular os objetos internos**, controlando o acesso a eles através de métodos.

```
public class Biblioteca {
    private Livro livro;

    public Biblioteca(Livro l) {
        livro = l;
    }

    public void mostrarLivro() {
        livro.exibirInfo();
    }
}
```

➡ A Biblioteca **não expõe diretamente o objeto Livro**. Só o usa **internamente**, com controlo.

---

## 📋 9.5 Composição com listas (coleções de objetos)

```
import java.util.ArrayList;

public class Turma {
    private ArrayList<Aluno> alunos = new ArrayList<>();

    public void adicionarAluno(Aluno a) {
        alunos.add(a);
    }

    public void listarAlunos() {
        for (Aluno a : alunos) {
```

```
        a.apresentar();
    }
}
}
```

- ➡ A Turma tem vários Alunos. Cada um é tratado individualmente, mas faz parte de um todo.
- 

## ✍ 9.6 Mini-laboratório

Cria:

- Classe Endereco com rua, cidade, código postal
  - Classe Pessoa com nome e um atributo Endereco
  - No main(), cria uma pessoa e mostra o nome + morada
- 

## ⌚ 9.7 Desafio

Cria uma classe Loja que **tem uma lista de Produto**.  
Cada produto tem nome, preço e quantidade.

A loja deve:

- Adicionar produtos
  - Mostrar todos os produtos
  - Calcular o valor total em stock
- 

## ⚠ 9.8 Composição excessiva? Nem sempre.

A composição é poderosa, mas **deve ser usada com bom senso**:

- Usa se **uma classe fizer sentido como parte da outra**
- Evita se as classes forem **independentes e sem ligação semântica**

Exemplo:

- ✓ Carro tem Roda  
✗ Carro tem Cão (não faz sentido!)
-

## 9.9 Resumo final

Conceito	Explicação
Composição	Uma classe contém instâncias de outras classes
Relação “tem-um”	Ex: Pessoa tem um Endereço
Modularidade	Permite trocar ou evoluir partes sem mexer no todo
Delegação	Passa tarefas a objetos internos (ex: <code>motor.ligar()</code> dentro de Carro)

---

## Capítulo 10 – Interfaces: Contratos de Comportamento

---

### 10.1 O que é uma interface?

Uma **interface** é um tipo especial de classe que **define um conjunto de métodos que devem ser implementados**, mas **não contém código** (apenas os nomes dos métodos). Pensa nela como um **contrato**: quem assina o contrato compromete-se a cumprir aquilo que lá está definido.

#### Analogia simples:

Uma ficha elétrica tem uma forma padrão. Cada aparelho que se liga à tomada tem de **seguir esse formato**, mesmo que o interior funcione de maneira diferente.

---

### 10.2 Como se define uma interface?

```
public interface Imprimivel {  
    void imprimir(); // sem corpo!  
}
```

 Todos os métodos de uma interface são **públicos e abstratos por defeito** (não têm implementação).

---

### 10.3 Como se implementa uma interface?

Usa-se a palavra-chave **implements**:

```
public class Documento implements Imprimivel {  
    public void imprimir() {  
        System.out.println("A imprimir documento...");  
    }  
}
```

Se a classe não implementar **todos os métodos** da interface, o programa **não compila**.

---

## 10.4 Interface ≠ Herança

Interface	Herança
implements	extends
Define <b>comportamento obrigatório</b>	Reutiliza código existente
Permite implementar <b>várias interfaces</b>	Só pode estender uma classe
Não tem atributos nem código	Pode ter atributos e métodos com lógica

-  Interfaces são usadas **quando diferentes classes devem partilhar comportamentos, mas não são do mesmo tipo.**
- 

## 10.5 Interfaces no mundo real

Imagina estas situações:

- Todos os documentos devem poder **ser impressos** → Imprimivel
- Todos os objetos num jogo devem poder **mover-se** → Movivel
- Todos os formulários devem poder **validar os seus dados** → Validavel

Cada classe trata o comportamento **à sua maneira**, mas **todas concordam com o "contrato" da interface.**

---

## 10.6 Exemplo prático com várias classes

```
public interface Imprimivel {  
    void imprimir();  
}  
  
public class Relatorio implements Imprimivel {  
    public void imprimir() {  
        System.out.println("Relatório: estatísticas do mês...");  
    }  
}  
  
public class Fatura implements Imprimivel {  
    public void imprimir() {  
        System.out.println("Fatura: total a pagar €123,45");  
    }  
}
```

```
public class Impressora {  
    public static void imprimirTudo(Imprimivel i) {  
        i.imprimir();  
    }  
}
```

No main():

```
public class Main {  
    public static void main(String[] args) {  
        Imprimivel doc1 = new Relatorio();  
        Imprimivel doc2 = new Fatura();  
  
        Impressora.imprimirTudo(doc1);  
        Impressora.imprimirTudo(doc2);  
    }  
}
```

➡ Mesmo que Relatorio e Fatura sejam classes **completamente diferentes**, podemos tratá-las **de forma polimórfica**.

---

## ⌚ 10.7 Desafio

Cria:

- Uma interface Desenhavel com método desenhar()
  - Classes Circulo, Quadrado e Triangulo que implementam Desenhavel
  - Um método que receba uma lista de Desenhavel e desenhe todos os objetos
- 

## ⌚ 10.8 Interface com múltiplas implementações

Uma classe pode implementar várias interfaces:

```
public class Robo implements Movivel, Desenhavel {  
    public void mover() {  
        System.out.println("Mover para a frente");  
    }  
  
    public void desenhar() {  
        System.out.println("Desenhar no chão");  
    }  
}
```

```
    }  
}
```

- ➡ Isto simula herança múltipla, algo que Java não permite com classes mas permite com interfaces!
- 

## 📦 10.9 Interface como forma de independência

Ao programar com interfaces, crias sistemas mais:

- Flexíveis
- Testáveis
- Fáceis de manter

- ➡ Porque o código não depende da classe concreta, mas da interface (do contrato).
- 

## 🧠 10.10 Resumo final

Conceito	Explicação
Interface	Contrato que define métodos obrigatórios
implements	Palavra-chave para implementar uma interface
Polimorfismo	Permite tratar objetos diferentes de forma genérica
Flexibilidade	Podes mudar a implementação sem afetar quem usa

---

## Capítulo 11 – Classes Abstratas: Quando Só Faz Sentido Completar Depois

---

### 11.1 O que é uma classe abstrata?

Uma **classe abstrata** é uma classe que:

- **Não pode ser instanciada diretamente**
- Serve como **base para outras classes**
- Pode ter **métodos com ou sem implementação**

#### Analogia simples:

Imagina um molde para peças de Lego. O molde define que todas as peças têm encaixe, mas **não define as cores nem o tamanho específico**. Cada tipo de peça vai **completar o molde à sua maneira**.

---

### 11.2 Quando usar uma classe abstrata?

Usamos **classes abstratas** quando queremos:

- Reunir comportamento comum a várias classes
- **Obrigar** as subclasses a implementar certos métodos
- Fornecer **implementações parciais**

 Ou seja, quando **há código comum**, mas **cada subclasse precisa de detalhes próprios**.

---

### 11.3 Como se declara uma classe abstrata?

```
public abstract class Animal {  
    String nome;  
  
    public void dormir() {  
        System.out.println(nome + " está a dormir...");  
    }  
}
```

```
    public abstract void fazerSom(); // sem implementação!
}
```

➡ A palavra-chave **abstract**:

- Pode ser usada na **classe**
  - Pode ser usada em **métodos** que não têm corpo
- 

#### 🚫 11.4 Uma classe abstrata não pode ser instanciada

```
Animal a = new Animal(); // ✗ ERRO: não se pode criar um objeto  
diretamente
```

➡ Tens de criar uma subclasse concreta que **complete os métodos abstratos**.

---

#### ☑ 11.5 Implementar uma subclasse concreta

```
public class Gato extends Animal {  
    public Gato(String nome) {  
        this.nome = nome;  
    }  
  
    @Override  
    public void fazerSom() {  
        System.out.println("Miau!");  
    }  
}
```

Agora podes fazer:

```
Animal a = new Gato("Felix");  
a.fazerSom(); // Miau!  
a.dormir(); // Felix está a dormir...
```

---

#### 🔍 11.6 Diferença entre classe abstrata e interface

Característica	Classe Abstrata	Interface
Pode ter atributos e código	✓ Sim	✗ (até Java 7) / ✓ (com default em Java 8+)

Característica	Classe Abstrata	Interface
Pode ter construtor	<input checked="" type="checkbox"/> Sim	<input checked="" type="checkbox"/> Não
Pode ter métodos com lógica	<input checked="" type="checkbox"/> Sim	<input checked="" type="checkbox"/> Não (a não ser default)
Pode herdar só uma	<input checked="" type="checkbox"/> Sim (só uma)	<input checked="" type="checkbox"/> Sim (várias interfaces)

💡 **Regra prática:**

- Usa **interface** → quando estás a **definir um comportamento comum**
  - Usa **classe abstrata** → quando há **dados e comportamentos comuns reais**
- 

## 📝 11.7 Exemplo completo

```
public abstract class Funcionario {
    String nome;

    public Funcionario(String nome) {
        this.nome = nome;
    }

    public abstract double calcularSalario();
}

public class Professor extends Funcionario {
    private double salarioFixo;

    public Professor(String nome, double salario) {
        super(nome);
        this.salarioFixo = salario;
    }

    @Override
    public double calcularSalario() {
        return salarioFixo;
    }
}
```

No main():

```
Funcionario f = new Professor("Luís", 1200);
System.out.println("Salário: €" + f.calcularSalario());
```

---

## ⌚ 11.8 Desafio

Cria:

- Classe abstrata `Animal` com `fazerSom()`
- Subclasses `Cão` e `Gato`
- Testa no `main()` com polimorfismo

Bónus: adiciona `comer()` como método comum **com implementação padrão**.

---

## 🧠 11.9 Resumo final

Conceito	Explicação
Classe abstrata	Classe incompleta, usada como base
<code>abstract</code>	Palavra-chave que define classes ou métodos abstratos
Subclasse concreta	Deve implementar os métodos abstratos
Polimorfismo	Também se aplica a classes abstratas

---

## Capítulo 12 – Coleções e Estruturas de Dados: Guardar e Organizar Objetos

---

### 12.1 O que são coleções?

**Coleções** são estruturas que te permitem **guardar e organizar múltiplos objetos** de forma eficiente.

Diferente das variáveis simples, que só guardam **um valor**, as coleções permitem armazenar **muitos elementos** — como listas, conjuntos e mapas.

 **Analogia simples:**

Imagina uma mochila onde guardas livros, ou uma estante cheia de DVDs. Cada compartimento ou prateleira guarda um elemento — e podes adicionar, remover ou procurar o que quiseres.

---

### 12.2 Tipos de coleções mais usados em Java

Tipo	Interface	O que faz
Lista	List	Mantém a ordem dos elementos, permite repetições
Conjunto	Set	Não permite elementos repetidos
Mapa	Map	Guarda pares chave → valor

---

### 12.3 A classe ArrayList (lista dinâmica)

É uma das formas mais comuns de guardar objetos em sequência.

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

```
ArrayList<String> nomes = new ArrayList<>();  
nomes.add("Ana");  
nomes.add("Carlos");  
nomes.add("Beatriz");
```

- ➡ A lista guarda os elementos **na ordem em que foram adicionados**
- ➡ Podes aceder a qualquer elemento por **posição** (índice)

```
System.out.println(nomes.get(1)); // Carlos
```

---

## 12.4 Percorrer listas com ciclos for-each

```
for (String nome : nomes) {  
    System.out.println("Olá, " + nome + "!");  
}
```

Este tipo de ciclo evita erros e é muito mais legível.

---

## 12.5 Remover e modificar elementos

```
nomes.remove("Ana");           // remove pelo valor  
nomes.remove(0);              // remove pelo índice  
nomes.set(1, "Tiago");        // altera o valor no índice 1
```

---

## 12.6 Trabalhar com objetos em listas

```
ArrayList<Pessoa> pessoas = new ArrayList<>();  
  
Pessoa p1 = new Pessoa("João", 20);  
Pessoa p2 = new Pessoa("Maria", 22);  
  
pessoas.add(p1);  
pessoas.add(p2);  
  
for (Pessoa p : pessoas) {  
    p.apresentar();  
}
```

→ Isto permite ter **coleções de objetos personalizados**, com métodos e comportamentos.

---

## 12.7 O Set: coleções sem repetições

```
import java.util.HashSet;  
  
HashSet<String> cores = new HashSet<>();  
cores.add("Azul");  
cores.add("Vermelho");  
cores.add("Azul"); // Ignorado!
```

```
System.out.println(cores); // Sem duplicados!
```

- Ideal para quando queres **evitar repetições automáticas**.
- 

## 🔗 12.8 O Map: pares chave → valor

```
import java.util.HashMap;  
  
HashMap<String, String> capitais = new HashMap<>();  
capitais.put("Portugal", "Lisboa");  
capitais.put("Espanha", "Madrid");  
  
System.out.println(capitais.get("Portugal")); // Lisboa
```

- Útil quando precisas de **acesso rápido a dados associados a uma chave** (como nomes de alunos → notas, países → capitais, etc.)
- 

## 📝 12.9 Mini-laboratório

Cria:

- Classe Produto com nome, preço
  - Lista de produtos (`ArrayList<Produto>`)
  - Percorre a lista para mostrar todos os nomes e preços
  - Calcula o total acumulado dos preços
- 

## ⌚ 12.10 Desafio

Cria uma classe Agenda que guarda:

- Contactos (nome, número) usando `HashMap<String, String>`
  - Permite adicionar, listar e procurar contactos por nome
- 

## 🧠 12.11 Resumo final

Conceito	Explicação
----------	------------

Conceito	Explicação
<b>ArrayList</b>	Lista dinâmica com acesso por índice
<b>HashSet</b>	Coleção sem elementos repetidos
<b>HashMap</b>	Guarda pares chave → valor
<b>for-each</b>	Forma prática de percorrer coleções
Reutilização	Podes armazenar qualquer tipo de objeto, inclusive os que tu criaste

---

## Capítulo 13 – Tratamento de Exceções: Lidar com Erros de Forma Segura

---

### 13.1 O que são exceções?

**Exceções são erros que acontecem durante a execução do programa.**

Podem ser causadas por várias situações, como:

- Dividir por zero
- Aceder a uma posição inválida numa lista
- Ler um ficheiro que não existe
- Introduzir dados inválidos

#### Analogia simples:

Imagina que estás a conduzir. De repente aparece um buraco na estrada. Uma exceção é esse buraco.

**Ignorar pode estragar o carro** (ou o programa).

Mas se o antecipares, podes **desviar ou travar a tempo** — isso é tratamento de exceções.

---

### 13.2 Erros em tempo de execução

```
int a = 10;  
int b = 0;  
int resultado = a / b; // ERRO! Divisão por zero
```

 Este erro provoca uma **ArithmeticException**

Sem tratamento, o programa **termina abruptamente**.

---

### 13.3 Usar try-catch para proteger o programa

```
try {  
    int resultado = 10 / 0;  
    System.out.println("Resultado: " + resultado);  
} catch (ArithmeticException e) {  
    System.out.println("Erro: não se pode dividir por zero!");  
}
```

- **try**: onde o erro pode acontecer
  - **catch**: o que fazer se acontecer
  - **e**: é o objeto que representa o erro (podes ver a mensagem, o tipo, etc.)
- 

### 13.4 Bloco **finally**: sempre executado

```
try {  
    System.out.println("Tentativa de divisão...");  
    int r = 10 / 2;  
} catch (Exception e) {  
    System.out.println("Erro!");  
} finally {  
    System.out.println("Fim do bloco.");  
}
```

→ O bloco **finally** é **executado sempre**, com ou sem erro.

---

### 13.5 Exceções com entrada de dados

```
import java.util.Scanner;  
  
Scanner sc = new Scanner(System.in);  
System.out.print("Número inteiro: ");  
  
try {  
    int numero = sc.nextInt();  
    System.out.println("Número: " + numero);  
} catch (Exception e) {  
    System.out.println("Erro: tens de escrever um número inteiro!");  
}
```

→ Muito útil quando o utilizador pode escrever mal — e o programa não deve "crashar".

---

### 13.6 Exceções mais comuns

Exceção	Causa típica
<code>ArithmeticException</code>	Divisão por zero
<code>NullPointerException</code>	Usar um objeto que não foi criado

Exceção	Causa típica
<code>ArrayIndexOutOfBoundsException</code>	Aceder a índice inválido
<code>InputMismatchException</code>	Introdução de tipo de dado errado
<code>FileNotFoundException</code>	Ficheiro não existe
<code>IOException</code>	Erro genérico de entrada/saída

---

### 13.7 Criar exceções personalizadas

```
public class IdadeInvalidaException extends Exception {
    public IdadeInvalidaException(String mensagem) {
        super(mensagem);
    }
}
```

Usar assim:

```
public void setIdade(int idade) throws IdadeInvalidaException {
    if (idade < 0) {
        throw new IdadeInvalidaException("Idade não pode ser negativa!");
    }
    this.idade = idade;
}
```

 Isto permite criar **regras de negócio** com mensagens mais claras e lógicas.

---

### 13.8 Mini-laboratório

Cria um programa que:

- Lê dois números
  - Divide o primeiro pelo segundo
  - Usa `try-catch` para evitar divisão por zero
  - Mostra uma mensagem amigável em caso de erro
- 

### 13.9 Desafio

Cria:

- Classe `ContaBancaria` com método `levantar(double valor)`

- Se o saldo for insuficiente, lança uma exceção personalizada `SaldoInsuficienteException`
  - No `main()`, testa o comportamento com e sem erro
- 

## 13.10 Resumo final

Conceito	Explicação
Exceção	Erro em tempo de execução
<code>try-catch</code>	Tenta uma operação e captura o erro, se ocorrer
<code>finally</code>	Executa sempre, mesmo que haja erro
Exceções personalizadas	Permitem criar regras de erro próprias
Robustez	Um bom programa não falha com entradas inesperadas

---

## Capítulo 14 – Interface Gráfica com Swing (GUI)

*Dar vida ao teu programa: de texto... para janelas interativas!*

---

### 14.1 O que é o Swing?

O **Swing** é uma biblioteca gráfica incluída no Java, usada para criar **interfaces visuais**: janelas, botões, menus, caixas de texto, listas e até **gráficos simples**.

 Até agora escrevemos programas em **modo de texto**, no terminal. Com o Swing, podemos criar **programas com janelas, botões e interatividade visual**, como qualquer aplicação moderna.

---

### 14.2 O essencial para começar

*Importações básicas:*

```
import javax.swing.*; // Componentes principais (JFrame, JButton, etc.)  
import java.awt.*; // Layouts, cores, fontes  
import java.awt.event.*; // Eventos (cliques, teclas, etc.)
```

*A tua primeira janela:*

```
import javax.swing.*;  
  
public class JanelaSimples {  
    public static void main(String[] args) {  
        JFrame janela = new JFrame("Olá Swing!");  
        janela.setSize(300, 200); // largura, altura  
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        janela.setVisible(true); // mostra a janela  
    }  
}
```

 **JFrame** é a **janela principal**.

Tens de a configurar, dar-lhe tamanho, título... e torná-la visível!

---

### 14.3 Adicionar botões e interações

```
import javax.swing.*;
import java.awt.event.*;

public class JanelaComBotao {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Botão");
        JButton botao = new JButton("Clica-me!");

        botao.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(frame, "Olá, mundo!");
            }
        });

        frame.add(botao); // adiciona o botão à janela
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

#### Componentes comuns:

Componente	Classe Swing
Janela principal	JFrame
Botão	JButton
Etiqueta (texto)	JLabel
Caixa de texto	JTextField
Caixa de texto grande	JTextArea
Lista suspensa	JComboBox
Caixa de seleção	JCheckBox
Botões de opção	JRadioButton
Painéis de organização	JPanel

---

### 14.4 Layouts: organizar os componentes

*FlowLayout (padrão)*

```
frame.setLayout(new FlowLayout());
```

```
BorderLayout (5 zonas: Norte, Sul, Centro, Este, Oeste)
frame.add(botao, BorderLayout.SOUTH);

GridLayout (grelha tipo tabela)
frame.setLayout(new GridLayout(2, 2)); // 2 linhas, 2 colunas
```

---

## 14.5 Formulário simples com campos e botão

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Formulario {
    public static void main(String[] args) {
        JFrame janela = new JFrame("Formulário");
        janela.setLayout(new FlowLayout());

        JLabel lblNome = new JLabel("Nome:");
        JTextField txtNome = new JTextField(20);
        JButton btnEnviar = new JButton("Enviar");

        btnEnviar.addActionListener(e -> {
            String nome = txtNome.getText();
            JOptionPane.showMessageDialog(janela, "Olá, " + nome + "!");
        });

        janela.add(lblNome);
        janela.add(txtNome);
        janela.add(btnEnviar);

        janela.setSize(300, 150);
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        janela.setVisible(true);
    }
}
```

---

## 14.6 Como desenhar um gráfico simples

Para fazer gráficos, criamos uma **subclasse de JPanel** onde desenharemos o que quisermos usando **Graphics**.

```

import javax.swing.*;
import java.awt.*;

class PainelGrafico extends JPanel {
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.BLUE);
        g.fillRect(50, 50, 100, 100); // x, y, largura, altura
        g.setColor(Color.RED);
        g.drawString("Gráfico simples!", 60, 45);
    }
}

```

Depois, usamos este painel na janela:

```

public class JanelaGrafico {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Gráfico");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        PainelGrafico painel = new PainelGrafico();
        frame.add(painel);

        frame.setVisible(true);
    }
}

```

➡ Este painel pode ser usado para **desenhar barras, pontos, formas e até gráficos de linhas**.

Para gráficos mais avançados, usa bibliotecas como **JFreeChart** (fora do alcance deste curso, mas fácil de integrar).

## ⌚ 14.7 Desafio

Cria uma janela com:

- Um campo de texto para introduzir o nome
- Um botão que, quando clicado, mostra uma mensagem de boas-vindas
- Um botão extra que fecha a aplicação (`System.exit(0)`)

## 14.8 Resumo final

Conceito	Explicação
JFrame	Janela principal
JButton, JLabel, JTextField	Componentes visuais
addActionListener	Responde a cliques ou ações
JOptionPane	Mostra caixas de diálogo
JPanel + Graphics	Permite desenhar manualmente gráficos ou formas
Layout	Organiza os componentes (Flow, Border, Grid)

---

## Capítulo 15 – Boas Práticas e Estruturação de Projetos

---

### 15.1 Porque é que estrutura e boas práticas são tão importantes?

Criar um programa que funciona é apenas o primeiro passo.

Mas um código bem **estruturado, legível e reutilizável** é o que te transforma num verdadeiro programador orientado a objetos.

#### Pensa no teu projeto como uma casa:

- Se for mal construída, desaba ao primeiro problema.
  - Se for bem pensada, é fácil de manter, ampliar e explicar a outros.
- 

### 15.2 Separar bem as classes

 Cada classe deve representar uma ideia clara (uma entidade, um papel, uma função).

Exemplos:

- Aluno, Professor, Curso → entidades
- Calculadora, Relatorio, Impressora → utilitários
- Main, App, TesteXYZ → pontos de entrada ou testes

 **Regra de ouro: 1 ficheiro por classe pública**, com o **mesmo nome da classe**.

---

### 15.3 Encapsular tudo o que puder

Não deixes atributos públicos.

Controla o acesso com **private** e oferece métodos **get/set** **apenas quando fizer sentido**.

 Um bom objeto:

- Protege os seus dados
  - Expõe apenas o que é essencial
  - Valida o que entra
-

## 15.4 Usar construtores com lógica

Evita isto:

```
Pessoa p = new Pessoa();  
p.nome = "João";  
p.idade = 0;
```

Prefere isto:

```
Pessoa p = new Pessoa("João", 0);
```

- Torna o código **mais limpo, legível e menos propenso a erros.**
- 

## 15.5 Relacionar classes com sentido

Antes de usar **herança**, pergunta:

"Faz sentido dizer que X é um Y?"

Se sim → usa **extends**

Se não → talvez estejas perante uma **composição** (X tem um Y)

Exemplo:

- Carro extends Veiculo ✓
  - Carro tem um Motor ✓
  - Carro extends Motor ✗
- 

## 15.6 Usar listas e mapas com significado

Preferir coleções quando:

- Tens muitos elementos do mesmo tipo (**List<Produto>**)
- Precisas de associar valores a nomes/chaves (**Map<String, Aluno>**)

- Nomeia bem as variáveis e escolhe o tipo certo de coleção!
- 

## 15.7 Comentar com utilidade

Não comentes o óbvio:

```
int idade = 20; // define a idade ← X
```

Comenta onde a **intenção ou a lógica não for imediata**:

```
// Valida se o utilizador tem idade suficiente para votar
if (idade >= 18) {
    permitirVoto();
}
```

➡ E usa **JavaDoc** nos métodos e classes principais:

```
/*
 * Representa um aluno com nome e nota final.
 */
public class Aluno {
    private String nome;
    private double nota;

    /**
     * Cria um aluno com nome e nota.
     */
    public Aluno(String nome, double nota) { ... }
}
```

---

## ⌚ 15.8 Escrever código limpo

- ✓ Usa nomes significativos (`quantidadeProdutos`, `calcularMedia`)
  - ✓ Evita duplicação de código
  - ✓ Divide funções grandes em partes pequenas
  - ✓ Indenta bem o código — facilita a leitura!
- 

## ✍ 15.9 Mini-laboratório

Pega num dos teus projetos anteriores (ex: gestão de produtos, biblioteca, banco...) e:

- Reorganiza o código em várias classes
  - Usa encapsulamento corretamente
  - Acrescenta comentários JavaDoc úteis
  - Evita repetições e melhora os nomes das variáveis
-

## 15.10 Desafio

Cria um projeto simples com as seguintes características:

- 3 ou mais classes com relações claras (`Pessoa`, `Livro`, `Emprestimo`)
- Uso correto de construtores, `get/set` e encapsulamento
- Um `main()` que demonstre os comportamentos
- JavaDoc nas classes e métodos principais

Este projeto pode ser o **esqueleto para o teu trabalho final!** 

---

## 15.11 Resumo final

Prática	Explicação
Separar classes	Uma classe por ficheiro e por responsabilidade
Encapsular bem	Atributos privados + métodos controlados
Comentários úteis	JavaDoc nos métodos públicos, comentários nas partes complexas
Construtores limpos	Criar objetos com dados válidos logo no início
Relações lógicas	Usar herança e composição com critério



Luís Simões da Cunha (2025)