Leonardo R. Orabona

_Algoritmos e Estrutura de Dados

UNIDADE I

Sumário

1

O que é Java?

Write Once, Run Anywhere

Java é uma linguagem de programação e plataforma de computação liberada pela primeira vez pela Sun Microsystems, em 1995, e, mais tarde, foi adquirida pela Oracle Corporation.

De um início humilde, ela evoluiu para uma grande participação no mundo digital dos dias atuais, oferecendo a plataforma confiável na qual muitos serviços e aplicativos são desenvolvidos. Produtos e serviços novos e inovadores projetados para o futuro continuam a confiar no Java também.

O Java é uma das linguagens de programação mais populares e amplamente utilizadas no mundo da tecnologia. Desde sua criação, foi desenvolvido com características fundamentais que o tornam uma escolha robusta para aplicações diversas, desde desenvolvimento *web* e *mobile* até sistemas embarcados e aplicações empresariais. Três dos principais pilares que destacam o Java entre outras linguagens são sua portabilidade, segurança e simplicidade.

- **Portabilidade:** o Java foi projetado para ser independente de plataforma, o que significa que os programas Java podem ser executados em qualquer dispositivo ou sistema compatível com a máquina virtual Java (JVM).
- **Segurança:** o Java introduziu recursos de segurança, como a execução de código em ambientes sandbox e verificação de tipos,

para evitar vulnerabilidades comuns de segurança.

- **Simplicidade:** Java procurou oferecer uma sintaxe simples e fácil de entender, tornando-o acessível a programadores de diferentes níveis de habilidade.

A combinação desses três fatores faz do Java uma linguagem extremamente versátil e confiável para o desenvolvimento de *software*. Sua portabilidade garante que programas possam ser executados em diferentes sistemas sem modificações significativas. A segurança incorporada reduz riscos e torna o Java uma escolha ideal para aplicações críticas. Além disso, sua simplicidade facilita o aprendizado e a adoção por programadores iniciantes e experientes. Com essas características, o Java continua sendo uma das linguagens mais relevantes no cenário da programação moderna.

O Java teve papel significativo no desenvolvimento de aplicativos para celulares mais antigos, especialmente antes da ascensão do sistema operacional Android. Antes dos *smartphones* modernos, os chamados *feature phones* (celulares com recursos básicos) eram

populares e muitos deles suportavam aplicativos Java ME (Java Platform, Micro Edition).

Java ME (Micro Edition): o Java ME foi uma plataforma Java projetada para dispositivos com recursos limitados, como celulares mais antigos. Ele oferecia uma máquina virtual Java mais leve e um subconjunto de APIs Java padrão para desenvolver aplicativos móveis.

O Java ME (Micro Edition) foi fundamental para o desenvolvimento de aplicativos móveis em uma era em que os dispositivos não tinham a capacidade de processamento, memória e recursos gráficos dos *smartphones* modernos. Sua criação visava atender à necessidade de uma plataforma eficiente para rodar em dispositivos com *hardware* limitado, como celulares de baixo custo e com telas pequenas.

- **Características do Java ME**:
  - **Máquina Virtual Java (JVM) leve**:

A JVM do Java ME foi adaptada para ser mais compacta, permitindo que o código Java fosse executado em dispositivos que não tinham grande capacidade de processamento. Isso permitia que os aplicativos fossem executados de forma eficaz, mesmo em celulares com especificações mais simples.

- **APIs específicas para dispositivos limitados**:

O Java ME ofereceu um subconjunto de APIs otimizadas para dispositivos móveis, como bibliotecas para manipulação de gráficos,

interface de usuário simples, redes móveis e armazenamento de dados, tudo em um formato reduzido para não sobrecarregar o dispositivo.

- **Compatibilidade multiplataforma**:

Assim como o conceito de "Write Once, Run Anywhere" no Java, o Java ME permitia que os aplicativos desenvolvidos para uma plataforma específica (como celulares de uma marca) fossem executados em diferentes dispositivos compatíveis, facilitando a distribuição e manutenção de aplicativos móveis.

- **Aplicações diversificadas**:

A plataforma Java ME era usada para jogos, aplicativos de mensagens, navegadores de internet, calculadoras e outros aplicativos simples, que eram fundamentais para os celulares da época. Aplicativos como J2ME (Java 2 Platform, Micro Edition) permitiam que esses dispositivos realizassem tarefas mais avançadas do que apenas fazer chamadas e enviar mensagens de texto.

O Impacto na era *pré-smartphone*

Durante o auge dos *feature phones*, o Java ME dominava o mercado de aplicativos móveis. Fabricantes como Nokia, Samsung e Sony Ericsson tinham muitos de seus dispositivos rodando aplicativos Java, e muitos jogos icônicos daquela época, como Snake e Tetris, eram desenvolvidos para rodar em Java ME.

O Java ME representava uma ponte para a mobilidade e tecnologia em um momento em que os celulares estavam se tornando mais do que simples dispositivos de comunicação. Ele permitiu que os usuários tivessem uma experiência interativa e conectada de maneira inovadora.

Declínio e transição para o Android

Com o advento dos *smartphones* modernos, o Java ME foi gradualmente substituído por sistemas mais sofisticados, como o Android, que utiliza Java de maneira mais avançada e com mais funcionalidades. O Android trouxe uma plataforma mais robusta e integrada, baseada no Java SE (Standard Edition), com uma máquina

virtual própria chamada Dalvik (posteriormente substituída pela ART - Android Runtime).

Mesmo com a ascensão do Android, o legado do Java ME ainda é significativo, pois foi fundamental para criar a base do mercado de aplicativos móveis e influenciou o desenvolvimento de tecnologias móveis mais avançadas.

O Android, um dos sistemas operacionais móveis mais populares do mundo, foi inicialmente desenvolvido com Java como a linguagem de programação principal.

Foi fundado em 2003 por Andy Rubin, Rich Miner, Nick Sears e Chris White. Em 2005, a empresa Android Inc. foi adquirida pela Google, e esse foi um passo crucial no desenvolvimento do sistema operacional Android.

Desde o início, o Android adotou o Java como a linguagem de programação principal para o desenvolvimento de aplicativos. Isso se deveu em parte à familiaridade dos desenvolvedores com Java e à portabilidade que a linguagem oferecia.

O uso do Java como linguagem principal no desenvolvimento do Android desempenhou papel fundamental na popularização do sistema operacional. A familiaridade dos desenvolvedores com Java e sua capacidade de ser executado em diferentes dispositivos garantiram que a criação de aplicativos para Android fosse acessível e eficiente.

A aquisição da Android Inc. pela Google, em 2005, foi um marco que impulsionou o crescimento do sistema, transformando-o no líder do mercado de dispositivos móveis. Com uma base sólida em Java, o Android evoluiu

continuamente, incorporando novas tecnologias e expandindo seu ecossistema de aplicativos.

Mesmo com o surgimento de outras linguagens compatíveis, como Kotlin, o Java continua sendo um pilar essencial para o desenvolvimento de aplicativos Android, consolidando seu legado na história da computação móvel.

Preparando o ambiente

Antes de começar a programar em Java, é essencial configurar corretamente o ambiente de desenvolvimento. Isso inclui a instalação das ferramentas necessárias para escrever, compilar e executar código Java de forma eficiente.

Instalando o Java Development Kit (JDK)

O **JDK (Java Development Kit)** é um conjunto de ferramentas fundamentais para o desenvolvimento em Java. Ele inclui:

- **Compilador (javac):** transforma o código-fonte Java em bytecode.
- **Máquina Virtual Java (JVM):** executa os programas Java.
- **Bibliotecas e ferramentas:** contêm classes e métodos essenciais para a programação.

Passos para instalar o JDK:

- **Baixe** a versão mais recente do **JDK** no *site* oficial da Oracle ou utilize uma distribuição **OpenJDK**.
- **Instale** o JDK no sistema operacional.
- **Configure a variável de ambiente JAVA_HOME** (opcional, mas útil para algumas ferramentas).

Escolhendo uma IDE para programação em Java

Uma **IDE (*Integrated Development Environment*)** facilita a escrita, organização e depuração do código Java.

Algumas das IDEs mais utilizadas são:

- **Eclipse:** popular e amplamente usado para desenvolvimento Java.
- **IntelliJ IDEA:** muito otimizado e recomendado para produtividade.
- **NetBeans:** simples e eficiente para quem busca uma alternativa *open-source*.

Após a instalação, a IDE precisa ser configurada para reconhecer o JDK e permitir a criação de projetos Java.

Escrevendo e executando seu primeiro programa Java

Com o ambiente pronto, já é possível escrever o primeiro programa Java.

Crie um arquivo Java chamado HelloWorld.java.

Escreva o código abaixo:

```
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println(“Olá, mundo!”);

    }

}
```

Compile o programa no terminal usando:

```
javac HelloWorld.java
```

Execute o programa

java HelloWorld

Se tudo estiver configurado corretamente, a saída será:

Olá, mundo!

A preparação do ambiente é um passo importante para quem deseja programar em Java. Com o JDK instalado e uma IDE configurada, os desenvolvedores podem começar a escrever, compilar e executar seus programas sem dificuldades. Essa configuração inicial permite explorar os conceitos fundamentais da linguagem e avançar para projetos mais complexos no futuro.

3

Variáveis primitivas

Variáveis primitivas em Java

As variáveis primitivas são os tipos de dados mais básicos em Java. Elas armazenam diretamente os valores na memória e são altamente eficientes em termos de desempenho. Diferente dos objetos, as variáveis primitivas não são referências a instâncias de classes – elas contêm o dado em si.

Principais tipos primitivos em Java

O Java possui oito tipos primitivos, divididos em números inteiros, números de ponto flutuante, caracteres e valores booleanos.

Tipo	Tamanho (bits)	Valores possíveis	Exemplo
byte	8 bits	-128 a 127	`byte idade = 25;`
short	16 bits	-32.768 a 32.767	`short ano = 2025;`
int	32 bits	-2^{31} a $2^{31}-1$	`int população = 1000000;`
long	64 bits	-2^{63} a $2^{63}-1$	`long distância = 123456789L;`
float	32 bits	±3.4 x 10^{38} (~7 dígitos de precisão)	`float temperatura = 36.5f;`
double	64 bits	±1.8 x 10^{308} (~15 dígitos de precisão)	`double preço = 99.99;`
char	16 bits	Caracteres Unicode (0 a 65.535)	`char letra = 'A';`
boolean	1 bit	`true` ou `false`	`boolean ativo = true;`

Fonte: elaborado pelo autor.

Exemplo de uso das variáveis primitivas

```
public class VariaveisPrimitivas {
    public static void main(String[] args) {
        int idade = 25;
        double salario = 4500.75;
        char inicial = ‘J’;
        boolean ativo = true;
        System.out.println(“Idade: “ + idade);
        System.out.println(“Salário: “ + salario);
        System.out.println(“Inicial: “ + inicial);
        System.out.println(“Ativo: “ + ativo);
    }
}
```

Saída esperada no console

Idade: 25

Salário: 4500.75

Inicial: J

Ativo: true

Conversões entre tipos primitivos (*casting*)

✔ Conversão implícita (automática)

Ocorre quando o tipo de destino é maior que o de origem.

```
int numero = 100;

long numeroMaior = numero; // OK: int → long
```

⚠ **Conversão explícita (*casting*)**

» Necessária quando há risco de perda de dados.

```
double altura = 1.75;

int alturaInteira = (int) altura; // Perde a parte decimal
```

As variáveis primitivas são fundamentais para armazenar dados de forma eficiente. Cada tipo tem um tamanho fixo e deve ser escolhido conforme a necessidade. O uso correto das variáveis otimiza desempenho e consumo de memória.

Variáveis não primitivas

O que são variáveis não primitivas?

As variáveis não primitivas, também chamadas de tipos de referência, são aquelas que armazenam um endereço de memória que aponta para um objeto, em vez de guardar diretamente um valor como as variáveis primitivas.

Diferente dos tipos primitivos, as variáveis não primitivas podem armazenar dados mais complexos, permitindo o uso de métodos e manipulações avançadas. Elas são fundamentais na programação orientada a objetos (POO), possibilitando a criação de estruturas de dados flexíveis.

Principais tipos de variáveis não primitivas

Em Java, os principais tipos de variáveis não primitivas incluem:

Tipo	Descrição	Exemplo
String	Armazena sequências de caracteres	`String nome = "Java";`
Arrays	Conjunto ordenado de valores do mesmo tipo	`int[] numeros = {1, 2, 3};`
Classes	Criam objetos personalizados com atributos e métodos	`Pessoa aluno = new Pessoa();`
Interfaces	Definem comportamentos para classes implementarem	`Runnable tarefa = new MinhaTarefa();`
Collections	Estruturas de dados como List, Set e Map	`List<String> lista = new ArrayList<>();`

Fonte: elaborado pelo autor.

Principais tipos de variáveis não primitivas

Em Java, os principais tipos de variáveis não primitivas incluem:

Tipo	Descrição	Exemplo
String	Armazena sequências de caracteres	`String nome = "Java";`
Arrays	Conjunto ordenado de valores do mesmo tipo	`int[] numeros = {1, 2, 3};`
Classes	Criam objetos personalizados com atributos e métodos	`Pessoa aluno = new Pessoa();`
Interfaces	Definem comportamentos para classes implementarem	`Runnable tarefa = new MinhaTarefa();`
Collections	Estruturas de dados como List, Set e Map	`List<String> lista = new ArrayList<>();`

Fonte: elaborado pelo autor.

String (Texto)

A variável ***String*** é usada para armazenar **sequências de caracteres**. Ao contrário dos tipos primitivos, ela é tratada como um **objeto** e possui métodos úteis para manipulação de texto.

```
String saudacao = "Olá, Mundo!";

System.out.println(saudacao.length()); // Exibe o tamanho da string

System.out.println(saudacao.toUpperCase()); // Converte para maiúsculas
```

Arrays (conjuntos de dados)

Os ***arrays*** permitem armazenar **múltiplos valores do mesmo tipo** em uma única variável. Eles são úteis para organizar coleções de dados de forma estruturada.

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

System.out.println(numeros[0]); // Exibe o primeiro elemento (10)
```

Classes e objetos (programação orientada a objetos – POO)

Uma **classe** define um **molde** para criar objetos, que representam entidades do mundo real. Variáveis não primitivas podem ser usadas para armazenar **objetos criados a partir de classes**.

```
class Pessoa {
    String nome;
    int idade;
    Pessoa(String nome, int idade) {
        this.nome = nome;
        this.idade = idade;
    }
}
public class Teste {
    public static void main(String[] args) {
        Pessoa aluno = new Pessoa(“João”, 25);
        System.out.println(“Nome: ” + aluno.nome + ”, Idade: ” + aluno.idade);
    }
}
```

Interfaces (modelagem de comportamentos)

As **interfaces** são um tipo especial de variável não primitiva. Elas definem um conjunto de métodos que devem ser implementados por outras classes.

```
interface Animal {
    void fazerSom();
}
class Cachorro implements Animal {
    public void fazerSom() {
        System.out.println(“Latido!”);
    }
}
public class Teste {
    public static void main(String[] args) {
        Animal pet = new Cachorro();
        pet.fazerSom(); // Exibe “Latido!”
    }
}
```

Obs.: aqui, *pet* é uma variável do tipo Animal, mas guarda um objeto da classe Cachorro.

Estruturas de dados (collections – list, set, map)

Além dos *arrays*, o Java possui classes especializadas para manipular coleções de dados de forma dinâmica, como listas, conjuntos e mapas.

Exemplo de Lista (*ArrayList*):

```
import java.util.ArrayList;

public class ExemploLista {

    public static void main(String[] args) {

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

        lista.add("Java");

        lista.add("Python");

        System.out.println(lista.get(0)); // Exibe
"Java"

    }

}
```

Obs.: Diferente dos *arrays*, uma *ArrayList* pode crescer dinamicamente conforme novos elementos são adicionados.

Estruturas de dados (collections – list, set, map)

Além dos *arrays*, o Java possui classes especializadas para manipular coleções de dados de forma dinâmica, como listas, conjuntos e mapas.

Exemplo de lista (*ArrayList*):

```
import java.util.ArrayList;

public class ExemploLista {

    public static void main(String[] args) {

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

        lista.add("Java");

        lista.add("Python");

        System.out.println(lista.get(0)); // Exibe
"Java"

    }

}
```

Obs.: Diferente dos arrays, uma ArrayList pode crescer dinamicamente conforme novos elementos são adicionados.

Comparação entre variáveis primitivas e não primitivas

Característica	Primitivas	Não Primitivas
Armazenamento	Guarda o valor diretamente	Guarda a referência de um objeto
Exemplo	`int idade = 30;`	`String nome = "Java";`
Aceitam métodos?	❌ Não	✅ Sim (ex.: `nome.length()`)
Podem ser nulas?	❌ Não	✅ Sim (`String nome = null;`)
Flexibilidade	Baixa (apenas valores fixos)	Alta (objetos, listas, manipulação avançada)

Fonte: elaborado pelo autor.

As variáveis não primitivas em Java são essenciais para lidar com dados complexos e objetos, diferentemente dos tipos primitivos (como int, char, boolean etc.), que armazenam valores diretamente. Enquanto os tipos primitivos guardam o valor em si, as variáveis não primitivas armazenam **referências** (ou endereços de memória) que apontam para o local onde o dado ou objeto está guardado. Esse mecanismo é fundamental para a programação orientada a objetos (POO), pois permite a criação e a manipulação de estruturas de dados mais sofisticadas.

Por que são importantes?

As variáveis não primitivas permitem trabalhar com estruturas de dados dinâmicas e complexas, como ***Strings*** (sequências de caracteres), ***Arrays*** (listas ordenadas de elementos), **Objetos** (instâncias de classes) e **Coleções** (como *ArrayList*, *HashMap* etc.). Esses tipos de dados são amplamente utilizados em aplicações reais, como sistemas *web*, bancos de dados e aplicativos móveis, pois oferecem maior flexibilidade e funcionalidade.

Vantagens:

- **Flexibilidade**: permitem criar estruturas de dados adaptáveis, como listas que crescem dinamicamente.
- **Reutilização de código**: por meio de classes e objetos, é possível encapsular comportamentos e atributos, seguindo os princípios da POO.
- **Gerenciamento de memória**: o Java gerencia automaticamente a alocação e liberação de memória para objetos não primitivos.

No desenvolvimento moderno, o uso de variáveis não primitivas é indispensável, especialmente em *frameworks* como ***Spring*** (para aplicações empresariais) e **Android** (para desenvolvimento móvel). Além disso, com a evolução da linguagem, recursos como ***Streams*** e ***Lambda Expressions*** (introduzidos no Java 8) facilitam a manipulação de coleções e objetos, tornando o código mais conciso e legível.

Dominar o uso de variáveis não primitivas é crucial para construir aplicações robustas, escaláveis e alinhadas com os princípios da POO.

5

Arrays e Matrizes

Em programação, armazenar e manipular conjuntos de dados de forma eficiente é essencial. Para isso, o Java oferece *arrays* e matrizes, que são estruturas usadas para organizar múltiplos valores dentro de uma única variável.

O que são *arrays*?

Um *array* é uma estrutura que permite armazenar vários elementos do mesmo tipo em uma sequência ordenada. Em vez de criar várias variáveis individuais, um *array* permite armazenar vários valores em uma única variável, acessando-os por meio de um índice numérico.

Os *arrays* em Java possuem um tamanho fixo, ou seja, uma vez criados, não podem ser redimensionados. Eles são úteis para lidar com coleções de dados de forma estruturada e eficiente.

Exemplo de uso de um *array* em Java:

```
public class ExemploArray {
    public static void main(String[] args) {
        // Criando um array de números inteiros
        int[] numeros = {10, 20, 30, 40, 50};
        // Acessando elementos do array
        System.out.println("Primeiro número: " + numeros[0]); // Exibe 10
        System.out.println("Último número: " + numeros[4]); // Exibe 50
        // Percorrendo o array com um laço
        System.out.println("Elementos do array:");
        for (int i = 0; i < numeros.length; i++) {
            System.out.println(numeros[i]);
        }
    }
}
```

Nesse exemplo, criamos um *array* chamado numeros que contém cinco valores inteiros. Podemos acessar cada elemento usando seu índice, que começa em 0.

Para percorrer o *array* inteiro, usamos um laço *for*, que itera por todos os índices e exibe os valores armazenados.

O que são matrizes?

Matrizes, também conhecidas como ***arrays* multidimensionais**, são estruturas de dados que permitem armazenar informações de forma organizada em linhas e colunas, semelhante a uma tabela. Enquanto um ***array* unidimensional** (ou vetor) armazena uma lista de elementos em uma única linha, as matrizes oferecem uma estrutura mais complexa, ideal para representar dados que dependem de múltiplas dimensões.

Matrizes bidimensionais (2D)

Uma matriz bidimensional (2D) é uma estrutura de dados que organiza elementos em linhas e colunas, semelhante a uma tabela. Em

muitas linguagens de programação, pode ser representada como um "*array* de *arrays*", onde cada linha é um *array* que contém os elementos da matriz. No entanto, em algumas linguagens, a matriz é armazenada como um bloco contínuo de memória.

Para acessar um elemento específico, utilizamos dois índices: o primeiro para a linha e o segundo para a coluna. Por exemplo, em uma matriz 3x3 (3 linhas e 3 colunas), o elemento da segunda linha e terceira coluna é acessado como [1][2], considerando que a indexação começa em 0.

Exemplo de uso de uma matriz em Java:

```
public class ExemploMatriz {

    public static void main(String[] args) {

        // Criando uma matriz 3x3

        int[][] matriz = {

            {1, 2, 3},

            {4, 5, 6},

            {7, 8, 9}

        };
```

```
        // Acessando um elemento específico
        System.out.println("Elemento da segunda linha e terceira coluna: " + matriz[1][2]); // Exibe 6

        // Percorrendo a matriz e exibindo todos os valores
        System.out.println("Elementos da matriz:");

        for (int i = 0; i < matriz.length; i++) { // Percorre as linhas
            for (int j = 0; j < matriz[i].length; j++) { // Percorre as colunas
                System.out.print(matriz[i][j] + " "); // Exibe o valor da posição [i][j]
            }
            System.out.println(); // Quebra de linha após cada linha da matriz
        }
    }
}
```

Nesse exemplo, criamos uma matriz 3x3, onde cada linha contém três elementos. Para acessar um valor específico, usamos dois índices: matriz[1][2] refere-se à segunda linha e terceira coluna (lembrando que os índices começam do zero). Para percorrer toda a matriz, utilizamos um laço *for* aninhado, em que o primeiro laço percorre as linhas e o segundo percorre as colunas dentro de cada linha.

Quando usar *arrays* e matrizes?

Arrays são ideais quando precisamos armazenar uma lista de valores do mesmo tipo, como notas de alunos, preços de produtos ou nomes de clientes. Matrizes são úteis quando os dados precisam ser organizados em uma estrutura mais complexa, como um tabuleiro de xadrez, um mapa de jogo, ou até mesmo dados tabulares (tabelas de preços, registros de funcionários etc.).

Conclusão

- *Arrays* e matrizes são fundamentais para o armazenamento e manipulação de múltiplos valores de forma eficiente.

- *Arrays* são unidimensionais e ideais para armazenar listas ordenadas de dados, enquanto matrizes permitem representar tabelas e estruturas mais complexas.
- O uso de laços de repetição é essencial para percorrer essas estruturas, garantindo maior eficiência no processamento dos dados.

Objetos

Em Java, objetos são a base da programação orientada a objetos (POO). Eles representam entidades do mundo real dentro do código, encapsulando dados (atributos) e comportamentos (métodos).

Enquanto variáveis armazenam valores simples, como números e textos, os objetos armazenam informações mais completas e organizadas.

O que é um objeto?

Um objeto é uma instância de uma classe. Ou seja, para criar um objeto, primeiro precisamos definir uma classe, que funciona como um "molde" ou "modelo".

Exemplo do mundo real:

Imagine que queremos representar um carro em Java. Um carro tem características como marca, modelo, cor e velocidade, além de comportamentos, como acelerar e frear. Para modelar isso em Java, criamos uma classe Carro e depois criamos objetos dessa classe.

Exemplo em código:

```
// Definição da classe Carro

class Carro {

    // Atributos (características do carro)

    String marca;

    String modelo;

    String cor;

    int velocidade;

    // Método para exibir informações do carro

    void exibirDetalhes() {

        System.out.println(“Carro: “ + marca + “ “ +
modelo + “ - Cor: “ + cor);
```

```
    }

    // Método para acelerar

    void acelerar(int aumento) {

        velocidade += aumento;

        System.out.println("O carro acelerou para " +
velocidade + " km/h");

    }

}

// Classe principal para testar os objetos

public class TesteCarro {

    public static void main(String[] args) {

        // Criando um objeto da classe Carro

        Carro meuCarro = new Carro();

        // Definindo valores para os atributos

        meuCarro.marca = "Toyota";

        meuCarro.modelo = "Corolla";

        meuCarro.cor = "Prata";
```

```
        meuCarro.velocidade = 0;

        // Chamando métodos do objeto

        meuCarro.exibirDetalhes();

        meuCarro.acelerar(50);

    }

}
```

Explicação do código

- Criamos a classe Carro, que contém atributos (marca, modelo, cor, velocidade) e métodos (exibirDetalhes() e acelerar(int aumento)).
- Dentro da classe principal TesteCarro, criamos um objeto meuCarro do tipo Carro.
- Definimos valores para os atributos do carro e depois usamos métodos para exibir os detalhes e acelerar.
- O objeto guarda informações e interage com o programa, tornando o código mais organizado e reutilizável.

Vários objetos da mesma classe

- Podemos criar vários objetos a partir da mesma classe, cada um com valores diferentes.

```
public class TesteVariosCarros {

    public static void main(String[] args) {

        Carro carro1 = new Carro();

        carro1.marca = “Honda”;

        carro1.modelo = “Civic”;

        carro1.cor = “Vermelho”;

        carro1.velocidade = 0;

        Carro carro2 = new Carro();

        carro2.marca = “Ford”;

        carro2.modelo = “Fiesta”;

        carro2.cor = “Azul”;

        carro2.velocidade = 0;

        carro1.exibirDetalhes();
```

```
        carro2.exibirDetalhes();

    }

}
```

Obs.: Cada objeto tem seus próprios valores e pode ser manipulado independentemente.

Diferença entre classe e objeto

Termo	Descrição	Exemplo
Classe	Modelo ou molde para criar objetos	`class Carro {}`
Objeto	Instância de uma classe, com atributos e métodos próprios	`Carro meuCarro = new Carro();`

Fonte: elaborado pelo autor.

Obs.: Pense na classe como uma receita de bolo e nos objetos como bolos feitos com essa receita.

Conclusão

- Objetos são fundamentais na programação orientada a objetos, permitindo organizar o código de forma eficiente.
- Eles encapsulam dados (atributos) e comportamentos (métodos), tornando o programa mais estruturado e reutilizável.

- Ao utilizar classes e objetos, podemos criar múltiplas instâncias e manipular dados de forma modular e flexível.

6

Laços de repetição em Java

Em programação, muitas vezes precisamos executar uma mesma ação várias vezes sem precisar repetir o código manualmente. Para isso, utilizamos os laços de repetição (ou *loops*).

Os laços permitem repetir blocos de código enquanto uma condição for verdadeira ou até atingir determinado número de execuções.

Tipos de laços em Java

Java oferece três tipos principais de laços de repetição:

- **for:** ideal quando sabemos o número exato de repetições.
- **while:** executa o código enquanto uma condição for verdadeira.

- **do-while:** semelhante ao *while*, mas garante que o bloco seja executado pelo menos uma vez.

Laço *for*

O laço *for* é mais utilizado quando sabemos **quantas vezes** um bloco de código precisa ser executado. Ele permite definir uma **inicialização**, uma **condição de parada** e um **incremento ou decremento** dentro de um único comando.

Esse tipo de laço é útil para percorrer listas, *arrays* e realizar repetições controladas de forma organizada.

Laço *while*

O laço *while* executa um bloco de código **enquanto** uma condição for verdadeira. Diferente do *for*, ele é mais indicado quando **não sabemos exatamente quantas vezes** a repetição ocorrerá, pois a execução depende de uma verificação feita antes de cada iteração.

Ele é frequentemente usado quando lidamos com **entrada de dados** do usuário, leituras de

arquivos ou processos que podem variar de acordo com condições dinâmicas.

Laço do-while

O laço do-while é semelhante ao while, mas com uma diferença importante: **ele executa o bloco de código pelo menos uma vez**, pois a verificação da condição ocorre apenas **depois** da primeira execução.

Isso o torna útil quando queremos garantir que o código seja rodado pelo menos uma vez antes de checar a condição. Um exemplo comum é pedir ao usuário para inserir um valor, garantindo que pelo menos uma entrada seja processada.

Laços aninhados

É possível utilizar um laço dentro de outro, formando **laços aninhados**. Isso é útil para percorrer **estruturas multidimensionais**, como matrizes, ou para gerar padrões estruturados.

Cada laço interno é executado completamente para cada iteração do laço externo, tornando essa abordagem essencial para situações como a manipulação de tabelas e listas encadeadas.

Controle de fluxo em laços

Java oferece dois comandos especiais para controlar a execução dos laços:

- **break** → interrompe o laço imediatamente e continua a execução do programa fora do *loop*.
- **continue** → pula a execução da iteração atual e avança para a próxima repetição do laço.

Esses comandos são úteis para evitar execuções desnecessárias e otimizar o fluxo do programa.

Conclusão

Os laços de repetição são fundamentais para tornar o código mais **eficiente, organizado e reutilizável**.

- O ***for*** é indicado para repetições com um número definido de iterações.
- O ***while*** é utilizado quando a quantidade de repetições **depende de uma condição**.
- O ***do-while*** garante que o código seja executado pelo menos uma vez.

- O uso de **laços aninhados** e comandos como *break* e continue permite maior controle sobre a execução.

Esses conceitos são amplamente utilizados em aplicações reais, desde **operações matemáticas** até **manipulação de grandes volumes de dados**.

Condicionais em Java

As estruturas condicionais permitem que um programa tome decisões com base em condições predefinidas. Elas são essenciais para controlar o fluxo de execução e permitir que o código responda a diferentes situações dinamicamente.

Tipos de condicionais em Java

Em Java, as principais estruturas condicionais são:

if → Executa um bloco de código **se** uma condição for verdadeira.

if-else → Define um **caminho alternativo** caso a condição seja falsa.

if-else if-else → Permite testar **múltiplas condições** sequencialmente.

switch → Uma alternativa ao *if-else if* para comparar um valor específico com várias opções.

Condicional *if*

O *if* é a estrutura mais simples e verifica **se uma condição é verdadeira**. Se for, o bloco de código correspondente é executado. Caso contrário, o programa segue sem executar esse trecho.

Essa estrutura é útil quando precisamos realizar verificações simples, como determinar se um número é positivo ou se um usuário tem permissão para acessar um sistema.

Condicional *if-else*

O *if-else* expande a funcionalidade do *if* ao permitir uma ação alternativa **caso a condição inicial não seja atendida**. Isso é útil quando há **dois cenários possíveis** e apenas um deve ser executado.

Por exemplo, pode ser usado para verificar se um aluno foi aprovado ou reprovado com base em sua nota. Se a condição do *if* for verdadeira, um conjunto de ações é realizado; se for falsa, o bloco *else* será executado.

Condicional *if-else if-else*

Essa estrutura permite testar **múltiplas condições** em sequência. Se a primeira condição não for atendida, o programa verifica a próxima, e assim por diante, até encontrar uma verdadeira ou executar o bloco *else*.

Isso é útil quando há **mais de duas possibilidades** para um mesmo cenário, como classificar um usuário em diferentes categorias com base na idade.

Condicional *switch*

O *switch* é uma alternativa ao *if-else if* quando precisamos comparar um valor específico com várias opções predefinidas. Ele verifica um único valor e executa o bloco correspondente à opção correspondente.

Essa estrutura é especialmente útil para **evitar múltiplos *if-else if* seguidos**, tornando o código mais organizado e legível, principalmente ao lidar com valores fixos, como dias da semana, opções de menu ou categorias de produtos.

Operadores relacionais e lógicos

As estruturas condicionais em Java utilizam **operadores relacionais** para comparar valores e **operadores lógicos** para combinar múltiplas condições.

- **Operadores relacionais**: ==, !=, <, >, <=, >=
- **Operadores lógicos**: && (E), || (OU), ! (NÃO)

Esses operadores são essenciais para criar expressões condicionais mais avançadas e flexíveis.

Conclusão

As **condicionais** são fundamentais para **controlar a execução do programa**, permitindo que ele reaja a diferentes situações.

- O ***if*** verifica uma condição simples.
- O ***if-else*** define uma ação alternativa.
- O ***if-else if-else*** permite múltiplas verificações sequenciais.
- O ***switch*** é útil para comparar um valor com várias opções.

O uso adequado dessas estruturas torna o código **mais inteligente, dinâmico e interativo**, permitindo decisões automáticas dentro do programa.