



Programação em Java

OXETECH LAB

Módulo 1

Introdução à Programação Java

- História da Linguagem Java
- Compilador
- Como o Java Funciona ?
- Estrutura Básica de um Programa Java
- Lógica de Programação
- Tipos de Dados e Variáveis
- Operadores
- Entrada e Saída de Dados
- Conversão de Tipos e Casting
- Funções e Métodos

História da linguagem java

Java foi criada em 1991, inicialmente como parte de um projeto para desenvolver uma linguagem de programação para dispositivos eletrônicos, como televisões e eletrodomésticos. O time liderado por **James Gosling** começou o projeto na empresa **Sun Microsystems**, e o nome original da linguagem era **Oak**, em homenagem a uma árvore de carvalho fora do escritório de Gosling.

Em 1995, a linguagem foi renomeada para **Java** e se tornou uma linguagem de programação de propósito geral, com ênfase na portabilidade entre diferentes plataformas de software e hardware. Isso significa que um programa escrito em Java pode ser executado em qualquer computador ou dispositivo que tenha a **Java Virtual Machine (JVM)**, sem precisar ser recompilado.

Java rapidamente ganhou popularidade devido a essas características e sua utilização em aplicativos de internet, especialmente após o lançamento dos **applets**, pequenos programas que podiam ser executados diretamente no navegador. Com o tempo, Java se expandiu para outras áreas, incluindo desenvolvimento de aplicativos para dispositivos móveis (principalmente através da plataforma Android), aplicações corporativas, e até mesmo jogos e sistemas embarcados.

A Oracle adquiriu a Sun Microsystems em 2010 e continuou o desenvolvimento de Java. Atualmente, Java é uma das linguagens de programação mais populares no mundo, com milhões de desenvolvedores e uma vasta quantidade de aplicações em diferentes áreas.

Compilador

O que é um compilador ?

Um compilador é um programa que traduz o código-fonte escrito em uma linguagem de programação de alto nível (como Java, C ou Python) para uma linguagem de nível mais baixo, como código de máquina ou bytecode, que pode ser executado diretamente pelo computador ou por uma máquina virtual. Em termos gerais, o compilador atua como um intermediário entre o programador e o hardware, transformando as instruções compreensíveis por humanos em instruções que a máquina pode processar.

O processo de compilação pode ser dividido em várias etapas, cada uma com uma função específica:

1. Análise léxica (Lexer): Nesta fase, o compilador quebra o código-fonte em pequenos componentes chamados tokens. Cada token representa uma unidade atômica da linguagem, como palavras-chave (if, while), identificadores (nomes de variáveis), operadores (+, -, *), etc. O objetivo é reconhecer cada parte do código e classificá-la de acordo com as regras da linguagem.

2. Análise sintática (Parser): O compilador então organiza os tokens em uma estrutura hierárquica, geralmente na forma de uma árvore de análise sintática ou árvore de sintaxe abstrata (AST).

3. Análise semântica: Depois de analisar a estrutura do código, o compilador verifica se as operações fazem sentido no contexto da linguagem. Por exemplo, se você tentar somar uma string com um número, a análise semântica identificaria isso como um erro, porque não é uma operação válida.

4. Otimização: Alguns compiladores realizam a otimização do código, o que significa que eles tentam melhorar o código gerado, eliminando instruções desnecessárias ou repetitivas e organizando as instruções para torná-las mais eficientes. O objetivo é gerar um código que seja executado de forma mais rápida ou que use menos recursos.

5. Geração de código: Nesta fase, o compilador gera o código de saída. Dependendo da linguagem, este código pode ser código de máquina (como ocorre em linguagens compiladas como C ou C++) ou bytecode (como no caso do Java). Este é o código que será executado pela máquina, seja diretamente pelo processador ou através de uma máquina virtual.

6. Ligação (Linking): Em muitas linguagens, a compilação envolve várias partes de um programa. O código compilado pode ser dividido em diferentes módulos ou bibliotecas, e a fase de ligação combina essas partes em um único programa executável.

O Compilador no Contexto do Java

Em Java, o compilador tem uma função ligeiramente diferente de outras linguagens que compilam diretamente para código de máquina. O compilador Java, conhecido como `javac`, converte o código-fonte Java para um formato intermediário chamado **bytecode**. Esse **bytecode** é armazenado em arquivos `.class`.

Aqui está como o compilador funciona no contexto do Java:

1. Código-fonte Java (.java): O programador escreve o código em arquivos com a extensão `.java`. Cada classe Java deve ser declarada em um arquivo separado.

2. Compilação para Bytecode: Quando o programador executa o comando `javac` seguido pelo nome do arquivo `.java`, o compilador Java analisa o código e o converte em **bytecode**. Esse **bytecode** não é executável diretamente pelo sistema operacional, mas é uma representação compacta e eficiente das instruções do programa. Os arquivos resultantes possuem a extensão `.class`.

3. Independência de Plataforma: Diferente de compiladores tradicionais que geram código de máquina específico para um tipo de hardware, o **bytecode** Java é independente de plataforma. Isso significa que o mesmo **bytecode** pode ser executado em qualquer sistema que possua uma Java Virtual Machine (JVM), sem a necessidade de recompilar o código para cada sistema operacional ou tipo de hardware.

4. Execução na JVM: O bytecode gerado pelo compilador é interpretado pela JVM, que traduz o bytecode em instruções específicas para o sistema operacional e o processador onde o programa está sendo executado. Isso dá ao Java sua famosa característica de portabilidade: o mesmo código pode rodar em diferentes sistemas sem precisar de alterações.

Funções Específicas do Compilador javac

Verificação de Erros: Durante a fase de compilação, o javac verifica o código-fonte em busca de erros de sintaxe (como uma chave de fechamento ausente) e de erros semânticos (como tentar acessar uma variável não declarada). O compilador não prossegue com a geração de bytecode até que todos os erros sejam corrigidos.

Conversão para Bytecode: Após verificar o código, o javac traduz o código-fonte para bytecode, armazenando-o em arquivos .class. Esses arquivos podem ser distribuídos e executados em qualquer ambiente Java, desde que tenha uma JVM instalada.

Múltiplas Classes: O javac pode compilar múltiplas classes ao mesmo tempo. Um único comando pode compilar vários arquivos .java para gerar os correspondentes arquivos .class. O compilador também pode lidar com dependências entre classes, garantindo que elas sejam compiladas na ordem correta.

Como o java funciona ?

Ao falar sobre o compilador javac já conseguimos entender um pouco de como o java funciona, mas para que fica bem claro vamos ver passo a passo como o java funciona. Vamos dividir o funcionamento do java em três componentes principais:

- **Código-fonte:** É o que o desenvolvedor escreve em arquivos .java. Contém as classes, métodos e instruções que compõem o programa.
- **Compilação para Bytecode:** O compilador Java (javac) converte o código-fonte em bytecode (arquivos .class). O bytecode é uma forma intermediária que não depende do sistema operacional, o que garante a portabilidade do Java.
- **Java Virtual Machine (JVM):** A JVM é responsável por interpretar e executar o bytecode. Cada plataforma (Windows, Linux, Mac, Android) tem sua própria implementação da JVM, mas todas seguem as mesmas especificações. Isso significa que o mesmo bytecode pode ser executado em diferentes plataformas sem alteração no código-fonte.

A vantagem desse sistema é que o desenvolvedor pode escrever o código uma única vez e ele será executado em qualquer lugar que tenha uma JVM instalada, seja um servidor, um computador pessoal ou um dispositivo móvel.

É importante entender também o que é **JRE** e **JDK**.

O **JRE (Java Runtime Environment)** é o ambiente necessário para executar programas escritos em Java. Ele contém tudo o que é necessário para rodar aplicativos Java, incluindo:

- **Máquina Virtual Java (JVM):** Componente responsável por interpretar e executar o bytecode Java.
- **Bibliotecas Java:** Conjunto de classes e bibliotecas padrões do Java que são usadas pelos aplicativos em tempo de execução.
- **Arquivos de suporte:** Recursos e ferramentas auxiliares para rodar o programa.

O JRE é o que permite que um aplicativo Java seja executado em uma máquina. Ele não inclui ferramentas de desenvolvimento, como compiladores e depuradores. Se você deseja apenas rodar um aplicativo Java, o JRE é suficiente.

O **JDK (Java Development Kit)** é o kit completo de ferramentas para desenvolver e executar programas Java. Ele inclui:

- O próprio **JRE** (com a JVM, bibliotecas e arquivos de suporte).

- Ferramentas de desenvolvimento, como:
 - **javac** (o compilador, que transforma código-fonte Java em bytecode).
 - **java** (ferramenta que executa programas Java).
 - **jdb** (depurador para encontrar e corrigir erros no código).
 - **javadoc** (gera documentação para o código Java).

O JDK é necessário se você estiver desenvolvendo aplicações Java, pois ele permite compilar o código-fonte e criar programas executáveis. Portanto, **o JDK é para desenvolvedores**, enquanto **o JRE é para usuários finais** que apenas querem executar os programas.

Estrutura básica de um programa java

Estrutura Geral de um Programa Java

Um programa Java começa com a definição de uma **classe**, que é a unidade básica da linguagem. Dentro dessa classe, existe o **método main**, que é o ponto de entrada da aplicação, ou seja, o primeiro lugar onde o programa começa a ser executado. Vamos analisar um exemplo simples de um programa Java:

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

Aqui estão os principais componentes desse código:

- **Classe:** No exemplo, a classe `HelloWorld` é declarada usando a palavra-chave `public class`. Em Java, todos os programas estão organizados em classes. Cada classe define um conjunto de atributos (dados) e métodos (funções ou comportamentos). A classe pode ser comparada a um modelo que define o comportamento de objetos.
- **Método main:** Esse é o ponto de partida do programa. A JVM começa a execução por aqui. A assinatura do método main sempre segue a forma `public static void main(String[] args)`. O

método é public, para ser acessível pela JVM, e static, porque ele não depende de uma instância da classe. O parâmetro `String[] args` permite que o programa receba argumentos da linha de comando.

- **System.out.println():** Esse método imprime uma mensagem no console. Ele faz parte da classe `System`, que é parte da biblioteca padrão do Java.

Características Fundamentais do Java

A estrutura de um programa Java segue uma ordem clara: você define uma ou mais **classes**, e dentro delas declara **atributos** e **métodos** que descrevem o comportamento de objetos criados a partir dessas classes. O ponto de entrada de execução do programa é o método `main`. Resumindo, em Java:

- **Classes** são moldes que definem o comportamento e as características de objetos.
- **Objetos** são instâncias dessas classes.
- **Métodos** definem as ações que esses objetos podem realizar.
- **Construtores** são usados para inicializar os objetos.

Esses conceitos básicos são a fundação para qualquer programa Java e para a programação orientada a objetos em geral.

Lógica de programação

A **lógica de programação** é o conjunto de princípios e técnicas usados para resolver problemas e automatizar tarefas através de instruções que o computador pode entender e executar. Para quem nunca programou, isso pode parecer complicado no começo, mas a ideia básica é criar uma sequência de passos lógicos, que chamamos de **algoritmo**, para atingir um objetivo.

O que é um algoritmo ?

Um **algoritmo** é uma sequência de instruções finitas, claras (não ambíguas) e bem definidas para resolver um problema ou realizar uma tarefa. Pense em um algoritmo como uma **receita de bolo**: a receita é composta por uma série de etapas que devem ser seguidas em ordem para que o bolo seja feito corretamente. Da mesma forma, um algoritmo é uma série de passos lógicos que o computador seguirá para realizar uma tarefa.

Exemplo de algoritmo para fazer um bolo:

1. Pegar os ingredientes.
2. Misturar os ingredientes.
3. Colocar a mistura no forno.
4. Esperar 30 minutos.
5. Tirar o bolo do forno.

Cada um desses passos é claro e segue uma ordem específica para garantir que o bolo seja feito corretamente. Um algoritmo em programação funciona de maneira semelhante, mas em vez de fazer um bolo, ele pode calcular números, processar informações, controlar o comportamento de um programa, entre outras tarefas.

Componentes Básicos da Lógica de Programação

Para entender a lógica de programação, é importante conhecer alguns conceitos básicos que formam a estrutura de qualquer algoritmo. Vamos explicar cada um desses conceitos de forma simples.

1. Instruções Sequenciais

Instruções sequenciais são as ações executadas uma após a outra, na ordem em que aparecem. Na lógica de programação, a sequência é o comportamento padrão: o computador executa uma linha de código após a outra.

Exemplo de um algoritmo sequencial simples:

- Passo 1: Leia dois números.
- Passo 2: Some os dois números.
- Passo 3: Exiba o resultado da soma.

Neste exemplo, cada instrução é executada na ordem em que foi

escrita. Não há nenhuma decisão ou repetição envolvida, apenas uma sequência de ações.

2. Decisões (Estruturas Condicionais)

Às vezes, precisamos tomar decisões com base em certas condições. Em um algoritmo, isso é feito através das **estruturas condicionais**, como o **if** em Java. Essas estruturas permitem que o programa tome diferentes caminhos dependendo das condições verificadas.

Por exemplo, imagine que você quer verificar se uma pessoa tem idade suficiente para votar:

- Se a idade for maior ou igual a 18, então exiba "Pode votar".
- Se a idade for menor que 18, então exiba "Não pode votar".

Isso seria representado assim em pseudocódigo:

```
Se idade >= 18 então
    Exibir "Pode votar"
Senão
    Exibir "Não pode votar"
```

Aqui, o algoritmo toma uma decisão com base na idade da pessoa. Esse tipo de estrutura permite que o programa mude seu comportamento dependendo de diferentes condições.

3. Repetições (Estruturas de Repetição)

Em muitas situações, é necessário repetir uma ação várias vezes. Isso é feito com as **estruturas de repetição** ou **loops**. Um loop executa um bloco de código várias vezes, até que uma condição seja satisfeita.

Exemplo: Imagine que você quer imprimir os números de 1 a 5. Isso seria feito com uma estrutura de repetição, como o **for** ou **while** em Java.

Pseudocódigo para imprimir os números de 1 a 5:

```
Para i de 1 até 5 faça  
    Exibir i  
Fim
```

Neste exemplo, o loop começa em 1 e vai até 5, exibindo o valor de **i** a cada repetição. Assim, o computador segue repetindo a ação até que a condição (**i <= 5**) não seja mais verdadeira.

4. Variáveis

As variáveis são usadas para armazenar valores que podem mudar durante a execução de um programa. Elas são como **caixas** onde podemos guardar informações. Cada variável tem um nome (que escolhemos) e um valor (que pode mudar).

Por exemplo, se você quiser armazenar a idade de uma pessoa, você pode criar uma variável chamada **idade** e atribuir a ela o valor da idade da pessoa. No pseudocódigo, ficaria assim:

```
idade = 25
```

Você pode alterar o valor da variável ao longo do programa:

```
idade = 30
```

Em Java, toda variável precisa ter um tipo, como **int** para números inteiros, **double** para números decimais, ou **String** para texto. Variáveis são fundamentais na lógica de programação porque nos permitem armazenar dados e usá-los mais tarde.

5. Funções/Métodos

À medida que os programas se tornam mais complexos, é útil dividir o código em partes menores e mais gerenciáveis. Isso é feito usando **funções** ou **métodos**. Uma função é um bloco de código que realiza uma tarefa específica e pode ser reutilizado várias vezes ao longo do programa.

Imagine que você tem um código que calcula a área de um quadrado várias vezes. Em vez de repetir o cálculo em todo o programa, você pode criar uma função que faz isso:

Função `calcularAreaQuadrado(lado)`

Retornar `lado * lado`

Fim

Agora, sempre que você precisar calcular a área de um quadrado, você pode simplesmente chamar essa função, passando o valor do lado do quadrado, em vez de reescrever o cálculo. Isso torna o código mais limpo e fácil de manter.

6. Entrada e Saída de Dados

Programas precisam interagir com o usuário, e isso é feito através da **entrada e saída de dados**. A entrada permite que o usuário forneça informações para o programa (como um número ou texto), e a saída exibe informações para o usuário.

Em pseudocódigo, um exemplo de entrada e saída seria:

Leia o número `do` usuário

Exiba o número + `1`

Em Java, isso seria feito usando a classe **Scanner** para leitura de dados e o **System.out.println()** para exibir os resultados. Entrada e saída são fundamentais para tornar os programas interativos.

Lógica de Programação na Prática

Para resumir, a **lógica de programação** envolve entender como organizar e estruturar esses componentes (sequências, decisões, repetições, variáveis, etc.) de forma lógica e coerente para resolver um problema ou executar uma tarefa. A habilidade de **quebrar um problema em pequenos passos** é essencial para se tornar um bom programador.

Exemplo Prático: Algoritmo para Calcular a Média de Dois Números

Aqui está um exemplo simples de um algoritmo que calcula a média de dois números fornecidos pelo usuário:

1. Leia o primeiro número.
2. Leia o segundo número.
3. Some os dois números.
4. Divida a soma por 2 para obter a média.
5. Exiba a média.

Pseudocódigo:

Leia `num1`

Leia `num2`

`soma` = `num1` + `num2`

`media` = `soma` / 2

Exibir `media`

Esse é um exemplo básico, mas demonstra como você pode usar variáveis, operações matemáticas e entrada/saída para resolver um problema simples.

Por que a Lógica de Programação é Importante?

Aprender **lógica de programação** é o primeiro passo para entender como construir software. Antes de aprender qualquer linguagem de programação, como Java, Python ou C, é crucial compreender a **estrutura lógica** por trás da programação.

Com uma boa base em lógica, você poderá entender e resolver problemas de forma eficiente, independentemente da linguagem que estiver usando. A prática de criar algoritmos e organizar pensamentos de maneira lógica ajudará a desenvolver sua capacidade de resolver problemas e construir programas funcionais.

Tipos de Dados e Variáveis

Quando programamos, lidamos com diferentes tipos de informações, como números, textos, caracteres e valores booleanos (verdadeiro ou falso). Para organizar e trabalhar com esses dados, as linguagens de programação usam **tipos de dados**. Em Java, os tipos de dados especificam o tipo de valor que uma variável pode armazenar e como o computador irá tratar esses valores.

A compreensão dos **tipos de dados** é fundamental, especialmente para iniciantes, pois ela nos ajuda a organizar o tipo de informação que será usada no programa e como essas informações podem ser manipuladas.

O que são Tipos de Dados?

Um **tipo de dado** define o tipo de valor que uma variável pode armazenar, e cada tipo de dado ocupa uma quantidade específica de memória. Em Java, os tipos de dados são divididos em duas grandes categorias:

- **Tipos de Dados Primitivos:** São os tipos mais básicos, diretamente suportados pela linguagem. Eles incluem números inteiros, números decimais, caracteres e valores booleanos.

- **Tipos de Dados Referenciados** (ou não primitivos): São os tipos mais complexos, como objetos e arrays, que são derivados de classes e definidos pelo programador.

Vamos explorar cada um desses tipos em detalhes, começando pelos **tipos primitivos**.

Tipos de Dados Primitivos

Java possui **oito** tipos de dados primitivos, que são usados para representar dados básicos, como números e caracteres. Esses tipos são:

1. byte
2. short
3. int
4. long
5. float
6. double
7. char
8. boolean

1. int (Números Inteiros)

O tipo int é usado para armazenar números inteiros, ou seja, números sem parte decimal. Este é um dos tipos de dados mais

comuns e amplamente usados em Java.

- **Exemplo:** 10, -5, 1000
- **Tamanho:** 4 bytes (32 bits)
- **Valor mínimo:** -2.147.483.648
- **Valor máximo:** 2.147.483.647

Exemplo de uso:

```
int idade = 25;  
int ano = 2024;
```

2. double (Números Decimais)

O tipo double é usado para armazenar **números com casas decimais** (também chamados de números de ponto flutuante). Ele é útil quando precisamos de mais precisão ao lidar com números fracionários.

- **Exemplo:** 10.5, -2.75, 3.14159
- **Tamanho:** 8 bytes (64 bits)
- **Valor mínimo:** $\pm 4.9E-324$
- **Valor máximo:** $\pm 1.7976931348623157E308$

Exemplo de uso:

```
double peso = 72.5;  
double altura = 1.80;
```

Note que, ao armazenar um valor decimal, usamos um ponto (.) em vez de uma vírgula.

3. float (Números Decimais com Menor Precisão)

O float também é usado para armazenar números decimais, mas tem menos precisão que o double. Por isso, ele ocupa menos memória.

- **Exemplo:** 5.75f, -3.14f (observe o "f" no final)
- **Tamanho:** 4 bytes (32 bits)
- **Valor mínimo:** $\pm 1.4E-45$
- **Valor máximo:** $\pm 3.4028235E38$

Exemplo de uso:

```
float temperatura = 36.6f;
```

Sempre que usamos um float, precisamos adicionar o sufixo "f" para indicar que o valor é um número float.

4. char (Caracteres)

O tipo `char` é usado para armazenar **um único caractere**. Um caractere pode ser uma letra, número ou símbolo, como 'A', '7', ou '?'.

- **Tamanho:** 2 bytes (16 bits)

- **Valores:** Qualquer caractere dentro do padrão Unicode, que inclui letras de vários idiomas, números e símbolos.

Exemplo de uso:

```
char letra = 'A';  
char simbolo = '#';
```

Note que os valores de char são colocados entre aspas simples (').

5. boolean (Valores Lógicos)

O tipo boolean armazena **valores lógicos**: true (verdadeiro) ou false (falso). Ele é útil para controlar fluxos de decisão em programas, como em estruturas condicionais (if e else).

- **Tamanho:** 1 bit (armazena apenas dois valores: true ou false)

Exemplo de uso:

```
boolean isJavaFacil = true;  
boolean temPermissao = false;
```

6. byte (Números Inteiros Pequenos)

O tipo byte é usado para armazenar **números inteiros pequenos**, ocupando menos memória que o int.

- **Tamanho:** 1 byte (8 bits)

- **Valor mínimo:** -128
- **Valor máximo:** 127

Exemplo de uso:

```
byte idade = 25;
```

7. short (Números Inteiros Pequenos)

O tipo short também armazena números inteiros, mas tem um intervalo maior que o byte, ocupando mais memória.

- **Tamanho:** 2 bytes (16 bits)
- **Valor mínimo:** -32.768
- **Valor máximo:** 32.767

Exemplo de uso:

```
short populacao = 15000;
```

8. long (Números Inteiros Grandes)

O long é usado para armazenar **números inteiros muito grandes**. Quando um número é muito grande para ser armazenado em um int, usamos o long.

- **Tamanho:** 8 bytes (64 bits)
- **Valor mínimo:** -9.223.372.036.854.775.808
- **Valor máximo:** 9.223.372.036.854.775.807

Exemplo de uso:

```
long distanciaEstrela = 9876543210L;
```

Note que usamos o sufixo "L" para indicar que o valor é um long.

Tipos de Dados Referenciados (Não Primitivos)

Os **tipos de dados referenciados** são mais complexos e incluem **arrays**, **objetos** e **strings**. Eles são baseados em classes, e os mais comuns são:

1. String (Texto)

Embora String não seja tecnicamente um tipo primitivo, ela é muito usada para armazenar texto em Java. Uma String é na verdade um objeto que contém uma sequência de caracteres.

- Exemplo: "Olá, Mundo!", "Java é divertido!"

Exemplo de uso:

```
String saudacao = "Olá, Mundo!";  
System.out.println(saudacao);
```

Strings são usadas em praticamente todos os programas para manipular textos e interagir com o usuário.

2. Arrays (Coleções de Dados)

Um **array** é uma coleção de dados do mesmo tipo. Você pode pensar em um array como uma lista de valores. Por exemplo, um array de inteiros pode armazenar uma lista de números.

Exemplo de uso:

```
int[] numeros = {1, 2, 3, 4, 5};
```

Aqui, o array `numeros` contém 5 elementos, todos do tipo `int`.

Por Que os Tipos de Dados São Importantes?

Os tipos de dados são essenciais porque ajudam o computador a entender como armazenar e processar as informações de forma eficiente. Eles também ajudam o programador a organizar o código e garantir que cada variável tenha o tipo de dado correto para a tarefa que será realizada.

Ao escolher o tipo de dado correto, você otimiza a memória usada pelo programa e evita erros lógicos, como tentar realizar operações matemáticas com textos ou usar um valor decimal em uma variável que deveria conter um inteiro.

Operadores

Em programação, os **operadores** são símbolos ou palavras reservadas que permitem realizar operações sobre os dados armazenados nas variáveis. Em Java, temos três principais categorias de operadores que são usados frequentemente:

1. **Operadores Aritméticos** – Realizam cálculos matemáticos.
2. **Operadores Relacionais** – Comparam valores e retornam verdadeiro (true) ou falso (false).
3. **Operadores Lógicos** – Combinam expressões e retornam valores booleanos.

Esses operadores são essenciais para controlar o fluxo dos programas e executar cálculos e comparações. Vamos analisar cada um deles com exemplos detalhados. Mas antes vamos tratar do **operador de atribuição**.

Operador de Atribuição

Em Java, o sinal de igual não tem a interpretação dada em matemática. Representa a atribuição da expressão à sua direita à variável à sua esquerda. Por exemplo:

```
x = 2000;
```

atribui o valor 2000 à variável de nome x. A ação é executada da direita para a esquerda.

Para ficar claro que a interpretação matemática não cabe aqui, vejo o seguinte exemplo:

```
in a = 3;  
int b = 7;  
a = a + b
```

Na interpretação da linguagem Java o que vai acontecer é que o valor de a (3) e b (7) serão somados e o resultado dessa soma será atribuído a variável a, então o novo valor de a será 10.

Já na interpretação matemática, a expressão “a = a + b” não faz sentido, pois implica que “a” é igual a “a” mais “b” ao mesmo tempo, o que é impossível.

Operadores Aritméticos

Os operadores aritméticos são usados para realizar operações matemáticas básicas, como adição, subtração, multiplicação, divisão e outras. Eles operam sobre **números inteiros** ou **números de ponto flutuante** (decimais).

Lista de Operadores Aritméticos:

Os operadores aritméticos são usados para realizar operações matemáticas básicas, como adição, subtração, multiplicação, divisão e outras. Eles operam sobre **números inteiros** ou **números de ponto flutuante** (decimais).

Lista de Operadores Aritméticos:

Operador	Função	Exemplo	Resultado
+	Adição	5 + 3	8
-	Subtração	10 - 7	3
*	Multiplicação	4 * 2	8
/	Divisão	10 / 2	5
%	Módulo (resto)	10 % 3	1

```
int soma = 10 + 20;           // soma será 30
int subtracao = 15 - 5;       // subtracao será 10
int multiplicacao = 5 * 4;     // multiplicacao será 20
int divisao = 20 / 4;          // divisao será 5
int resto = 11 % 2;            // resto será 1 (11 divi-
                                do por 2 tem resto 1)
```


Atenção: Quando é feita a divisão entre dois números e ambos os números são inteiros, o resultado será um valor inteiro, descartando qualquer parte decimal.

Ainda sobre no contexto do operadores aritméticos, temos os **operadores aritméticos de atribuição**. Esses operadores combinam as operações aritméticas com a operação de atribuição, oferecendo uma maneira curta e clara de escrita de certas expressões de atribuição.

A regra geral é:

$x \text{ op} = \text{exp}$ equivale a **$x = x \text{ op} (\text{exp})$**

Onde x é a variável, op é a operação (+, -, *, /) e exp é a expressão.

Exemplos:

$i += 3;$ equivale a $i = i + 3;$

$j -= 4;$ equivale a $j = j - 4;$

$k *= 2;$ equivale a $k = k * 2;$

$m /= 5;$ equivale a $m = m / 5;$

$n \%= 4;$ equivale a $n = n \% 4;$

Operadores Relacionais

Os operadores relacionais são usados para **comparar dois valores**. O resultado dessa comparação é sempre um valor booleano (true ou false). Eles são comumente usados em estruturas de controle, como if-else.

Lista de Operadores Relacionais:

Operador	Função	Exemplo	Resultado
==	Igualdade	5 == 5	true
!=	Diferente	5 != 3	true
>	Maior que	7 > 3	true
<	Menor que	2 < 5	true
>=	Maior ou igual a	5 >= 5	true
<=	Menor ou igual a	3 <= 5	true

Exemplos de Operações Relacionais em Java:

```
int a = 10;  
int b = 5;  
boolean resultado1 = (a > b); // true  
boolean resultado2 = (a == b); // false  
boolean resultado3 = (a != b); // true
```

Os operadores relacionais são amplamente usados em decisões, como em estruturas condicionais. Por exemplo:

```
if (a > b) {  
    System.out.println("a é maior que b");  
}
```

Operadores Lógicos

Os operadores lógicos são usados para **combinar duas ou mais condições** e também retornam valores booleanos (true ou false). Eles são particularmente úteis quando queremos verificar múltiplas condições ao mesmo tempo.

Lista de Operadores Lógicos:

Operador	Função	Exemplo	Resultado
&&	E lógico (AND)	(5 > 3) && (2 < 4)	true
	OU lógico (OR)	(3 > 1) (5 > 7)	true
!	Negação lógica (NOT)	!true	false

Exemplos de Operações Lógicas em Java:

```
int idade = 20;
boolean habilitada = true;
boolean podeDirigir = (idade >= 18) && habilitada;
if (podeDirigir) {
    System.out.println("Pode dirigir!");
}
```

Neste exemplo, usamos o operador **&&** para garantir que as duas condições sejam verdadeiras para permitir que a pessoa dirija.

Ainda temos outros operadores como os operadores de **incremento e decremento** e o **operador ternário**, vamos falar sobre eles.

Operadores de Incremento e de Decremento

O operador de incremento (++) opera sobre o nome de uma variável e adiciona 1 ao valor da variável operando. O operador de incremento pode ser usado de duas maneiras: prefixado, quando aparece antes do nome da variável, e pós-fixado, quando aparece em seguida ao nome da variável. As três instruções abaixo são equivalentes, pois todas incrementam o valor da variável x em 1.

```
x = x + 1
```

```
++x;
```

```
x++;
```

A diferença entre as operações executadas pelo operador *prefixado* e o *pós-fixado* aparece em instruções que fazem mais do que só incrementar a variável operando. Por exemplo:

```
int numero = 5;
```

```
System.out.println(++numero);
```

O operador de incremento prefixado incrementa o valor da variável numero e depois imprime na tela o valor incrementado, ou seja, 6.

```
int numero = 5;
```

```
System.out.println(numero++);
```

Já o operador de incremento pós-fixado, primeiro executa a instrução, ou seja, imprimir o valor de número que é 5 e só depois incrementa o valor. O código acima vai imprimir o valor 5.

A sintaxe e o modo de uso do operador de decremento (--) prefixado e pós-fixado é idêntica a do operador de incremento, exceto porque a variável é decrementada de 1.

Operador Condicional Ternário

O operador condicional possui uma construção um pouco estranha. É o único operador Java que opera sobre três expressões. Sua sintaxe geral possui a seguinte construção:

`exp1 ? exp2 : exp3`

A `exp1` é avaliada primeiro. Se o seu valor for 1 (verdadeira), a `exp2` será avaliada e o seu resultado será o valor da expressão condicional como um todo. Se a `exp1` for zero (falso), a `exp3` será avaliada e será o valor da expressão condicional como um todo. Por exemplo:

`max = (a > b) ? a : b;`

Se **a** for maior que **b**, então o resultado da expressão "**a > b**" será 1 (verdadeiro), logo o valor de **a** será atribuído a variável **max**. Caso contrário, se **a** não for maior que **b** a expressão "**a > b**" será 0 (falso), logo o valor de **b** será atribuído a variável **max**. A expressão acima está obtendo o maior valor entre **a** e **b**.

Entrada e Saída de Dados em Java

Em Java, a **entrada** e **saída** de dados (E/S) é a maneira pela qual o programa interage com o usuário ou com outros sistemas. A **entrada de dados** permite ao programa receber informações do usuário ou de outras fontes, enquanto a **saída de dados** exibe informações ao usuário ou envia dados para outros sistemas.

1. Entrada de Dados com a Classe Scanner

Para permitir a entrada de dados em Java, utilizamos a classe Scanner, que faz parte do pacote java.util. A classe Scanner facilita a leitura de diferentes tipos de dados fornecidos pelo usuário, como números inteiros, textos e números de ponto flutuante.

Como usar a classe Scanner ?

Para usar o Scanner, você precisa primeiro importar a classe e, em seguida, criar um objeto dela. O objeto criado pode ser usado para ler diferentes tipos de dados a partir da entrada padrão (normalmente o teclado).

Exemplo:

```
import java.util.Scanner;

public class EntradaDados {
    public static void main(String[] args) {
        // Criando o objeto Scanner para ler a entrada do usuário
        Scanner scanner = new Scanner(System.in);

        // Solicitando o nome do usuário
        System.out.print("Digite seu nome: ");
        String nome = scanner.nextLine(); // Lê uma linha de texto

        // Solicitando a idade do usuário
        System.out.print("Digite sua idade: ");
        int idade = scanner.nextInt(); // Lê um número inteiro

        // Exibindo os dados fornecidos
        System.out.println("Seu nome é " + nome + " e sua idade é " + idade);

        // Fechando o objeto Scanner
        scanner.close();
    }
}
```

Neste Exemplo:

- **scanner.nextLine()** é usado para ler uma linha completa de texto (o nome do usuário).
- **scanner.nextInt()** é utilizado para ler um número inteiro (a idade do usuário).

É importante fechar o objeto Scanner após a leitura dos dados, chamando o método `scanner.close()`.

2. Tipos de Métodos da Classe Scanner

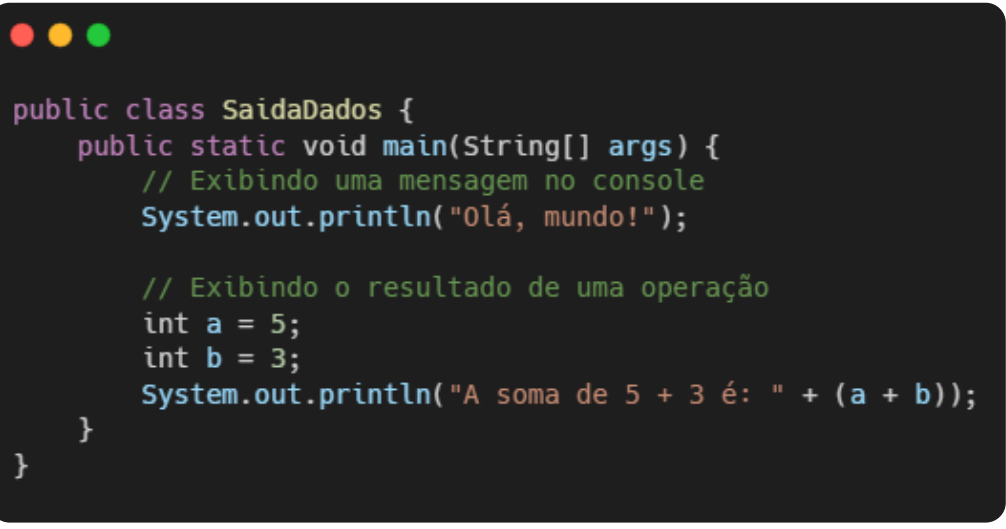
A classe Scanner oferece vários métodos para ler diferentes tipos de dados:

Método	Descrição
<code>nextByte()</code>	Lê um número inteiro do tipo byte
<code>nextShort()</code>	Lê um número inteiro do tipo short
<code>nextInt()</code>	Lê um número inteiro do tipo int
<code>nextLong()</code>	Lê um número inteiro do tipo long
<code>nextFloat()</code>	Lê um número decimal do tipo float
<code>nextDouble()</code>	Lê um número de ponto flutuante (decimal)
<code>nextLine()</code>	Lê uma linha completa de texto
<code>next()</code>	Lê uma única palavra
<code>nextBoolean()</code>	Lê um valor booleano (true ou false)
<code>next().charAt(0)</code>	Lê um caractere

3. Saída de Dados com `System.out.println`

A saída de dados em Java é feita utilizando o método `System.out.println()`, que imprime um texto seguido de uma nova linha no console. Ele é muito útil para exibir informações ao usuário, como resultados de cálculos ou mensagens de confirmação.

Exemplo:



```
public class SaidaDados {  
    public static void main(String[] args) {  
        // Exibindo uma mensagem no console  
        System.out.println("Olá, mundo!");  
  
        // Exibindo o resultado de uma operação  
        int a = 5;  
        int b = 3;  
        System.out.println("A soma de 5 + 3 é: " + (a + b));  
    }  
}
```

Neste exemplo:

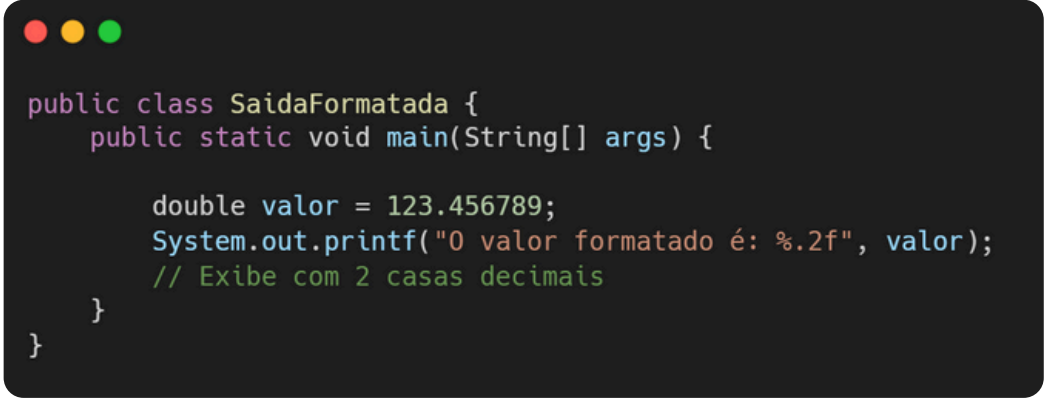
- **System.out.println()** exibe mensagens no console.
- O operador **+** é utilizado para concatenar (juntar) strings com variáveis e expressões matemáticas.

4. Comparação com Outros Métodos de Saída

Além do `System.out.println()`, o Java também oferece outros métodos para saída de dados:

- **System.out.print():** Imprime uma mensagem sem adicionar uma nova linha ao final.
- **System.out.printf():** Permite formatar a saída com precisão. É útil quando queremos controlar a forma como os dados são exibidos, como o número de casas decimais de um número.

Exemplo com `printf()`:



```
public class SaidaFormatada {  
    public static void main(String[] args) {  
  
        double valor = 123.456789;  
        System.out.printf("O valor formatado é: %.2f", valor);  
        // Exibe com 2 casas decimais  
    }  
}
```

Aqui, `%.2f` indica que queremos exibir o valor com 2 casas decimais.

Com essas ferramentas, você já consegue interagir com o usuário em seus programas, coletando informações e exibindo resultados de maneira eficiente.

Conversão de Tipos e Casting

Ao programar em Java, muitas vezes precisamos transformar um tipo de dado em outro. Isso pode ser necessário para realizar operações matemáticas, comparar valores, ou simplesmente manipular dados de maneiras diferentes. Esse processo é conhecido como **conversão de tipos** ou **casting**.

1. Tipos Primitivos em Java

Antes de falar sobre conversão de tipos, é importante lembrar que Java tem vários **tipos primitivos** de dados. Alguns dos mais comuns são:

- **int**: números inteiros (ex: 10, -5)
- **double**: números de ponto flutuante, ou seja, números com casas decimais (ex: 3.14, -0.001)
- **boolean**: valores verdadeiros ou falsos (true ou false)
- **char**: caracteres individuais (ex: 'a', 'b', '1')

2. Conversão Implícita

Java é capaz de realizar conversões automáticas entre tipos de dados, desde que a conversão não envolva perda de informação. Isso geralmente acontece quando convertemos de um tipo de dado menor para um tipo maior (por exemplo, de int para double).

Exemplo:

```
int numInteiro = 10;  
double numDecimal = numInteiro;
```

Neste exemplo, a conversão de int para double é feita automaticamente. O número inteiro 10 é convertido para 10.0 (um número decimal).

3. Conversão Explícita (Casting)

Nem todas as conversões podem ser feitas automaticamente. Quando há risco de perda de informação, como ao converter de um tipo maior para um tipo menor, é necessário utilizar o **casting** explícito. Isso significa que o programador deve dizer ao Java exatamente qual tipo de conversão ele deseja fazer.

O casting explícito é feito usando parênteses com o tipo de dado desejado antes da variável que queremos converter.

Exemplo:

```
double numDecimal = 9.78;  
int numInteiro = (int) numDecimal;
```

Neste exemplo, o número 9.78 é um double, mas ao convertê-lo para int com (int) numDecimal, o valor convertido será 9. Note que a parte decimal (0.78) é perdida, porque números inteiros não podem representar frações.

4. Tipos de Conversão

Abaixo estão os tipos de conversão que Java permite:

- **Conversão de menor para maior (implícita):** de byte para short, de short para int, de int para long, de long para float, e de float para double.
- **Conversão de maior para menor (explícita):** quando queremos converter um double para float, long para int, e assim por diante, devemos usar casting explícito.

Exemplo de Diferentes Conversões:

```
int numeroInteiro = 100;  
double numeroDouble = numeroInteiro;  
float numeroFloat = (float) numeroDouble;
```

5. Conversão de Strings para Números e Vice-Versa

Frequentemente, precisamos converter strings para números, por exemplo, ao coletar entrada do usuário, já que a classe Scanner lê tudo como texto. Para isso, usamos métodos específicos, como:

- **Integer.parseInt()** que é utilizado para converter uma String que contém um valor numérico em um número inteiro (int).

```
String numeroStr = "123";  
int numeroInt = Integer.parseInt(numeroStr);
```

- **String.valueOf()** que converte diferentes tipos de dados (como int, double, char, etc.) em uma String.

```
int numero = 456;  
String numeroStr = String.valueOf(numero);
```

6. Atenção com Erros de Conversão

É importante ter cuidado ao realizar conversões de tipos. Algumas conversões podem resultar em perda de dados ou erros em tempo de execução. Por exemplo, tentar converter uma string que não contém um número válido resultará em uma exceção.

```
String textoInvalido = "abc";  
int numero = Integer.parseInt(textoInvalido);
```

Funções e Métodos

Em Java, as **funções** são blocos de código reutilizáveis que executam uma tarefa específica. No contexto da linguagem Java, o termo mais comum para funções é **métodos**. Entender como os métodos funcionam é essencial para estruturar programas eficientes e bem organizados.

1. O que são Funções/Métodos

Funções, ou métodos, são conjuntos de instruções que realizam uma ação específica dentro de um programa. Elas permitem que você divida seu código em partes menores e mais gerenciáveis. Isso facilita a manutenção e a reutilização do código. Os benefícios de usar funções/métodos são:

- **Reutilização de código:** Uma vez que um método é criado, ele pode ser chamado várias vezes no programa.
- **Organização:** Ajuda a quebrar o programa em blocos lógicos.
- **Manutenção:** Com o código separado em métodos, é mais fácil corrigir erros ou modificar funcionalidades.
- **Modularidade:** Cada método cuida de uma tarefa específica.

2. Definição de um Método em Java

A estrutura de um método em Java é composta por vários elementos:


```
public class ExemploMetodo {  
    // Definição de um método simples  
    public static void saudacao() {  
        System.out.println("Olá, bem-vindo ao curso de Java!");  
    }  
  
    public static void main(String[] args) {  
        // Chamando o método saudacao  
        saudacao();  
    }  
}
```

Neste exemplo, criamos um método chamado `saudacao`, que imprime na tela “Olá, bem-vindo ao curso de Java!”, esse método é chamado no método principal (`main`).

Estrutura de um Método:

- **Modificadores de Acesso:** Define quem pode acessar o método (ex: `public`, `private`). Veremos sobre isso no Módulo 4.
- **Tipo de Retorno:** Indica o tipo de valor que o método vai retornar. Se o método não retornar nenhum valor, usamos `void`.
- **Nome do Método:** O nome do método, que deve ser um identificador único dentro de sua classe.
- **Parâmetros (opcionais):** São valores que podem ser passados para o método quando ele é chamado.
- **Bloco de Código:** O conjunto de instruções que o método vai executar, contido dentro de chaves `{}`.

3. Parâmetros de Métodos

Os **parâmetros** são dados que você passa para um método para ele trabalhar. Eles são definidos dentro dos parênteses na declaração do método.

Exemplo de Método com Parâmetro:

```
public class ExemploParametros {  
    // Método com parâmetros  
    public static void saudacaoComNome(String nome) {  
        System.out.println("Olá, " + nome + "! Bem-vindo ao curso de Java.");  
    }  
  
    public static void main(String[] args) {  
        // Chamando o método saudacaoComNome com um argumento  
        saudacaoComNome("João");  
    }  
}
```

Neste exemplo, o método `saudacaoComNome` recebe um parâmetro do tipo `String` chamado `nome`. Quando chamamos o método e passamos o valor `"João"`, ele usa esse valor para exibir a saudação personalizada.

Um método para ter múltiplos parâmetros, separados por vírgulas:

```
public static void mosrtar(String nome, int idade)
```

4. Retorno de Valor

Um método pode retornar um valor ao final de sua execução. O tipo de valor retornado é indicado no cabeçalho do método.

Exemplo de Método que Retorna Valor:

```
public static int soma(int a, int b) {  
    return a + b;  
}
```

Aqui, o método soma retorna um valor do tipo int. O valor calculado pela soma de a e b é devolvido ao ponto onde o método foi chamado e armazenado na variável resultado.

```
public static void main(String[] args) {  
    int resultado = soma(5, 3);  
}
```

O método soma foi chamado passando os valores 5 e 3, a função soma recebe esses valores e retorna o resultado de 5 + 3. Esse valor é armazenado na variável resultado.

5. Escopo de Variáveis

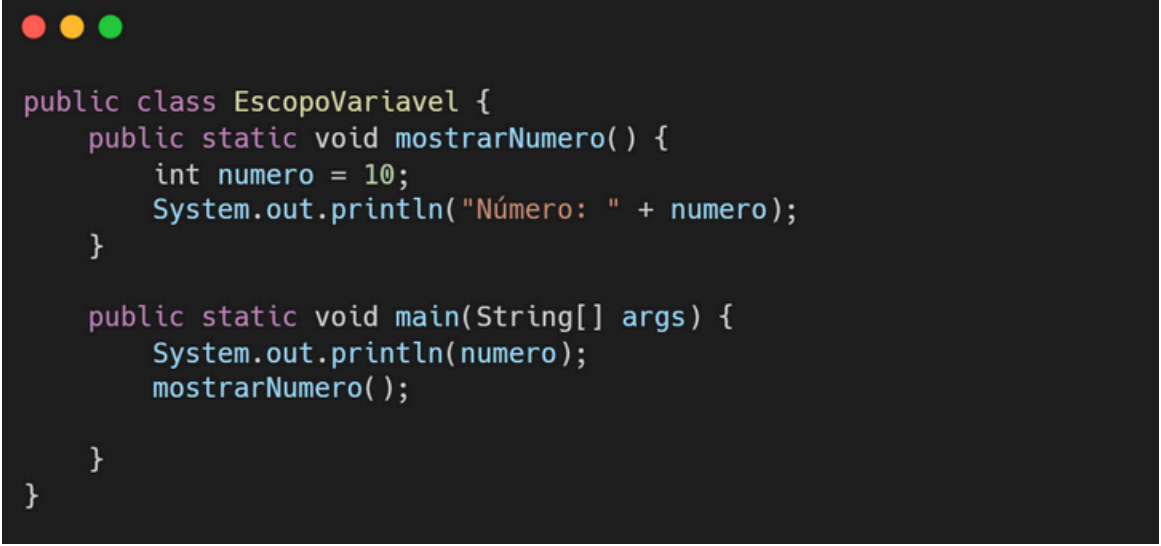
O **escopo** de uma variável refere-se à região do código onde ela é visível e pode ser usada. Em métodos, isso é muito importante para evitar confusão ao usar variáveis com o mesmo nome.

Existem dois tipos principais de escopo:

- **Escopo Local:** Variáveis declaradas dentro de um método são locais e só podem ser acessadas dentro desse método.

- **Escopo Global (ou de Classe):** Variáveis declaradas fora dos métodos, diretamente na classe, podem ser acessadas por todos os métodos da classe.

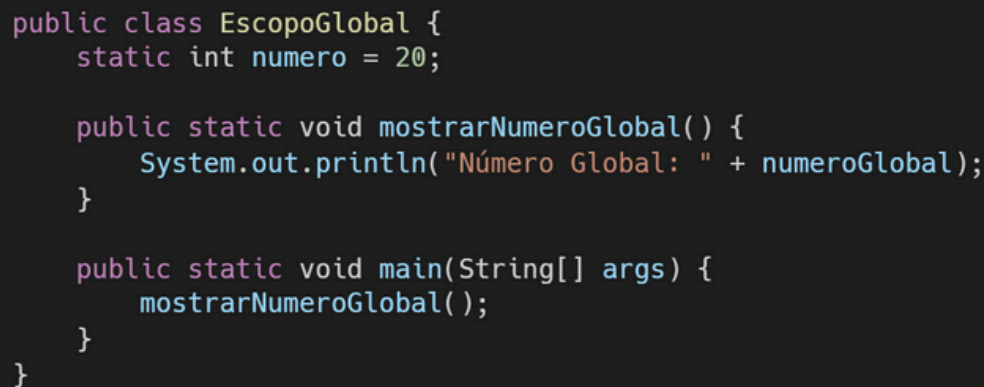
Exemplo de Escopo Local:



```
public class EscopoVariavel {  
    public static void mostrarNumero() {  
        int numero = 10;  
        System.out.println("Número: " + numero);  
    }  
  
    public static void main(String[] args) {  
        System.out.println(numero);  
        mostrarNumero();  
    }  
}
```

A variável `numero` só existe dentro do método `mostrarNumero`, pois foi declarada dentro do método. Ao tentar imprimir o valor da variável `numero` com `System.out.println(numero)` teremos um erro, pois a variável `numero` não existe dentro do método `main`. Porém podemos saber o valor da variável `numero` usando o método `mostrarNumero()`.

Exemplo de Escopo Global:



```
public class EscopoGlobal {  
    static int numero = 20;  
  
    public static void mostrarNumeroGlobal() {  
        System.out.println("Número Global: " + numeroGlobal);  
    }  
  
    public static void main(String[] args) {  
        mostrarNumeroGlobal();  
    }  
}
```

Neste exemplo, a variável `numero` não foi declarada dentro um método específico, isso faz com que ele seja “vista” por todos os métodos da classe `EscopoGlobal`.

6. Métodos Estáticos e Não Estáticos

Em Java os métodos podem ser **estáticos** ou **não estáticos**.

- **Métodos Estáticos:** Podem ser chamados diretamente usando o nome da classe, sem a necessidade de criar um objeto.
- **Métodos Não Estáticos:** São associados a objetos. Para chamar um método não estático, você precisa criar uma instância da classe.

Esse questão vai ficar mais clara quando falarmos sobre objetos no módulo 4.

Exemplo de Método Estático:

```
public class MetodoEstatico {  
    public static void mostrarMensagem() {  
        System.out.println("Este é um método estático.");  
    }  
  
    public static void main(String[] args) {  
        mostrarMensagem(); // Chamada direta, sem objeto  
    }  
}
```

Exemplo de Método Não Estático:

```
public class MetodoNaoEstatico {  
    public void mostrarMensagem() {  
        System.out.println("Este é um método não estático.");  
    }  
  
    public static void main(String[] args) {  
        MetodoNaoEstatico exemplo = new MetodoNaoEstatico();  
        exemplo.mostrarMensagem(); // Chamada através de um objeto  
    }  
}
```

7. Sobrecarga de Métodos

Java permite que você crie métodos com o **mesmo nome**, desde que eles tenham uma **assinatura** diferente (ou seja, um número ou tipo de parâmetros diferente). Isso é conhecido como **sobrecarga de métodos**.

Exemplo de Sobrecarga de Métodos:

```
public class SobrecargaMetodos {  
  
    // Método que recebe dois parâmetros int  
    public static int soma(int a, int b) {  
        return a + b;  
    }  
  
    // Método que recebe três parâmetros int  
    public static int soma(int a, int b, int c) {  
        return a + b + c;  
    }  
  
    public static void main(String[] args) {  
        System.out.println("Soma de 5 e 3: " + soma(5, 3));  
        System.out.println("Soma de 5, 3 e 2: " + soma(5, 3, 2));  
    }  
}
```

Neste exemplo, o método soma é sobrecarregado, aceitando diferentes números de parâmetros.

Exercícios

1. Qual a principal função do compilador em Java?
2. Como o bytecode gerado pelo compilador Java permite que o código seja executado em diferentes plataformas?
3. Explique o que são estruturas condicionais e dê um exemplo em pseudocódigo.
4. Escreva um algoritmo que resolva o problema de verificar se um número é par ou ímpar.
5. Escreva um programa que receba um número inteiro e realize operações de incremento e decremento, mostrando o valor antes e depois de cada operação.
6. Crie um programa que leia três números e faça operações aritméticas e lógicas com eles. Por exemplo, calcule a média dos três números e verifique se o valor é maior que 50, se for imprima true caso contrário imprima false.
7. Escreva um código que leia o nome de uma pessoa e exiba uma mensagem de boas-vindas personalizada.
8. Escreva um código simples que leia um número inteiro usando a classe Scanner.

- 9.** Escreva um código que leia a idade de uma pessoa e use operadores lógicos para verificar múltiplas condições, como se a idade está entre 16 e 18 (inclusivo) e se é maior que 65. Exiba os resultados de cada operação como true ou false.
- 10.** Crie um programa que leia dois números reais do usuário e exiba a soma deles no console.
- 11.** Escreva um programa que leia um valor em metros e converta para centímetros, exibindo o resultado.
- 12.** Escreva um código que leia um número decimal (ponto flutuante) e exiba sua parte inteira, realizando a conversão de float para int.
- 13.** Escreva um programa que receba um valor em double, converta para int usando casting explícito, e depois utilize esse valor em uma operação aritmética com outro número real.
- 14.** Crie um método chamado imprimirMensagem que receba uma string como argumento e exiba essa string no console.
- 15.** Crie um método sobrecarregado chamado calcularArea que receba diferentes parâmetros e calcule a área de um quadrado ou círculo, dependendo dos argumentos fornecidos.