E-BOOK
Java Iniciante

Dominando o Java Básico

Explorando Variáveis, Operadores, Controle de Fluxo a Estruturas Fundamentais da Linguagem Java

Ana Raquel de Holanda

Powered by
ChatGPT & Engenh

✍️ Sobre a Autora

Estudante de Java há 7 meses, iniciei minha jornada na linguagem por meio da escola de formação de desenvolvedores Fuctura. Além disso, sou assinante da plataforma Dio, onde aprofundo meus conhecimentos e domino as tecnologias por trás das IAs. Utilizando a Engenharia de Prompts, elaborei este e-book com o auxílio do ChatGPT, construindo cada capítulo de forma interativa. É com prazer e satisfação que entrego a você este material, criado para desenvolvedores que desejam ir muito além dos conceitos básicos de Java. 😉

📖 Introdução

Este e-book é destinado a iniciantes que desejam compreender os fundamentos da linguagem Java. Baseado na documentação oficial do Dev.java, abordaremos desde a criação de variáveis até estruturas de controle de fluxo, proporcionando uma base sólida para o desenvolvimento em Java.

Sumário

📖 Capítulo 1 – Introdução ao Java

1.1 O que é Java?

Java é uma linguagem de programação de propósito geral, orientada a objetos, desenvolvida pela Sun Microsystems em 1995 (hoje mantida pela Oracle). É uma das linguagens mais populares no mundo da tecnologia, utilizada para desenvolver desde aplicações desktop até aplicações empresariais robustas e sistemas embarcados.

💡 **Dica**: A portabilidade é uma das maiores vantagens do Java — ou seja, o mesmo código pode ser executado em diferentes sistemas operacionais sem modificações significativas.

1.2 Características Principais

- **Portabilidade (Write Once, Run Anywhere – WORA)**
  ➡️ O código Java é compilado em bytecode, que é executado na Java Virtual Machine (JVM), garantindo independência de plataforma.
- **Orientação a Objetos**
  ➡️ Java foi projetado para ser uma linguagem orientada a objetos, promovendo a reutilização de código, modularidade e facilidade de manutenção.
- **Segurança**
  ➡️ O Java fornece mecanismos robustos de segurança, como a verificação de bytecode e o gerenciamento automático de memória.
- **Linguagem de alto nível**
  ➡️ Fácil de aprender, com sintaxe clara e bem definida.

💡 **Dica**: Apesar de ser considerado fácil de aprender, dominar Java requer prática constante — especialmente para entender como os frameworks e as APIs modernas se integram ao ecossistema Java.

1.3 Histórico e Evolução

- **1995**: Java é lançado pela Sun Microsystems.
- **1996-2000**: Ganha popularidade com a promessa de criar applets para web.
- **2004**: Introdução do Java 5 com melhorias na linguagem (generics, annotations).
- **2009**: Oracle adquire a Sun Microsystems.
- **2017 em diante**: Java adota um ciclo de lançamentos mais rápido (Java 9+).

1.4 Configuração do Ambiente de Desenvolvimento

Antes de escrever qualquer código Java, é necessário configurar o ambiente de desenvolvimento.

1.4.1 Instalando o JDK

1️⃣ Baixe o **JDK** (Java Development Kit) do site oficial da Oracle ou do Dev.java.
2️⃣ Siga as instruções de instalação para seu sistema operacional (Windows, Linux ou macOS).

💡 **Dica**: Recomendo instalar a versão LTS (Long Term Support), como o Java 17 ou superior, para ter estabilidade e suporte prolongado.

1.4.2 Configurando o PATH

- No Windows, adicione a pasta `bin` do JDK às variáveis de ambiente (PATH).
- No Linux/macOS, adicione no `.bashrc` ou `.zshrc`:

```
export JAVA_HOME=/caminho/do/jdk
export PATH=$JAVA_HOME/bin:$PATH
```

💡 **Dica**: Após configurar, execute `java -version` e `javac -version` no terminal para verificar se tudo está funcionando.

1.5 O Primeiro Programa em Java

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

📌 **Explicação**:

- `public class HelloWorld`: Define uma classe pública chamada HelloWorld.
- `public static void main(String[] args)`: O ponto de entrada para a execução do programa.
- `System.out.println("Olá, mundo!");`: Imprime o texto no console.

💡 **Dica**: Salve o arquivo como `HelloWorld.java` e compile com:

```
javac HelloWorld.java
java HelloWorld
```

1.6 Revisão do Capítulo

Neste capítulo, você aprendeu:
✅ O que é a linguagem Java e suas principais características.

✅ O histórico e a evolução da linguagem.
✅ Como configurar o ambiente de desenvolvimento Java.
✅ Como escrever, compilar e executar o primeiro programa em Java.

💡 **Próximo Passo:** No próximo capítulo, vamos nos aprofundar em **variáveis e tipos de dados**, essenciais para qualquer aplicação Java.

📖 Capítulo 2 – Variáveis e Tipos de Dados

2.1 O que são Variáveis?

Em Java, variáveis são espaços na memória do computador reservados para armazenar dados temporários durante a execução do programa. Elas possuem **nome**, **tipo** e **valor**.

💡 **Dica**: O nome da variável deve seguir boas práticas de nomenclatura — ser descritivo, começar com letra minúscula e, se composto, utilizar o padrão camelCase (ex.: `idadeAluno`).

2.2 Tipos de Dados

Os tipos de dados em Java são divididos em:

1️⃣ **Tipos Primitivos**
2️⃣ **Tipos de Referência (Objetos)**

2.2.1 Tipos Primitivos

São os tipos básicos da linguagem, não possuem métodos associados e armazenam diretamente os valores na memória.

Tipo	Tamanho	Faixa de Valores
`byte`	8 bits	-128 a 127
`short`	16 bits	-32.768 a 32.767
`int`	32 bits	-2^{31} a $2^{31}-1$
`long`	64 bits	-2^{63} a $2^{63}-1$
`float`	32 bits	Precisão de até 7 casas decimais
`double`	64 bits	Precisão de até 15 casas decimais
`char`	16 bits	Um único caractere Unicode (ex.: 'A', 'b', '3')
`boolean`	1 bit	`true` ou `false`

💡 **Dica**: Use `int` como tipo padrão para números inteiros e `double` para números reais, a menos que tenha restrições de memória ou necessidades específicas de precisão.

2.2.2 Tipos de Referência

São usados para armazenar referências (endereços de memória) de objetos e arrays. Exemplos:

- Objetos: `String`, `Scanner`, `List`, `Map`, etc.
- Arrays: `int[]`, `String[]`, etc.

💡 **Dica**: Diferentemente dos tipos primitivos, os tipos de referência possuem métodos e podem ser manipulados através da **Programação Orientada a Objetos (POO)**.

2.3 Declarando Variáveis

A sintaxe básica para declarar uma variável é:

```
tipo nomeDaVariavel;
```

Exemplos:

```
int idade;
double salario;
char letra;
```

2.4 Inicializando Variáveis

É possível declarar e inicializar variáveis na mesma linha:

```
int idade = 25;
double salario = 4500.75;
char letra = 'A';
boolean ativo = true;
```

💡 **Dica**: Sempre inicialize as variáveis antes de usá-las para evitar erros de compilação ou comportamentos inesperados.

2.5 Convenções de Nomenclatura

✔️ Use nomes significativos:

- `idadeAluno` em vez de `x`
- `salarioMensal` em vez de `s`

✔️ Evite nomes de variáveis que colidam com palavras reservadas do Java (ex.: `class`, `public`, `static`).

2.6 Exemplo Prático

Vamos declarar e inicializar algumas variáveis em um programa:

```
public class VariaveisDemo {
    public static void main(String[] args) {
        int idade = 30;
        double salario = 3500.50;
        char genero = 'F';
        boolean ativo = true;

        System.out.println("Idade: " + idade);
        System.out.println("Salário: " + salario);
        System.out.println("Gênero: " + genero);
        System.out.println("Ativo: " + ativo);
    }
}
```

📌 **Explicação**:

- `int idade = 30;` ➡️ Declaração e inicialização de variável inteira.
- `double salario = 3500.50;` ➡️ Valor real com ponto decimal.
- `char genero = 'F';` ➡️ Um caractere Unicode.
- `boolean ativo = true;` ➡️ Variável booleana.

2.7 Boas Práticas

✅ Declare variáveis próximas do local onde são usadas.
✅ Use nomes claros e objetivos.
✅ Inicialize variáveis sempre que possível.
✅ Prefira tipos primitivos quando a simplicidade for suficiente.

💡 **Dica**: Para manipular números grandes ou cálculos financeiros, estude as classes `BigDecimal` e `BigInteger` (não são tipos primitivos, mas são muito usados no mercado).

2.8 Revisão do Capítulo

Neste capítulo, você aprendeu:
✅ O que são variáveis em Java.
✅ Diferença entre tipos primitivos e tipos de referência.
✅ Como declarar, inicializar e nomear variáveis.
✅ Boas práticas de uso de variáveis em aplicações Java.

💡 **Próximo Passo:** No próximo capítulo, exploraremos os **Operadores em Java**, fundamentais para construir expressões e realizar cálculos em seus programas.

📖 Capítulo 3 – Operadores em Java

3.1 O que são Operadores?

Operadores são símbolos especiais que realizam operações em variáveis e valores, como cálculos aritméticos, comparações lógicas ou atribuições. São fundamentais para criar expressões e tomar decisões no código.

💡 **Dica**: Os operadores em Java são agrupados por funcionalidade: **aritméticos**, **relacionais**, **lógicos**, **unários**, **ternários** e **de atribuição**.

3.2 Operadores Aritméticos

Permitem realizar cálculos matemáticos:

Operador	Descrição	Exemplo (`int a = 10; int b = 3;`)
`+`	Adição	`a + b // 13`
`-`	Subtração	`a - b // 7`
`*`	Multiplicação	`a * b // 30`
`/`	Divisão	`a / b // 3` (divisão inteira)
`%`	Módulo (resto da divisão)	`a % b // 1`

💡 **Dica**: Em divisão inteira, a parte decimal é descartada. Para resultados decimais, use `double` ou `float`.

3.3 Operadores Relacionais

Usados para comparar valores e retornar um resultado booleano (`true` ou `false`).

Operador	Descrição	Exemplo (`a = 10, b = 3`)
`==`	Igualdade	`a == b // false`
`!=`	Diferente	`a != b // true`
`>`	Maior que	`a > b // true`
`<`	Menor que	`a < b // false`
`>=`	Maior ou igual	`a >= b // true`
`<=`	Menor ou igual	`a <= b // false`

💡 **Dica**: Use operadores relacionais em estruturas de controle como `if`, `while` e `for`.

3.4 Operadores Lógicos

Permitem combinar expressões booleanas.

Operador	Descrição	Exemplo
`&&`	E lógico (AND)	`(a > 5) && (b < 5) // true`

Operador	Descrição	Exemplo
!	Negação (NOT)	`!(a < 5) // true`

💡 **Dica**: Use parênteses para deixar a expressão mais clara e evitar erros de precedência.

3.5 Operadores Unários

São aplicados a apenas uma variável.

Operador	Descrição	Exemplo (`int x = 5;`)
+	Valor positivo	`+x // 5`
-	Negativo	`-x // -5`
++	Incremento	`x++ // pós-incremento`
--	Decremento	`x-- // pós-decremento`
!	Negação lógica (boolean)	`!true // false`

💡 **Dica**: O `++` e `--` podem ser prefixados (`++x`) ou postfixados (`x++`), alterando a ordem de avaliação.

3.6 Operador Ternário

Uma forma compacta de usar o `if-else`.

```
resultado = (condição) ? valorSeVerdadeiro : valorSeFalso;
```

💡 **Dica**: Use o operador ternário para atribuições simples e diretas, evitando usar em lógicas complexas para manter a legibilidade.

Exemplo:

```
int idade = 18;
String status = (idade >= 18) ? "Adulto" : "Menor";
```

3.7 Operadores de Atribuição

Atribuem valores às variáveis.

Operador	Descrição	Exemplo (`x = 10;`)
=	Atribuição simples	`x = 5;`
+=	Soma e atribui	`x += 3; // x = x + 3`
-=	Subtrai e atribui	`x -= 2; // x = x - 2`
*=	Multiplica e atribui	`x *= 2; // x = x * 2`
/=	Divide e atribui	`x /= 5; // x = x / 5`
%=	Resto e atribui	`x %= 3; // x = x % 3`

💡 **Dica**: Use operadores compostos para tornar o código mais conciso.

3.8 Precedência de Operadores

Define a ordem de avaliação das expressões.

📌 **Ordem de precedência (do mais alto para o mais baixo):**
1️⃣ `( )` (parênteses)
2️⃣ `++`, `--`, `+`, `-`, `!` (unários)
3️⃣ `*`, `/`, `%`
4️⃣ `+`, `-`
5️⃣ `>`, `<`, `>=`, `<=`
6️⃣ `==`, `!=`
7️⃣ `&&`
8️⃣ `||`
9️⃣ `=`, `+=`, `-=`, etc.

💡 **Dica**: Sempre utilize parênteses para tornar a expressão mais clara e reduzir a possibilidade de erro.

3.9 Exemplo Prático

```
public class OperadoresDemo {
    public static void main(String[] args) {
        int a = 10;
        int b = 3;
        double resultado = (a + b) * 2 / 3.0;

        boolean maior = a > b;
        boolean ativo = true;
        boolean condicao = (maior && ativo) || (b == 3);

        System.out.println("Resultado: " + resultado);
        System.out.println("Maior que: " + maior);
        System.out.println("Condição final: " + condicao);
    }
}
```

📌 **Explicação**:

- `(a + b) * 2 / 3.0`: Operadores aritméticos com precedência e conversão automática para `double`.
- `maior && ativo`: Operador lógico para avaliar se `a > b` e `ativo` são verdadeiros.

3.10 Revisão do Capítulo

Neste capítulo, você aprendeu:
✅ Operadores aritméticos, relacionais, lógicos, unários e ternário.
✅ Como e quando usar cada tipo de operador.
✅ Precedência de operadores para evitar erros em expressões complexas.
✅ Boas práticas para legibilidade e clareza de código.

💡 **Próximo Passo:** No próximo capítulo, abordaremos **Estruturas de Controle de Fluxo (if, switch, while, for)**, essenciais para a tomada de decisões em seus programas.

📖 Capítulo 4 – Estruturas de Controle de Fluxo

4.1 Introdução

Estruturas de controle de fluxo são fundamentais para direcionar o caminho que o programa deve seguir, permitindo tomar decisões, executar comandos repetidamente ou escolher entre diferentes opções.

💡 **Dica**: As principais estruturas de controle em Java são:

- **if-else**
- **switch**
- **while**
- **do-while**
- **for**

4.2 Estrutura Condicional – if-else

Permite executar blocos de código com base em uma condição booleana.

📌 **Sintaxe:**

```
if (condição) {
    // bloco de código executado se condição for verdadeira
} else {
    // bloco de código executado se condição for falsa
}
```

💡 **Dica**: O `else` é opcional. Você pode encadear vários `if-else if-else` para testar diferentes condições.

📌 **Exemplo:**

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

4.3 Estrutura Condicional – switch

Útil para escolher entre múltiplas opções de forma mais organizada.

📌 **Sintaxe:**

```
switch (expressão) {
    case valor1:
        // código
        break;
    case valor2:
        // código
        break;
    default:
        // código
}
```

💡 **Dica**: O `break` impede que o fluxo continue para o próximo `case` (chamado **fall-through**).

📌 **Exemplo:**

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

4.4 Estruturas de Repetição – while

Executa um bloco de código enquanto uma condição for verdadeira.

📌 **Sintaxe:**

```
while (condição) {
    // bloco de código
}
```

📌 **Exemplo:**

```
int contador = 0;
while (contador < 5) {
    System.out.println("Contador: " + contador);
    contador++;
}
```

4.5 Estruturas de Repetição – do-while

Semelhante ao `while`, mas garante a execução do bloco de código pelo menos uma vez.

📌 **Sintaxe:**

```
do {
    // bloco de código
} while (condição);
```

📌 **Exemplo:**

```
int contador = 0;
do {
    System.out.println("Contador: " + contador);
    contador++;
} while (contador < 5);
```

4.6 Estruturas de Repetição – for

Usada quando sabemos o número de repetições antecipadamente.

📌 **Sintaxe:**

```
for (inicialização; condição; incremento) {
    // bloco de código
}
```

📌 **Exemplo:**

```
for (int i = 0; i < 5; i++) {
    System.out.println("Contador: " + i);
}
```

💡 **Dica**: O `for` é muito usado para percorrer arrays e coleções.

4.7 Controle de Fluxo com break e continue

- **break**: Interrompe o loop ou o switch e pula para o próximo bloco de código.
- **continue**: Interrompe a iteração atual e pula para a próxima repetição.

📌 **Exemplo de uso do `break`:**

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break; // interrompe o loop
    }
    System.out.println("i: " + i);
}
```

📌 **Exemplo de uso do `continue`:**

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
```

```
            continue; // pula os números pares
        }
        System.out.println("Número ímpar: " + i);
    }
```

4.8 Revisão do Capítulo

Neste capítulo, você aprendeu:
✅ Como usar `if`, `else if` e `else` para tomar decisões.
✅ Como usar `switch` para lidar com múltiplos casos.
✅ Como usar `while`, `do-while` e `for` para criar laços de repetição.
✅ Como usar `break` e `continue` para controlar o fluxo de execução.

💡 **Próximo Passo:** No próximo capítulo, vamos abordar **Arrays e Strings**, essenciais para trabalhar com coleções de dados e textos em Java.

📖 Capítulo 5 – Arrays e Strings

5.1 Introdução

Neste capítulo, exploraremos os conceitos fundamentais de arrays (vetores) e Strings em Java. Arrays permitem armazenar múltiplos valores do mesmo tipo de dados, enquanto as Strings lidam com sequências de caracteres.

💡 **Dica**: Arrays são ideais para lidar com coleções fixas de dados homogêneos (mesmo tipo), enquanto as Strings são frequentemente usadas para manipular textos.

5.2 Arrays em Java

Arrays são estruturas de dados que armazenam múltiplos valores em uma única variável, utilizando um índice numérico para acessar cada elemento.

📌 **Sintaxe de declaração:**

```
tipo[] nomeDoArray;
nomeDoArray = new tipo[tamanho];
```

Ou:

```
tipo[] nomeDoArray = new tipo[tamanho];
```

💡 **Dica**: O índice de um array em Java começa em **0** e vai até **tamanho - 1**.

📌 **Exemplo:**

```
int[] numeros = new int[5];
numeros[0] = 10;
numeros[1] = 20;
numeros[2] = 30;
numeros[3] = 40;
numeros[4] = 50;
```

💡 **Dica**: Você também pode inicializar diretamente:

```
int[] numeros = {10, 20, 30, 40, 50};
```

5.3 Acessando elementos do array

Use o índice do array para acessar ou modificar elementos.

📌 **Exemplo:**

```
System.out.println("Primeiro elemento: " + numeros[0]); // 10
numeros[2] = 35; // Modifica o terceiro elemento
```

5.4 Iterando sobre um array

Geralmente utilizamos laços `for` para percorrer os elementos.

📌 **Exemplo:**

```
for (int i = 0; i < numeros.length; i++) {
    System.out.println("Elemento na posição " + i + ": " + numeros[i]);
}
```

💡 **Dica**: A propriedade `length` retorna o tamanho do array.

5.5 Arrays multidimensionais

Podemos criar arrays de duas ou mais dimensões.

📌 **Exemplo de array bidimensional (matriz):**

```
int[][] matriz = new int[2][3];
matriz[0][0] = 1;
matriz[0][1] = 2;
matriz[0][2] = 3;
matriz[1][0] = 4;
matriz[1][1] = 5;
matriz[1][2] = 6;
```

📌 **Iterando em uma matriz:**

```
for (int i = 0; i < matriz.length; i++) {
    for (int j = 0; j < matriz[i].length; j++) {
        System.out.println("Elemento[" + i + "][" + j + "]: " + matriz[i][j]);
```

```
    }
}
```

5.6 Strings em Java

Strings são objetos que representam sequências de caracteres.

📌 **Criação de String:**

```
String saudacao = "Olá, mundo!";
```

💡 **Dica**: Strings são imutáveis — qualquer alteração cria um novo objeto.

5.7 Métodos úteis da classe String

Método	Descrição	Exemplo
`length()`	Retorna o tamanho da String	`saudacao.length() // 11`
`charAt(int index)`	Retorna o caractere na posição `index`	`saudacao.charAt(0) // 'O'`
`toUpperCase()`	Converte para maiúsculas	`saudacao.toUpperCase()`
`toLowerCase()`	Converte para minúsculas	`saudacao.toLowerCase()`
`substring(int, int)`	Extrai parte da String	`saudacao.substring(0, 3) // "Olá"`
`equals(String)`	Compara Strings (sensível a maiúsculas)	`"Java".equals("java") // false`
`equalsIgnoreCase(String)`	Compara ignorando maiúsculas/minúsculas	`"Java".equalsIgnoreCase("java") // true`
`contains(String)`	Verifica se contém uma substring	`"Hello".contains("ll") // true`

5.8 Concatenando Strings

Podemos concatenar Strings com o operador `+`.

📌 **Exemplo:**

```
String nome = "Ana";
String mensagem = "Bem-vinda, " + nome + "!";
System.out.println(mensagem);
```

5.9 Boas Práticas

✅ Utilize `equals()` ou `equalsIgnoreCase()` para comparar Strings, e não `==`.
✅ Use laços `for-each` quando possível para tornar o código mais legível.
✅ Documente os índices e dimensões dos arrays para facilitar a manutenção.

5.10 Revisão do Capítulo

Neste capítulo, você aprendeu:
✅ Como criar e manipular arrays unidimensionais e multidimensionais.
✅ Como acessar elementos e percorrer arrays com laços.
✅ Como criar, comparar e manipular Strings em Java.
✅ Métodos essenciais da classe String.

💡 **Próximo Passo:** No próximo capítulo, vamos explorar **Métodos e Estruturas de Programação Modular**, essenciais para organizar e reutilizar código.

📖 Capítulo 6 – Métodos e Estruturas de Programação Modular

6.1 Introdução

Neste capítulo, exploraremos como estruturar programas em Java de forma modular, utilizando métodos. Métodos permitem dividir o código em partes menores e mais gerenciáveis, promovendo reutilização e legibilidade.

💡 **Dica**: Uma boa estrutura modular torna o código mais limpo e fácil de manter.

6.2 O que são Métodos?

Métodos são blocos de código que realizam uma tarefa específica e podem ser chamados repetidamente dentro do programa.

📌 **Sintaxe básica de um método:**

```
tipoDeRetorno nomeDoMetodo(parâmetros) {
    // bloco de código
}
```

💡 **Dica**: Se o método não retorna nenhum valor, use o tipo de retorno `void`.

6.3 Declaração e Uso de Métodos

📌 **Exemplo:**

```
// Método que exibe uma mensagem
void saudacao() {
    System.out.println("Bem-vindo ao Java!");
}
```

📌 **Chamando o método:**

```
saudacao();
```

💡 **Dica**: O uso de métodos ajuda a evitar duplicação de código.

6.4 Métodos com Parâmetros

Podemos passar informações para o método através de parâmetros.

📌 **Exemplo:**

```
void exibirMensagem(String nome) {
    System.out.println("Olá, " + nome + "!");
}
```

📌 **Chamando o método:**

```
exibirMensagem("Ana");
```

6.5 Métodos com Retorno de Valor

Quando precisamos que o método devolva um valor, definimos um tipo de retorno.

📌 **Exemplo:**

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

📌 **Chamando o método:**

```
int resultado = somar(5, 3);
System.out.println("Soma: " + resultado);
```

6.6 Escopo de Variáveis

Variáveis declaradas dentro de um método são **locais** e não podem ser acessadas fora dele.

💡 **Dica**: O escopo define onde a variável é visível no código.

📌 **Exemplo:**

```
void exemploEscopo() {
    int x = 10; // x é local e só existe dentro deste método
    System.out.println(x);
}
// System.out.println(x); // Erro: x não é visível aqui
```

6.7 Sobrecarga de Métodos (Overloading)

Java permite criar métodos com o mesmo nome, desde que tenham diferentes assinaturas (quantidade ou tipo de parâmetros).

📌 **Exemplo:**

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

double somar(double a, double b) {
    return a + b;
}
```

💡 **Dica**: A sobrecarga aumenta a flexibilidade do código.

6.8 Boas Práticas

✅ Dê nomes descritivos aos métodos, utilizando verbos no infinitivo (ex.: `calcularMedia`, `exibirRelatorio`).
✅ Cada método deve realizar **apenas uma tarefa**.
✅ Evite métodos muito longos; se necessário, divida em submétodos.

6.9 Revisão do Capítulo

Neste capítulo, você aprendeu:
✅ O que são métodos e por que são importantes para modularidade.
✅ Como declarar métodos com ou sem parâmetros e valores de retorno.
✅ Sobre o escopo de variáveis e a importância de gerenciá-lo corretamente.
✅ Como usar a sobrecarga de métodos para maior flexibilidade.

💡 **Próximo Passo:** No próximo capítulo, abordaremos **Classes e Objetos**, o cerne da programação orientada a objetos em Java.

📖 Capítulo 7 – Classes e Objetos

7.1 Introdução

A Programação Orientada a Objetos (POO) é um paradigma que organiza o código em unidades chamadas **classes**, que descrevem **objetos** do mundo real. Em Java, tudo gira em torno de classes e objetos.

💡 **Dica**: POO facilita a modelagem de problemas complexos, aumentando a modularidade e a reutilização de código.

7.2 O que são Classes?

Uma classe é um molde ou uma estrutura que define as características (atributos) e comportamentos (métodos) de um objeto.

📌 **Sintaxe básica:**

```
class NomeDaClasse {
    // Atributos (variáveis de instância)
    // Métodos (comportamentos)
}
```

7.3 Criando Objetos

Um objeto é uma instância de uma classe. Ele representa uma entidade que combina estado (atributos) e comportamento (métodos).

📌 **Exemplo:**

```
class Pessoa {
    String nome;
    int idade;

    void apresentar() {
        System.out.println("Olá, meu nome é " + nome + " e tenho " + idade + "
anos.");
    }
}

// Criando objetos:
Pessoa pessoa1 = new Pessoa();
pessoa1.nome = "Ana";
pessoa1.idade = 25;
pessoa1.apresentar();
```

💡 **Dica**: Cada objeto criado com `new` tem seu próprio conjunto de atributos.

7.4 Construtores

Construtores são métodos especiais que inicializam os objetos. Eles têm o mesmo nome da classe e **não** têm tipo de retorno.

📌 **Exemplo:**

```
class Pessoa {
    String nome;
    int idade;

    // Construtor
    Pessoa(String n, int i) {
        nome = n;
        idade = i;
    }

    void apresentar() {
        System.out.println("Olá, meu nome é " + nome + " e tenho " + idade + "
anos.");
    }
}

// Uso:
Pessoa pessoa2 = new Pessoa("Carlos", 30);
pessoa2.apresentar();
```

7.5 Encapsulamento

O encapsulamento visa proteger os dados da classe, tornando-os privados e acessíveis apenas através de métodos públicos chamados **getters** e **setters**.

📌 **Exemplo:**

```
class ContaBancaria {
    private double saldo;

    public void depositar(double valor) {
        saldo += valor;
    }

    public double getSaldo() {
        return saldo;
    }
}
```

💡 **Dica**: Use o modificador de acesso `private` para restringir o acesso direto aos atributos.

7.6 Modificadores de Acesso

Modificador	Visibilidade
`public`	Acessível em qualquer parte do código
`private`	Acessível apenas na própria classe
`protected`	Acessível no mesmo pacote e subclasses

Modificador	Visibilidade
(sem)	Acessível apenas no mesmo pacote

7.7 Referência a Objetos

Em Java, variáveis de objetos são referências. Atribuir uma variável de objeto a outra faz com que ambas apontem para o mesmo objeto.

📌 **Exemplo:**

```
Pessoa pessoa3 = new Pessoa("Lucas", 22);
Pessoa pessoa4 = pessoa3; // Ambos apontam para o mesmo objeto
```

7.8 Boas Práticas

✅ Nomeie as classes com a primeira letra maiúscula e substantivos.
✅ Use nomes descritivos para atributos e métodos.
✅ Aplique encapsulamento para proteger os dados.
✅ Inicialize os objetos usando construtores.

7.9 Revisão do Capítulo

Neste capítulo, você aprendeu:
✅ O conceito de classes como moldes para objetos.
✅ Como criar objetos e inicializá-los usando construtores.
✅ A importância do encapsulamento e como aplicar getters e setters.
✅ O papel dos modificadores de acesso para proteger dados.

💡 **Próximo Passo:** No próximo capítulo, exploraremos **Herança e Polimorfismo**, conceitos essenciais para o reuso de código e a flexibilidade dos programas em Java.

📖 Capítulo 8 – Herança e Polimorfismo

8.1 Introdução

Herança e polimorfismo são pilares fundamentais da Programação Orientada a Objetos (POO). Esses conceitos permitem reutilizar código e criar programas mais flexíveis e fáceis de manter.

💡 **Dica**: Herança e polimorfismo trabalham juntos para criar uma estrutura de classes hierárquica e reutilizável.

8.2 Herança

Herança é o mecanismo que permite que uma classe **filha** (subclasse) herde atributos e métodos de uma classe **pai** (superclasse).

📌 **Sintaxe básica:**

```
class Superclasse {
    // atributos e métodos
}

class Subclasse extends Superclasse {
    // atributos e métodos adicionais
}
```

💡 **Dica**: Use herança para evitar repetição de código em várias classes que compartilham características comuns.

8.3 Exemplo Prático de Herança

📌 **Exemplo:**

```
// Superclasse
class Animal {
    void comer() {
        System.out.println("Animal comendo...");
    }
}

// Subclasse
class Cachorro extends Animal {
    void latir() {
        System.out.println("Cachorro latindo...");
    }
}

// Uso:
Cachorro cachorro = new Cachorro();
cachorro.comer();  // Método herdado
cachorro.latir();  // Método da subclasse
```

💡 **Dica**: A subclasse tem acesso a todos os métodos e atributos públicos ou protegidos da superclasse.

8.4 Polimorfismo

Polimorfismo permite que objetos de diferentes classes sejam tratados como objetos de uma mesma superclasse. Isso promove **flexibilidade** no código.

📌 **Exemplo:**

```
Animal meuAnimal = new Cachorro(); // Cachorro é um Animal
```

```
meuAnimal.comer(); // Executa método da superclasse ou da subclasse, dependendo do contexto
```

💡 **Dica**: O polimorfismo permite usar referências de tipos genéricos para instanciar objetos específicos.

8.5 Sobrescrita de Métodos (Override)

Uma subclasse pode sobrescrever métodos da superclasse para fornecer uma implementação específica.

📌 **Exemplo:**

```
class Animal {
    void emitirSom() {
        System.out.println("Som genérico de animal");
    }
}

class Gato extends Animal {
    @Override
    void emitirSom() {
        System.out.println("Miau");
    }
}

// Uso:
Animal animal = new Gato();
animal.emitirSom(); // Saída: Miau
```

💡 **Dica**: Use a anotação `@Override` para indicar que o método está sendo sobrescrito.

8.6 Herança Múltipla

Java **não permite herança múltipla direta**, ou seja, uma classe não pode estender mais de uma classe. Para resolver isso, utilizamos **interfaces** (veremos no próximo capítulo).

8.7 Boas Práticas

✅ Utilize herança apenas quando houver uma relação "é um" clara entre as classes.
✅ Evite heranças profundas (cadeias longas) para não dificultar a manutenção.
✅ Prefira a composição sobre a herança, quando apropriado.

8.8 Revisão do Capítulo

Neste capítulo, você aprendeu:
✅ Como usar herança para criar subclasses a partir de uma superclasse.
✅ A importância do polimorfismo para aumentar a flexibilidade do código.

✅ Como sobrescrever métodos para fornecer comportamentos específicos.
✅ Que o Java não suporta herança múltipla direta.

💡 **Próximo Passo:** No próximo capítulo, exploraremos **Interfaces e Classes Abstratas**, fundamentais para organizar e projetar sistemas complexos.

📖 Capítulo 9 – Interfaces e Classes Abstratas

9.1 Introdução

Interfaces e classes abstratas são ferramentas poderosas em Java para criar hierarquias flexíveis e reutilizáveis. Elas permitem definir contratos de implementação e organizar melhor o código, especialmente em projetos maiores.

💡 **Dica**: Use interfaces para definir comportamentos comuns e classes abstratas para fornecer implementações parciais.

9.2 Interfaces

Uma interface define um **contrato** que uma classe deve cumprir, especificando métodos que devem ser implementados.

📌 **Sintaxe básica:**

```
interface NomeDaInterface {
    void metodo();
}
```

📌 **Exemplo:**

```
interface Animal {
    void emitirSom();
}

class Cachorro implements Animal {
    public void emitirSom() {
        System.out.println("Au Au");
    }
}
```

💡 **Dica**: Interfaces permitem simular herança múltipla em Java, pois uma classe pode implementar várias interfaces.

9.3 Implementando Múltiplas Interfaces

Uma classe pode implementar várias interfaces, o que ajuda a compor comportamentos distintos.

📌 **Exemplo:**

```
interface Voador {
    void voar();
}

interface Nadador {
    void nadar();
}

class Pato implements Voador, Nadador {
    public void voar() {
        System.out.println("Pato voando...");
    }

    public void nadar() {
        System.out.println("Pato nadando...");
    }
}
```

9.4 Classes Abstratas

Uma classe abstrata é uma classe que não pode ser instanciada diretamente, servindo como base para outras classes. Ela pode conter métodos abstratos (sem corpo) e métodos concretos (com implementação).

📌 **Sintaxe básica:**

```
abstract class NomeDaClasse {
    abstract void metodoAbstrato();
    void metodoConcreto() {
        System.out.println("Método concreto.");
    }
}
```

📌 **Exemplo:**

```
abstract class Animal {
    abstract void emitirSom();
    void dormir() {
        System.out.println("Animal dormindo...");
    }
}

class Gato extends Animal {
    void emitirSom() {
        System.out.println("Miau");
    }
}
```

9.5 Diferenças entre Interfaces e Classes Abstratas

Característica	Interface	Classe Abstrata
Métodos com implementação	Não (Java 8+ permite `default`)	Sim
Construtores	Não	Sim
Herança múltipla	Sim (pode implementar várias interfaces)	Não
Uso principal	Definir contratos	Fornecer base comum + implementação

💡 **Dica**: Use interfaces para definir **comportamentos** e classes abstratas para fornecer **implementações parciais**.

9.6 Boas Práticas

✅ Use interfaces para definir papéis comuns entre classes não relacionadas.
✅ Prefira classes abstratas quando existir uma hierarquia clara.
✅ Implemente apenas os métodos necessários para cada classe.
✅ Evite criar interfaces muito genéricas que não façam sentido no domínio do projeto.

9.7 Revisão do Capítulo

Neste capítulo, você aprendeu:
✅ O que são interfaces e como usá-las para definir contratos.
✅ Como implementar múltiplas interfaces em uma classe.
✅ O conceito de classes abstratas e quando usá-las.
✅ As principais diferenças entre interfaces e classes abstratas.

💡 **Próximo Passo:** No próximo capítulo, veremos **Tratamento de Exceções**, essencial para construir programas robustos e confiáveis.

📖 Capítulo 10 – Tratamento de Exceções

10.1 Introdução

Erros e falhas podem ocorrer durante a execução de programas Java — como problemas de leitura de arquivos ou divisões por zero. O tratamento de exceções é um mecanismo que permite lidar com essas situações de forma controlada, garantindo a estabilidade do software.

💡 **Dica**: Exceções bem tratadas tornam o código mais seguro e confiável.

10.2 O que são Exceções?

Uma exceção é um evento que interrompe o fluxo normal de execução do programa. Em Java, exceções são representadas por objetos derivados da classe `Throwable`, que se divide em duas categorias principais:

- **Checked Exceptions**: devem ser tratadas explicitamente (e.g. IOException).
- **Unchecked Exceptions**: geralmente decorrem de erros de programação (e.g. NullPointerException).

10.3 Blocos try-catch-finally

O tratamento de exceções é feito com os blocos `try`, `catch` e `finally`.

📌 **Exemplo básico:**

```
try {
    int resultado = 10 / 0;
    System.out.println("Resultado: " + resultado);
} catch (ArithmeticException e) {
    System.out.println("Erro: Divisão por zero!");
} finally {
    System.out.println("Bloco finally executado.");
}
```

💡 **Dica**: O bloco `finally` é sempre executado, independentemente de uma exceção ter sido lançada ou não. Ideal para liberar recursos como arquivos ou conexões.

10.4 Hierarquia de Exceções

A hierarquia de exceções começa com `Throwable`:

- **Error**: indica erros graves que não devem ser tratados pelo código do programa (ex.: `OutOfMemoryError`).
- **Exception**: pode ser tratada pelo programa.
    - **RuntimeException**: exceções de tempo de execução, como `NullPointerException`.
    - **Checked Exceptions**: exceções obrigatórias de serem tratadas.

10.5 Criando Exceções Personalizadas

É possível criar exceções próprias estendendo a classe `Exception` ou `RuntimeException`.

📌 **Exemplo:**

```
class MinhaExcecao extends Exception {
```

```
    public MinhaExcecao(String mensagem) {
        super(mensagem);
    }
}

class Teste {
    void verificarIdade(int idade) throws MinhaExcecao {
        if (idade < 18) {
            throw new MinhaExcecao("Idade insuficiente!");
        }
    }
}
```

10.6 Propagando Exceções

Quando um método não trata a exceção internamente, ele pode lançá-la para o chamador utilizando `throws`.

📌 **Exemplo:**

```
void metodo() throws IOException {
    // código que pode lançar IOException
}
```

10.7 Multi-Catch

Desde o Java 7, é possível capturar múltiplas exceções no mesmo bloco `catch`.

📌 **Exemplo:**

```
try {
    // código
} catch (IOException | SQLException e) {
    System.out.println("Exceção capturada: " + e.getMessage());
}
```

10.8 Boas Práticas

✅ Sempre trate exceções específicas antes de exceções genéricas.
✅ Evite capturar exceções genéricas como `Exception` a menos que seja realmente necessário.
✅ Libere recursos no bloco `finally` ou use **try-with-resources** para classes que implementam a interface `AutoCloseable`.
✅ Nunca ignore exceções sem ao menos logar o erro.

10.9 Revisão do Capítulo

Neste capítulo, você aprendeu:
✅ O que são exceções e por que tratá-las é importante.
✅ Como usar os blocos `try`, `catch` e `finally` para capturar e lidar com erros.
✅ A diferença entre checked e unchecked exceptions.
✅ Como criar exceções personalizadas.
✅ Como propagar exceções com `throws`.

💡 **Próximo Passo:** No próximo capítulo, exploraremos **Coleções em Java**, fundamentais para organizar e manipular dados de forma eficiente.

📖 Capítulo 11 – Coleções em Java

11.1 Introdução

Coleções são estruturas de dados que armazenam grupos de objetos. Elas facilitam a manipulação de dados como listas, filas e conjuntos de forma eficiente e flexível.

💡 **Dica**: Utilize coleções para lidar com grandes volumes de dados de forma organizada e segura.

11.2 O Framework Collections

O Java Collections Framework (JCF) é um conjunto de interfaces e classes que provê implementações de estruturas de dados. Ele faz parte do pacote `java.util`.

📌 **Principais interfaces:**

- `Collection`: Interface raiz para todas as coleções.
- `List`: Permite elementos duplicados e mantém a ordem.
- `Set`: Não permite elementos duplicados.
- `Map`: Armazena pares chave-valor.

11.3 Listas (List)

A interface `List` é usada quando você precisa de uma coleção ordenada que permite elementos duplicados.

📌 **Principais implementações:**

- `ArrayList`: Rápido para buscas, tamanho dinâmico.
- `LinkedList`: Rápido para inserções e remoções.

📌 **Exemplo:**

```
List<String> nomes = new ArrayList<>();
nomes.add("Ana");
nomes.add("Bruno");
nomes.add("Carlos");
System.out.println(nomes);
```

💡 **Dica**: Prefira `ArrayList` para leituras frequentes e `LinkedList` para inserções/remissões frequentes.

11.4 Conjuntos (Set)

A interface `Set` não permite elementos duplicados.

📌 **Principais implementações:**

- `HashSet`: Não garante ordem dos elementos.
- `LinkedHashSet`: Mantém a ordem de inserção.
- `TreeSet`: Ordena elementos de acordo com a ordem natural ou comparador.

📌 **Exemplo:**

```
Set<String> cores = new HashSet<>();
cores.add("Azul");
cores.add("Verde");
cores.add("Azul"); // Ignorado, pois "Azul" já existe.
System.out.println(cores);
```

11.5 Mapas (Map)

`Map` armazena pares chave-valor.

📌 **Principais implementações:**

- `HashMap`: Não garante ordem.
- `LinkedHashMap`: Mantém ordem de inserção.
- `TreeMap`: Ordena as chaves.

📌 **Exemplo:**

```
Map<Integer, String> idNomes = new HashMap<>();
idNomes.put(1, "Ana");
idNomes.put(2, "Bruno");
System.out.println(idNomes);
```

11.6 Iterando sobre Coleções

Use `for-each`, `Iterator` ou expressões lambda (Java 8+) para percorrer coleções.

📌 **Exemplo com for-each:**

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

💡 **Dica**: Para evitar `ConcurrentModificationException`, use o `Iterator` ao remover elementos durante a iteração.

11.7 Generics

Coleções utilizam generics para garantir **segurança de tipos** e evitar erros de casting.

📌 **Exemplo:**

```
List<String> lista = new ArrayList<>();
lista.add("Java");
// lista.add(123); // Erro em tempo de compilação
```

11.8 Boas Práticas

✅ Utilize a interface (`List`, `Set`, `Map`) como tipo de referência, não a implementação diretamente.
✅ Evite misturar tipos de dados em coleções (use generics!).
✅ Prefira coleções imutáveis quando possível (Java 9+).

11.9 Revisão do Capítulo

Neste capítulo, você aprendeu:
✅ O que são coleções e sua importância.
✅ As principais interfaces e implementações: `List`, `Set` e `Map`.
✅ Como iterar sobre coleções de forma segura e eficiente.
✅ Como usar generics para segurança de tipos.

💡 **Próximo Passo:** No próximo capítulo, exploraremos **Streams e Lambda Expressions**, que facilitam a manipulação de dados de forma funcional.

📖 Capítulo 12 – Streams e Lambda Expressions

12.1 Introdução

A partir do Java 8, o conceito de **Streams** e **Lambda Expressions** foi introduzido para facilitar o processamento de dados de forma mais funcional e concisa, melhorando a produtividade do desenvolvedor e a legibilidade do código.

💡 **Dica**: Streams permitem operações em **pipeline**, transformando coleções de dados em resultados de forma expressiva.

12.2 O que são Streams?

Streams são fluxos de dados que permitem aplicar operações como filtragem, ordenação e transformação, evitando loops explícitos e promovendo código mais declarativo.

📌 **Exemplo básico:**

```
List<String> nomes = Arrays.asList("Ana", "Bruno", "Carlos");
nomes.stream()
     .filter(nome -> nome.startsWith("A"))
     .forEach(System.out::println);
```

12.3 Criação de Streams

Você pode criar streams de várias formas:

- A partir de coleções:

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

- A partir de arrays:

```
int[] valores = {1, 2, 3};
IntStream intStream = Arrays.stream(valores);
```

- Usando `Stream.of()`:

```
Stream<String> stream = Stream.of("A", "B", "C");
```

12.4 Operações Intermediárias

São operações que transformam ou filtram dados de um stream e retornam outro stream. Exemplos:

- `filter`: Filtra elementos com base em uma condição.

- `map`: Transforma elementos de um tipo em outro.
- `sorted`: Ordena elementos.

📌 **Exemplo:**

```
List<String> palavras = Arrays.asList("java", "stream", "lambda");
palavras.stream()
        .filter(p -> p.length() > 4)
        .map(String::toUpperCase)
        .sorted()
        .forEach(System.out::println);
```

12.5 Operações Finais

São operações que consomem o stream e produzem um resultado ou efeito colateral. Exemplos:

- `forEach`: Executa uma ação para cada elemento.
- `collect`: Converte o stream em outra coleção ou resultado.
- `count`: Conta o número de elementos.

📌 **Exemplo:**

```
List<String> lista = Arrays.asList("A", "B", "C");
long quantidade = lista.stream().count();
System.out.println("Quantidade: " + quantidade);
```

12.6 Lambda Expressions

As lambdas permitem escrever implementações de interfaces funcionais de forma concisa.

📌 **Sintaxe:**

```
(parametros) -> expressão
```

📌 **Exemplo:**

```
List<String> nomes = Arrays.asList("Ana", "Bruno");
nomes.forEach(nome -> System.out.println(nome));
```

12.7 Method References

Uma forma abreviada de lambda expressions, quando a implementação é apenas chamar um método.

📌 **Exemplo:**

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

12.8 Boas Práticas

✅ Use streams para processamento funcional de dados, mas evite streams em situações em que loops tradicionais são mais simples.
✅ Prefira lambdas em vez de classes anônimas para código mais legível.
✅ Evite usar streams quando o processamento é complexo e precisa de muitas variáveis de estado — isso pode reduzir a legibilidade.

12.9 Revisão do Capítulo

Neste capítulo, você aprendeu:
✅ O conceito de Streams e sua aplicação em coleções.
✅ Como usar operações intermediárias e finais para processar dados.
✅ Como utilizar lambda expressions e method references para simplificar o código.

💡 **Próximo Passo:** No próximo capítulo, vamos explorar **Programação Concorrente**, essencial para aplicações de alto desempenho.

📖 Capítulo 13 – Programação Concorrente

13.1 Introdução

No mundo moderno de aplicações Java, é cada vez mais comum lidar com tarefas que precisam ser executadas em paralelo, seja para melhorar a performance ou para oferecer melhor responsividade aos usuários. A **programação concorrente** é o ramo da computação que estuda como executar múltiplas tarefas ao mesmo tempo.

💡 **Dica**: Concorra com responsabilidade — o uso incorreto de threads pode causar problemas como deadlocks e condições de corrida.

13.2 Threads em Java

Uma **thread** é uma linha de execução independente dentro de um processo. Em Java, podemos criar threads de duas formas principais:

- Implementando a interface `Runnable`:

```
class MinhaTarefa implements Runnable {
    @Override
    public void run() {
        System.out.println("Executando em outra thread");
    }
}
```

- Estendendo a classe `Thread`:

```
class MinhaThread extends Thread {
    @Override
    public void run() {
        System.out.println("Executando em outra thread");
    }
}
```

13.3 Criando e Iniciando Threads

Para executar uma tarefa em paralelo:

```
Thread thread = new Thread(new MinhaTarefa());
thread.start();
```

💡 **Dica**: Sempre use `start()` para iniciar uma nova thread. Não chame `run()` diretamente, pois isso executará a tarefa na thread atual.

13.4 Concorrência com ExecutorService

A partir do Java 5, a classe `ExecutorService` simplificou a execução de múltiplas tarefas.

📌 **Exemplo com ExecutorService:**

```
ExecutorService executor = Executors.newFixedThreadPool(3);
executor.submit(() -> System.out.println("Executando tarefa"));
executor.shutdown();
```

💡 **Dica**: Use `shutdown()` para encerrar o executor de forma ordenada.

13.5 Sincronização

Quando múltiplas threads acessam recursos compartilhados, pode ocorrer **condição de corrida**. Para evitar inconsistências, usamos a palavra-chave `synchronized`.

📌 **Exemplo:**

```
public synchronized void incrementar() {
    contador++;
}
```

💡 **Dica**: Use sincronização apenas onde necessário para evitar gargalos.

13.6 Problemas comuns

- **Deadlock**: quando duas threads esperam por recursos bloqueados umas pelas outras.
- **Starvation**: quando uma thread nunca é executada por falta de acesso ao recurso.

- **Livelock**: quando threads continuam ativas mas sem progresso útil.

13.7 Recursos avançados

- **Locks explícitos** (ReentrantLock)
- **Semáforos**
- **CountDownLatch**
- **CyclicBarrier**
- **Concurrent Collections** (como `ConcurrentHashMap`)

💡 **Dica**: Use as coleções concorrentes para evitar bloqueios desnecessários e melhorar a performance.

13.8 Boas Práticas

✅ Evite criar muitas threads manualmente; prefira `ExecutorService`.
✅ Utilize sincronização apenas quando necessário.
✅ Use bibliotecas de concorrência modernas (`java.util.concurrent`).
✅ Teste com cuidado para evitar problemas como deadlocks e race conditions.

13.9 Revisão do Capítulo

Neste capítulo, você aprendeu:
✅ O que são threads e como usá-las para executar tarefas em paralelo.
✅ Como usar `ExecutorService` para gerenciar múltiplas tarefas de forma eficiente.
✅ A importância da sincronização e como evitar problemas comuns como deadlocks.
✅ Recursos avançados da API de concorrência para aplicações profissionais.

💡 **Próximo Passo:** No próximo capítulo, vamos explorar **Boas Práticas de Programação**, que vão te ajudar a escrever código mais limpo, legível e eficiente.

📖 Capítulo 14 – Boas Práticas de Programação

14.1 Introdução

Escrever código vai muito além de fazê-lo funcionar — ele deve ser claro, legível, fácil de manter e seguir boas práticas. Neste capítulo, abordaremos técnicas para tornar seu código Java mais profissional.

💡 **Dica**: O código que você escreve hoje pode ser lido por você ou por outros desenvolvedores no futuro. Facilite a leitura!

14.2 Convenções de Codificação

O Java tem um conjunto de convenções de codificação que devem ser seguidas para manter a padronização e a legibilidade:

- **Nomes de classes**: Inicie com letra maiúscula e use CamelCase.
  📌 Ex.: `MinhaClasse`
- **Nomes de métodos e variáveis**: Inicie com letra minúscula e use CamelCase.
  📌 Ex.: `calcularSalario`
- **Constantes**: Use letras maiúsculas e underscores.
  📌 Ex.: `TAXA_IMPOSTO`

14.3 Documentação

Use o **JavaDoc** para documentar suas classes, métodos e atributos. Isso facilita a manutenção e o entendimento do código.

📌 **Exemplo de JavaDoc:**

```
/**
 * Calcula o salário líquido de um funcionário.
 * @param salarioBruto Salário bruto.
 * @param desconto Desconto a ser aplicado.
 * @return Salário líquido.
 */
public double calcularSalarioLiquido(double salarioBruto, double desconto) {
    return salarioBruto - desconto;
}
```

14.4 Estruturação de Código

✅ Separe responsabilidades em classes diferentes seguindo o **princípio da responsabilidade única** (SRP).
✅ Mantenha métodos curtos e com uma única responsabilidade.
✅ Evite blocos de código duplicados.

💡 **Dica**: Se um método crescer demais, divida-o em métodos auxiliares.

14.5 Tratamento de Exceções

✅ Use exceções para lidar com condições anormais.
✅ Crie exceções personalizadas se necessário.
✅ Evite capturar exceções genéricas (`Exception`) sem tratá-las adequadamente.

📌 **Exemplo:**

```
try {
    // código que pode lançar exceção
} catch (IOException e) {
    // tratamento específico
} catch (Exception e) {
    // tratamento genérico (use com cuidado)
}
```

14.6 Uso de Collections

✅ Prefira usar interfaces (`List`, `Set`, `Map`) ao invés de implementações específicas para maior flexibilidade.
✅ Use **Generics** para evitar problemas de tipos em tempo de execução.

14.7 Uso de Streams

✅ Use **Streams** para processar dados de forma funcional e legível.
✅ Evite usar streams em cenários onde loops simples são mais claros.
✅ Combine lambdas e method references para código conciso.

14.8 Programação Concorrente

✅ Use `ExecutorService` ao invés de criar threads manualmente.
✅ Use coleções seguras para threads (`ConcurrentHashMap`, `CopyOnWriteArrayList`) quando necessário.
✅ Evite bloqueios desnecessários e teste cuidadosamente para evitar deadlocks.

14.9 Boas Práticas Gerais

✅ Comente apenas quando necessário; o código deve ser autoexplicativo.
✅ Nomeie variáveis e métodos de forma descritiva.
✅ Escreva testes automatizados para garantir a qualidade do código.
✅ Utilize controle de versão (como Git) para gerenciar alterações.

14.10 Revisão do Capítulo

Neste capítulo, você aprendeu:
✅ As convenções de codificação que tornam o código mais legível e padronizado.
✅ Como documentar seu código com JavaDoc.
✅ A importância de estruturar o código seguindo boas práticas de design.
✅ Como usar exceções, collections, streams e programação concorrente de forma profissional.

📖 Conclusão

Chegamos ao fim desta jornada pelo universo do Java Básico! Ao longo dos capítulos, percorremos uma trilha de aprendizado sólida, que preparou você para avançar ainda mais no desenvolvimento profissional com esta poderosa linguagem.

No **Capítulo 1**, você aprendeu a configurar o ambiente de desenvolvimento Java, garantindo as bases para executar e compilar seus programas com confiança.

No **Capítulo 2**, exploramos a estrutura básica de um programa Java, entendendo desde a função `main` até as convenções de nomenclatura.

No **Capítulo 3**, vimos como declarar e inicializar variáveis e constantes, fundamentais para o armazenamento e manipulação de dados.

No **Capítulo 4**, nos aprofundamos nos tipos de dados primitivos, essenciais para representar números, caracteres e valores booleanos.

No **Capítulo 5**, aprendemos sobre operadores aritméticos, relacionais e lógicos, que nos permitem realizar cálculos e tomar decisões no código.

No **Capítulo 6**, estudamos estruturas de controle como `if`, `else`, `switch`, `while`, `for` e `do-while`, ferramentas cruciais para controlar o fluxo de execução de programas.

No **Capítulo 7**, abordamos métodos (funções), aprendendo a declarar, invocar e passar parâmetros para modularizar o código.

No **Capítulo 8**, falamos sobre arrays e como manipular coleções de dados de forma eficiente.

No **Capítulo 9**, introduzimos o conceito de classes e objetos, pilares da Programação Orientada a Objetos (POO) em Java.

No **Capítulo 10**, exploramos o conceito de construtores e como eles inicializam objetos de maneira organizada.

No **Capítulo 11**, aprendemos sobre encapsulamento, visibilidade e como proteger os dados internos das classes.

No **Capítulo 12**, abordamos as expressões lambda e a API de Streams, recursos modernos que tornam o código mais funcional e legível.

No **Capítulo 13**, exploramos a programação concorrente, mostrando como executar tarefas simultâneas com threads e `ExecutorService`.

No **Capítulo 14**, revisamos as boas práticas de programação, essenciais para escrever código limpo, legível e manutenível.

Assim, reunimos as principais habilidades para que você seja capaz de projetar, desenvolver e manter aplicações Java com qualidade e confiança.

Obrigada por chegar até aqui, desenvolvedores! Que esta obra seja uma fonte de inspiração para sua jornada no desenvolvimento de software. Lembre-se: a prática constante é o caminho para o domínio — e este e-book é apenas o começo. Continue estudando, explorando e criando soluções incríveis com Java! 🚀

📚 Bibliografia

ORACLE. **Documentação Oficial Java.** Disponível em: https://dev.java/learn/language-basics/. Acesso em: 04 jun. 2025.

ORACLE. **The Java™ Tutorials.** Disponível em: https://docs.oracle.com/javase/tutorial/. Acesso em: 04 jun. 2025.

ECKEL, Bruce. *Thinking in Java*. 4. ed. Upper Saddle River: Prentice Hall, 2006.

HORSTMANN, Cay S.; CORNELL, Gary. *Core Java™ Volume I–Fundamentals*. 11. ed. Upper Saddle River: Prentice Hall, 2018.

BLOCH, Joshua. *Effective Java*. 3. ed. Boston: Addison-Wesley, 2018.

GOSLING, James et al. *The Java Language Specification*. 4. ed. Boston: Addison-Wesley, 2014.

DEITEL, Paul; DEITEL, Harvey. *Java: How to Program*. 10. ed. Upper Saddle River: Prentice Hall, 2015.
