Guia

Sumário

1 Introdução a Java \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ Página 2

2 Variáveis \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ Página 3

3 Operadores \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ Página 9

4 Tomando uma decisão \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ Página 20

5 IO básico \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ Página 25

6 Loops \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ Página 28

7 Métodos \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ Página 34

8 Noções básicas de Arrays \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ Página 39

9 Iterando sobre Arrays \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ Página 43

10 Operações de String \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ Página 46

11 Extra \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ Página 46

Introdução a Java

Java é uma linguagem de programação incrivelmente poderosa e versátil que permite criar tudo, desde aplicativos móveis até software empresarial, com a incrível capacidade de "escrever uma vez e executar em qualquer lugar".

# **Estrutura básica do programa**

Em Java, cada linha de código executada deve estar dentro de uma classe. Uma classe é como um projeto para a criação de objetos, que são instâncias da classe. A classe pode ter qualquer nome.

O main método é uma parte crucial de um programa Java porque serve como ponto de partida para a execução. Quando você executa um programa Java, a Máquina Virtual Java (JVM) procura o **main** método para iniciar a execução do código. Sem um **main** método, a JVM não saberia por onde começar.

Aqui está uma análise simples de um programa Java básico:

public class Main { // Declaração de classe

public static void main(String[] args) { // Main método

System.out.println("Hello, Coddy!"); // Declaração de saída

}

}

Observação importante: em Java, cada instrução deve terminar com um ponto e vírgula ( ;). O ponto e vírgula é obrigatório e informa ao Java que você chegou ao fim de uma instrução. Esquecer de adicionar um ponto e vírgula resultará em um erro de compilação. No entanto, observe que blocos de código entre chaves {} (como declarações de classe e método) não precisam de ponto e vírgula.

Variáveis

**Variáveis** são contêineres que armazenam valores de dados. Elas são usadas para armazenar, manipular e exibir informações dentro de um programa.

Resumindo, uma variável é como uma unidade de memória que podemos acessar digitando o nome da variável.

Cada variável tem um **nome** único e um **valor** que pode ser de diferentes tipos. Java possui vários tipos de dados integrados que definem o tipo de valor que uma variável pode conter.

Para inicializar uma variável, usamos o seguinte **formato** :

tipo\_de\_variável nome\_de\_variável = valor;

Ao declarar variáveis em Java, você precisa especificar o tipo da variável antes do nome dela. Isso é conhecido como **declaração de tipo** . Uma vez que uma variável é declarada com um determinado tipo, ela só pode conter valores desse tipo. Por exemplo, uma **int** variável só pode conter valores inteiros, e uma **String** variável só pode conter texto.

**Números**

Em Java, os números são normalmente representados usando dois tipos principais de dados: **int** e **double**.

**int** é usado para armazenar números inteiros sem vírgula decimal. Por exemplo:

int age = 30;

int temperature = -5;

int count = 100;

**double** é usado para armazenar números com ponto decimal. Por exemplo:

double price = 99.99;

double pi = 3.14159;

double fraction = 0.5;

# **String**

Um **char** é um único caractere (por exemplo: 1, 6, %, b, p, ., T, etc.)

O tipo String é um tipo especial que consiste em vários **char** s.

Para inicializar um valor de string em uma variável, coloque-o entre **aspas duplas** :

String s1 = "This is a string";

No exemplo acima, uma variável de string chamada s1é inicializada.

**Booleano**

Um tipo booleano tem apenas 2 valores possíveis: **true** ou **false**.

Para atribuir um valor booleano a uma variável, use a palavra-chave boolean seguida do nome da variável:

boolean variable\_true = true;

boolean variable\_false = false;

No exemplo acima, duas variáveis booleanas nomeadas variable\_true e variable\_false são inicializadas com os valores true e false, respectivamente.

**Char**

Um **char** é um único caractere (por exemplo: 1, 6, %, b, p, ., T, etc.). O tipo **char** é um tipo especial que consiste em um único caractere.

Para inicializar um valor char em uma variável, coloque-o entre aspas simples :

char c1 = 'h';

No exemplo acima, uma variável char chamada c1 é inicializada.

**Constantes**

Uma constante é um tipo especial de variável que não pode ser alterada depois de inicializada.

Para declarar uma constante, use a palavra-chave **final** seguida do tipo de variável:

final int MAX\_VALUE = 100;

No exemplo acima, uma constante chamada MAX\_VALUE é inicializada com o valor 100.

Se tentarmos alterar um valor constante:

final int MAX\_VALUE = 100;

MAX\_VALUE = 200; // Isso causará um erro

Isso resultará em um erro porque valores constantes não podem ser alterados.

**Convenções de Nomenclatura**

Em Java, é importante seguir as convenções de nomenclatura para manter seu código legível e de fácil manutenção. Aqui estão algumas regras importantes:

Nomes de Variáveis:

Use **camelCase**: Comece com uma letra minúscula e, em seguida, coloque a primeira letra de cada palavra subsequente em maiúscula (ex.: firstName, studentCount).

Escolha nomes descritivos que indiquem a finalidade da variável (ex.: userAge em vez de ua).

Evite nomes com apenas uma letra, exceto para contadores de loop

simples.

Nomes de Constantes:

Use **UPPER\_SNAKE\_CASE**: Escreva em letras maiúsculas, com as palavras separadas por sublinhados (ex.: MAX\_VALUE, PI\_VALUE).

Use para valores que não mudam ao longo do programa.

Regras Gerais:

Os nomes podem conter letras, dígitos, sublinhados e cifrões.

Os nomes devem começar com uma letra, um sublinhado \_ ou um cifrão $.

Os nomes diferenciam maiúsculas de minúsculas (minhaVariável é diferente de minhavariável).

Evite usar palavras-chave reservadas do Java (como int, class, public, etc.).

Seguir essas convenções ajuda a tornar seu código mais compreensível, especialmente ao trabalhar em equipe ou revisitá-lo posteriormente.

**Fundição de Tipos**

Conversão de tipo é o processo de conversão de um valor de um tipo de dado para outro.

Em Java, podemos converter inteiros em doubles, doubles em inteiros e muito mais. Existem dois tipos de conversão: implícita (automática) e explícita (manual).

Por exemplo **integer to double**:

Conversão implícita (automática):

int number = 5;

double decimal = number; // torna-se automaticamente 5.0

// com cálculo

int x = 7;

double result = x / 2.0; // o resultado é 3.5

Elenco explícito (manual) **double to integer**:

double decimal = 9,7;

int number = (int) decimal; // torna-se 9 (a parte decimal é truncada)

// com cálculo

double price = 19.99;

int roundedPrice = (int) price; // torna-se 19

Também é possível converter números e booleanos para strings e vice-versa. Para converter um valor para strings, podemos usar a **String.valueOf()** função:

int number1 = 789;

double number2 = 789;

boolean isValid = true;

String text1 = String.valueOf(number1); // becomes "789"

String text2 = String.valueOf(number2); // becomes "789.0"

String text3 = String.valueOf(isValid); // becomes "true"

Converter uma string para um tipo diferente é um pouco mais complicado:

**String to Integer:**

String numberText = "123";

int number = Integer.parseInt(numberText); // torna-se 123

**String to Double:**

String decimalText = "45.67";

double decimal = Double.parseDouble(decimalText); // torna-se 45.67

**String to Boolean:**

String boolText = "true";

boolean bool = Boolean.parseBoolean(boolText); // torna-se true

**parseBoolean** converterá qualquer string que não diferencia maiúsculas de minúsculas que tenha o valor **“true”**. Por exemplo **True**, **tRue**, **TRUE** todos se tornarão verdadeiros

Tentar converter uma string em um tipo inválido resultará em um erro:

String invalidNumber = "abc";

int number = Integer.parseInt(invalidNumber); // Isso causará uma NumberFormatException

Operadores

**Operadores Aritméticos**

Operadores são usados para executar operações em valores.

Primeiro, discutiremos os operadores aritméticos mais básicos , eles podem ser familiares nas aulas de matemática.

| **Operador** | **Operação** | **Exemplo** |
| --- | --- | --- |
| + | Adição | 3 + 2 = 5 |
| - | Subtração | 3 - 2 = 1 |
| \* | Multiplicação | 3 \* 2 = 6 |
| / | Divisão | 4/2 = 2 |

Vamos ver um exemplo de uso,

int a = 3;

int b = 5;

int c = a + b; // c contém 8

Ao trabalhar com números decimais em Java, usamos o tipo de dado double, que pode armazenar números com pontos decimais. Os mesmos operadores aritméticos (**+**, **-**, **\***, **/**) funcionam com doubles da mesma forma que com inteiros:

double x = 3.3;

double y = 4.1;

double z = x + y; // z contém 7.4

**Operador de módulo**

O operador módulo **%** fornece o resto de uma divisão. Em Java, ele é usado com uma sintaxe simples:

**result = dividend % divisor;**

dividend: O número que está sendo dividido.

divisor: O número que divide o dividendo.

result: O restante da divisão.

Por exemplo:

**result = 10 % 3;**

Aqui, 10 é dividido por 3. 3 cabe em 10 três vezes, com resto de 1. Então, result será 1.

Normalmente o módulo é usado para verificar se um número é par ou ímpar:

Se um número for par, dividi-lo por 2 deixará um resto de 0.

Se um número for ímpar, dividi-lo por 2 deixará um resto de 1.

Ao usar módulo com números de ponto flutuante (duplos), o funcionamento é semelhante ao dos inteiros, mas a precisão decimal é mantida. Por exemplo:

double result = 5.2 % 2.0;

// O resultado é 1.2

Veja como funciona: 2,0 cabe em 5,2 duas vezes (4,0), e o resto é 1,2 (5,2 - 4,0 = 1,2).

Outro exemplo:

double result = 7.8 % 3.5;

// O resultado é 0.8

**Incremento/Decremento**

Operadores de incremento e decremento são usados para aumentar ou diminuir o valor de uma variável em 1. Esses operadores são amplamente usados em programação, especialmente em loops e contadores.

O operador de incremento é representado por dois sinais de mais **++**, e o operador de decremento é representado por dois sinais de menos **--**.

Por exemplo, para incrementar uma variável chamada count, você pode usar o operador de incremento como este:

int count = 5;

count++; // count é agora 6

Da mesma forma, para decrementar uma variável chamada value, você pode usar o operador de decremento como este:

int value = 10;

value--; // value é agora 9

**Pós-incremento/decremento**

Os operadores de incremento (++) e decremento (--) podem ser usados de duas maneiras:

Pré-incremento/decremento (++x ou --x):

1.O operador vai ANTES da variável

2.O valor muda IMEDIATAMENTE

3.O novo valor é usado na expressão

int x = 5;

int y = ++x;

//x é aumentado para 6 primeiro, depois y se torna 6

Pós-incremento/decremento (x++ ou x--):

1.O operador vai DEPOIS da variável

2.O valor original é usado primeiro

3.O valor muda APÓS a expressão

int x = 5;

int y = x++;

// y se torna 5 primeiro, depois x aumenta para 6

Outro exemplo

pós-incremento:

int score = 5;

int res1 = score++;

// res1 é 5

// score é 6

pré-incremento:

int score = 5;

int result2 = ++score;

// result2 é 6

// score é 6

**Atalhos Aritméticos**

Java criou um atalho interessante para operações auto aritméticas.

Por exemplo, em vez de escrever:

int a = 5;

a = a + 3; // a contém 8

Podemos simplificar escrevendo +=:

int a = 5;

a += 3; // a contém 8

O **+=** está adicionando a **a** si mesmo somando o valor 3

Esta operação é válida para todas as operações aritméticas:

| Operador | Atalho |
| --- | --- |
| + | += |
| - | -= |
| \* | \*= |
| / | /= |
| % | %= |

**Operadores de comparação**

Operadores de comparação são usados para comparar entre dois operandos.

Às vezes, precisamos verificar se um operando é maior/menor/... que outro operando. A tabela a seguir mostra os operadores possíveis para comparação:

| Operador | Significado | Exemplo |
| --- | --- | --- |
| == | Igual | 1 == 2 retorna falso |
| != | Não igual | 1 != 2 retorna verdadeiro |
| > | Maior que | 1 > 2 retorna falso |
| < | Menor que | 1 < 2 retorna verdadeiro |
| >= | Maior ou igual | 1 >= 2 retorna falso |
| <= | Menor ou igual | 1 <= 2 retorna verdadeiro |

O operador de comparação retorna verdadeiro se a comparação estiver correta ou falso caso contrário.

Por exemplo:

int var1 = 13;

int var2 = 12;

boolean var3 = var1 != var2;

var3 será verdadeiro porque var1 e var2 não são iguais.

**Comparação de strings**

Em Java, comparar strings é diferente de comparar números. Embora você possa usar **<**, **>**, **==**, etc., para números, strings requerem métodos diferentes porque são objetos, não primitivos.

A maneira mais comum e correta de comparar se duas strings têm o mesmo conteúdo é usando o equals()método:

String str1 = "hello";

String str2 = "hello";

String str3 = "Hello";

boolean result1 = str1.equals(str2); // true

boolean result2 = str1.equals(str3); // false (case-sensitive)

Usando **equalsIgnoreCase()** Se você quiser comparar strings sem considerar letras maiúsculas/minúsculas:

String str1 = "Hello";

String str2 = "hello";

boolean result = str1.equalsIgnoreCase(str2); // true

Muitos iniciantes tentam usar **==** para comparar strings, mas isso geralmente está errado:

String str1 = "hello";

String str2 = "hello";

boolean result = (str1 == str2);

Não compare string com o **==** sinal!

**Operadores Lógicos**

Operadores lógicos são usados para verificar combinações de comparações que retornam true ou false.

Por exemplo, a seguinte declaração contém duas comparações:

5 é maior que 3 e menor que 6?

| Operador | Significado | Exemplo |
| --- | --- | --- |
| && | E - true se todos os operandos forem true | a && b |
| || | Ou - true se algum operando for true | a || b |
| ! | Não - true se o operando for false | !a |

Vamos ver alguns exemplos,

5 é maior que 3 e 1 é igual a 1,

boolean b1 = (5 > 3) && (1 == 1); // é true

Explicação : Todos os operandos são true, então b1 será true (e a operação será verdadeira se ambos os operandos forem verdadeiros).

5 não é igual a 4 nem cinco é igual a 2,

boolean b2 = !(5 == 4) || (5 == 2); // é true

Explicação : O primeiro operando (5 != 4) é true, então b2 também é true (ou a operação é verdadeira se qualquer um dos operandos for verdadeiro)

1 não é igual a 1 ou falso,

boolean b3 = !(1 == 1) || false; // é false

Explicação : Todos os operandos são falsos, então b3 manterá falso (ou operação).

não (3 maior que 4),

boolean b4 = !(3 > 4); // é true

Explicação : O operando é false, então b4 conterá true ( não operação).

não (5 maior que 10 ou 5 maior que 1),

boolean b5 = !(5 > 10 || 5 > 1); // é false

Explicação : 5 > 10 || 5 > 1 é true(um dos operandos é true), então no total b5 é false ( não operação).

Operadores lógicos têm uma tabela especial chamada "Tabela Verdade" que mostra o que a combinação de operadores lógicos retorna.

Tabela verdade para o operador and (&&):

| a | b | a && b |
| --- | --- | --- |
| false | false | false |
| false | true | false |
| true | false | false |
| true | true | true |

A única maneira de obter um resultado verdadeiro para o operador and (&&) é se a e b forem verdadeiros.

Tabela verdade para o operador or (||):

| a | b | a || b |
| --- | --- | --- |
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | true |

Neste caso, para obter um resultado verdadeiro, a ou b deve ser verdadeiro.

Tabela verdade para o operador not (!):

| a | !a |
| --- | --- |
| false | true |
| true | false |

Aqui, o valor de a é invertido. Se a for falso, então !a é verdadeiro.

Ao verificar múltiplas condições, o computador interrompe a verificação assim que conhece a resposta final (isso é chamado de avaliação em curto-circuito).

Por exemplo:

int x = 0;

int y = 5;

boolean result = x != 0 && y / x > 2;

Aqui, x é igual a 0, portanto, ele não avaliará y / x > 2. Se invertermos a ordem:

boolean result = y / x > 2 && x != 0;

Isso resultará em um erro, pois y será dividido por 0, o que é ilegal em matemática.

Essa técnica é usada para otimizar a avaliação de expressões lógicas. Por exemplo:

int a = 0;

int b = 2;

int c = 3;

int d = 5;

boolean result = (a > 0 && b < 2) || (c < -5 && d < 10);

Neste exemplo, b < 2 e d < 10 não serão avaliados porque a > 0 e c < -5 são ambos falsos.

A Lei de De Morgan é uma regra que nos ajuda a simplificar expressões que contêm NOT. Ao trabalhar com expressões lógicas, às vezes precisamos simplificá-las e reorganizá-las.

not Diante de duas condições unidas por e, você pode dividi-las em duas partes separadas. O and se torna ou, e cada parte recebe seu próprio não:

!(A && B)é o mesmo que(!A) || (!B)

Por exemplo:

// Vamos verificar se um número NÃO é (entre 1 e 10)

inter number = 15;

// Estas duas expressões são equivalentes:

boolean resultado1 =!(número >= 1 && número <= 10);

boolean resultado2 = !(número >= 1) || !(número <= 10);

System.out.println(resultado1); // true

System.out.println(resultado2); // true

O oposto também está correto: !(A || B)é o mesmo que(!A) && (!B)

Por exemplo:

// Verificando se uma pessoa NÃO é (estudante ou empregada)

boolean isStudent = false;

boolean isEmployed = false;

// Estas duas expressões são equivalentes:

boolean result1 = !(isStudent || isEmployed);

boolean result2 = !(isStudent) && !(isEmployed);

System.out.println(result1); // true

System.out.println(result2); // true

Tomando uma decisão

**Declaração If**

As instruções if nos permitem executar código com condições.

Por exemplo, vejamos o seguinte código:

int age = 20;

String status = "Child";

if (age > 18) {

status = "Adult";

}

age += 1;

O código acima verifica se a **age** variável é maior que 18. Se for, ele a definirá status para conter "Adult"string.

No final, o código será incrementado age de acordo com 1 a idade, seja ela maior que 18 anos ou não.

Para usar uma if instrução em Java, precisamos usar chaves {} para definir o bloco de código, e tudo dentro da instrução if deve ser colocado entre essas chaves:

if (condition) {

code;

code;

code;

}

Se a condição for true, colocaremos o bloco de código dentro do **if** (o código recuado)

**If - Else**

**if** nos permite executar um código específico se uma condição for atendida, mas e se quisermos executar outra coisa se a condição não for atendida?

Para isso, temos a instrução else:

int age = 15;

String status = "None";

if (age >= 18) {

status = "Adult";

} else {

status = "Young";

}

No exemplo acima, age é menor que 18, o que significa que entra no código else, e status conterá "Young".

Podemos até tornar isso mais complexo usando a instrução **else if**:

int age = 68;

String status = "None";

if (age < 18) {

status = "Young";

} else if (age >= 18 && age <= 65) {

status = "Adult";

} else {

status = "Old";

}

Aqui, ele verifica se a idade é menor que 18 anos; caso contrário, ele passa para a próxima condição e verifica se a idade está entre 18 e 65 anos. Se essa condição também não for atendida, o status será definido como "Antigo".

Podemos adicionar quantas instruções else if quisermos:

if (condição1) {

código;

} else if (condição2) {

código;

} else if (condição3) {

código;

}

**Instrução Switch**

A instrução **switch** é como uma instrução if multidirecional. Em vez de avaliar uma única condição, ela verifica o valor de uma variável em relação a vários casos e executa o código associado ao caso correspondente.

Aqui está a estrutura básica de uma instrução switch:

switch (variável) {

case valor1:

// Código a ser executado se a variável for igual a valor1

break;

case valor2:

// Código a ser executado se a variável for igual a valor2

break;

// ... mais casos

default:

// Código a ser executado se nenhum caso corresponder

}

A palavra-chave **switch** é seguida pela variável que você deseja testar entre parênteses.

Cada case representa um possível valor da variável.

O código dentro de cada caso é executado se a variável corresponder ao valor daquele caso.

A instrução break é crucial; ela encerra o switch após a execução de um caso. Sem ela, a execução "cairia" para o próximo caso.

O default case padrão é opcional e é executado se nenhum outro caso corresponder.

Aqui está um exemplo:

int dia = 3;

String dayName;

switch (dia) {

case 1:

dayName = "Segunda-feira";

break;

case 2:

dayName = "Terça-feira";

break;

// ... casos para outros dias

default:

dayName = "Dia inválido";

}

Você também pode combinar vários casos em um:

int day = 3;

String dayName;

switch (dia) {

case 1:

case 2:

case 3:

dayName = "Início da semana";

break;

// ... casos para outros dias

default:

dayName = "Dia inválido";

}

**Operador Ternário**

O operador ternário é uma instrução simples de uma linha **if-else**. Possui a seguinte sintaxe:

variable = (condition) ? value\_if\_true : value\_if\_false;

O operador ternário avalia a condição. Se for true, ele atribui value\_if\_true à variável; caso contrário, ele atribui value\_if\_false.

Por exemplo:

int age = 20;

String message = (age >= 18) ? "Adult" : "Minor";

Neste exemplo, se age for maior ou igual a 18, message será atribuído o valor "Adult". Se age for menor que 18, message será atribuído "Minor".

**Se - Senão aninhados**

Podemos aninhar if-elif-else declarações umas dentro das outras. Isso nos permite criar estruturas hierárquicas de tomada de decisão.

Por exemplo:

if (age > 18) {

if (hasLicense) {

System.out.println("You can drive");

} else {

System.out.println("Get a license first");

}

} else {

System.out.println("Too young to drive");

}

Pode ser aninhado infinitamente:

if (condition1) {

if (condition2) {

if (condition3) {

// if condition1, condition2 and condition3 are true

...

}

}

}

IO básico

**Métodos System.out**

Em Java, o **System.out** objeto fornece vários métodos para imprimir a saída no console. Aqui estão alguns dos System.out métodos mais comumente usados:

**print(String s)**: Imprime uma string no console. Não adiciona um caractere de nova linha no final, portanto, a saída subsequente continuará na mesma linha.

**println(String s)**: Imprime uma string no console seguida por um caractere de nova linha . Isso significa que a próxima saída começará em uma nova linha.

Veja como você pode usar esses métodos:

String name = "Alice";

int age = 30;

System.out.print("Name: ");

System.out.print(name);

System.out.println(" is " + age + " years old.");

System.out.println("Hello, " + name + "!");

Este código produzirá a seguinte saída:

Name: Alice is 30 years old.

Hello, Alice!

**Formatando a Saída**

Até agora, aprendemos a imprimir strings simples, mas às vezes precisamos inserir valores de variáveis na string.

Por exemplo:

int idade = 10;

System.out.println("Sua idade é: idade");

Isso imprimirá "Sua idade é: idade" em vez de "Sua idade é: 10".

Para que funcione, usaremos a impressão formatada printf:

int idade = 30;

String nome = "Alice";

double saldo = 1500.75;

System.out.printf("Nome: %s, Idade: %d, Saldo: %.2f\n", nome, idade, saldo);

**%s** é um espaço reservado para strings.

**%d** é um espaço reservado para inteiros.

**%f** é um espaço reservado para números de ponto flutuante.

**%.2f** formata o número de ponto flutuante com duas casas decimais.

Outra maneira de combinar strings com variáveis é com o operador **+**:

System.out.print("Nome: " + nome + " Idade: " + idade + " Saldo: " + saldo);

**Classe de scanner**

Até agora, armazenamos valores que pensamos em variáveis. Programas geralmente não funcionam dessa maneira. Recebemos valores de uma fonte externa, um usuário, por exemplo.

Em Java, a obtenção de dados de um usuário é feita por meio da **classe Scanner** . Esta classe fornece métodos para ler diferentes tipos de dados de entrada, como inteiros, números de ponto flutuante e strings.

Para usar a Scanner classe, primeiro você precisa criar uma instância dela e associá-la ao fluxo de entrada padrão System.in. Veja como fazer:

Scanner scanner = new Scanner([System.in](http://system.in));

Depois de obter um Scanner objeto, você pode usar seus métodos para ler a entrada. Aqui estão alguns métodos comumente usados:

**nextInt()**: Lê um inteiro.

**nextDouble()**: Lê um duplo.

**nextLine()**: Lê uma linha de texto (string).

Aqui está um exemplo de como obter informações de um usuário:

Scanner scanner = new Scanner(System.in);

System.out.print("Enter your age: ");

int age = scanner.nextInt();

System.out.println("Your age is: " + age);

Neste exemplo, o programa solicita que o usuário insira sua idade, lê o inteiro usando **nextInt()** e, em seguida, o imprime de volta no console.

Depois de terminar de usar o Scanner, é uma boa prática fechá-lo usando **scanner.close()** para liberar os recursos associados a ele.

Loops

**For Loop**

Às vezes, ao programar, é necessário executar a mesma ou quase a mesma operação algumas vezes.

Para evitar escrever a mesma coisa repetidamente, podemos usar Loops .

O for loop tem a seguinte sintaxe:

for (initialization; condition; update) {

code

}

O initialization, conditione e update determinam qual é o start valor e qual é o end valor.

Por exemplo, faça um loop de 0 a 5 (não incluindo):

for (int i = 0; i < 5; i++) {

System.out.println(i);

}

Ele executará a instrução print 5 vezes:

0

1

2

3

4

Os loops têm muitos casos de uso. Por exemplo, vamos somar todos os números de 1 a 100:

int sum\_numbers = 0;

for (int i = 1; i <= 100; i++) {

sum\_numbers += i;

}

System.out.println(sum\_numbers);

Isso primeiro percorrerá todos os números entre 1 e 100 (incluindo 100 por causa do **<=** sinal) e somará todos eles, então imprimirá a sum\_numbers variável.

**While Loop**

Um **while loop** é diferente de um for loop. Um for loop nos permite iterar sobre um objeto específico range, enquanto um **while loop** nos permite continuar iterando enquanto uma determinada condição for atendida.

Para usar um while loop escreva:

while (condition) {

code

}

O código será executado somente se a condição for true.

Há muitos casos de uso em que a while resolveria o problema, mas o for loop não.

**Do While Loop**

**do-while** loop é semelhante ao **while** loop, mas com uma diferença fundamental: o bloco de código é executado pelo menos uma vez antes da verificação da condição. Isso significa que o corpo do loop sempre será executado na primeira vez, independentemente de a condição ser verdadeira ou falsa.

Aqui está a estrutura básica de um do-while loop:

do {

// Code to be executed

} while (condition);

A **do** palavra-chave marca o início do loop, seguida pelo bloco de código entre chaves. Após o bloco de código, a **while** palavra-chave introduz a condição. O loop continuará em execução enquanto a condição for true.

Aqui está um exemplo:

int count = 0;

do {

System.out.println("Count: " + count);

count++;

} while (count < 5);

Neste exemplo, o código dentro do **do bloco** será executado primeiro, imprimindo "Contagem: 0" e incrementando count para 1. Em seguida, a condição **count < 5** é verificada. Como é true, o loop continua. Esse processo se repete até count que se torne 5, momento em que a condição se torna false e o loop termina.

A saída deste código será:

Count: 0

Count: 1

Count: 2

Count: 3

Count: 4

**break**

A **break** instrução interrompe o loop instantaneamente quando ele é encontrado.

Por exemplo,

for (int i = 0; i < 10; i++) {

if (i == 6) {

break;

}

System.out.println(i);

}

No exemplo a seguir, o loop itera regularmente até atingir o número 6. Em seguida, o programa insere a **if** instrução e a executa **break**. Isso encerra o loop imediatamente. A saída é:

0

1

2

3

4

5

**Continue**

A **continue** instrução interrompe a iteração atual e continua para a próxima. Por exemplo:

for (int i = 3; i < 9; i++) {

if (i == 5) {

continue;

}

System.out.println(i);

}

O loop repetirá por todos os números. Ao atingir , i=5 pulará essa iteração e passará para a próxima. A saída é:

3

4

6

7

8

Observe que o número 5 não está na saída.

**Loop aninhado**

Um loop aninhado é simplesmente um loop dentro de outro loop. O loop interno completará todas as suas iterações para cada iteração do loop externo.

Uma boa analogia para isso é um relógio: para cada hora (loop externo), o ponteiro dos minutos (loop interno) deve completar seu ciclo completo de 60 minutos.

Exemplo de um loop aninhado:

for (int x = 0; x < 2; x++) {

for (int y = 0; y < 2; y++) {

System.out.println(x + " " + y);

}

}

// Isto produzirá:

// 0 0

// 0 1

// 1 0

// 1 1

O loop externo (x) é executado duas vezes e, para cada uma dessas vezes, o loop interno (y) é executado duas vezes.

**Loop infinito**

Um **loop infinito** é um loop que nunca para porque sua condição sempre é avaliada como true. Esses loops podem ocorrer com qualquer tipo de loop ( for, while, do-while), mas são mais comumente associados aos loops **while** e **do-while** porque suas condições são explicitamente declaradas.

Aqui está um exemplo de um while loop infinito:

while (true) {

System.out.println("This will print forever!");

}

Neste caso, a condição é simplesmente true, que sempre é avaliada como true. Assim, o loop será executado indefinidamente, exibindo a mensagem repetidamente.

Outra maneira de criar um loop infinito é omitindo a condição em um for loop:

for (;;) {

System.out.println("This also prints forever!");

}

Aqui, as partes de inicialização, condição e atualização do **for** loop são omitidas. Isso cria um loop que dura para sempre.

Loops infinitos também podem ocorrer involuntariamente, como quando a condição do loop é sempre true devido a um erro lógico:

int i = 0;

while (i < 10) {

System.out.println("i is: " + i);

// Missing i++ to increment i

}

Neste exemplo, a variável i nunca é incrementada, então a condição i < 10 é sempre true. Isso cria um loop infinito que continuará imprimindo "i is: 0" indefinidamente.

Loops infinitos podem ser úteis em alguns casos, como em programas de servidor que precisam continuar em execução até serem parados manualmente. No entanto, eles também podem causar travamentos ou falhas em programas se não forem tratados corretamente.

Métodos

**Declarando Métodos**

Um método é uma sequência de código que possui um nome. O propósito de um método é reutilizar um trecho de código várias vezes.

Por exemplo, veja este código:

System.out.println("Welcome to Coddy");

System.out.println("New session...");

System.out.println("Welcome to Coddy");

System.out.println("Another session...");

System.out.println("Welcome to Coddy");

Usamos o mesmo código **System.out.println("Welcome to Coddy")** repetidamente. Outro problema com esse código é que, se quiséssemos alterar a mensagem Welcome to Coddy para algo diferente, como " Welcome aboard", teríamos que alterar três linhas de código diferentes. Para resolver esse problema, usaremos métodos.

Para declarar um método, usamos a seguinte sintaxe:

access\_modifier return\_type method\_name(parameters) {

// code

}

Para nosso exemplo, criaremos um método chamado greete ele ficará assim:

public static void greet() {

System.out.println("Welcome to Coddy");

}

Para usar/chamar/executar o método, escrevemos **greet()**;:

greet();

System.out.println("New session...");

greet();

System.out.println("Another session...");

greet();

Isso resultará na mesma saída acima.

Importante! O código do método deve vir antes da sua chamada/execução.

**Parâmetros do Método**

Um argumento em um método é um valor que você passa para o método ao chamá-lo. Para adicionar argumentos a um método, escrevemos os argumentos entre parênteses ():

return\_type method\_name(data\_type arg1, data\_type arg2, ...) {

// code

}

Podemos nomear os argumentos como quisermos e escrever quantos argumentos precisarmos.

Para chamar um método e passar argumentos para ele, escrevemos:

method\_name(value1, value2, value3, ...);

Passar muitos argumentos para um método que espera menos argumentos fará com que o programa falhe

Exemplo de uso:

public static void isEven(int number) {

if (number % 2 == 0) {

System.out.println(number + " is even");

} else {

System.out.println(number + " is odd");

}

}

for (int i = 15; i < 34; i++) {

isEven(i);

}

for (int i = 153; i < 219; i++) {

isEven(i);

}

Aqui, temos um método chamado isEven que aceita um argumento chamado number e imprime se o número é par ou ímpar. Em seguida, chamamos o método duas vezes: uma vez para todos os números entre 15 e 34, e a segunda vez para todos os números entre 153 e 219.

**Tipos de retorno**

A **return** instrução em um método é usada para especificar o(s) valor(es) que o método deve produzir como saída. Por exemplo, o método a seguir produzirá 100:

public static int functionName() {

return 100;

}

Para passar o valor para uma variável, escreva:

int number = functionName();

Agora a **number** variável será mantida 100 porque foi isso que o método retornou.

Observe que o tipo de retorno do método (int neste caso) deve corresponder ao tipo de dados da variável onde você está armazenando o valor retornado.

**Sobrecarga de método**

Em Java, a **sobrecarga de métodos** permite definir vários métodos com o mesmo nome, mas com parâmetros diferentes. Isso significa que você pode ter vários métodos que executam uma função semelhante, mas operam com diferentes tipos de dados ou um número diferente de argumentos. A chave é que cada método sobrecarregado deve ter uma assinatura de método única. A assinatura do método é determinada pelo nome do método e pelo tipo e ordem de seus parâmetros.

Aqui estão as principais regras para sobrecarga de métodos:

Mesmo nome : todos os métodos sobrecarregados devem ter o mesmo nome.

Parâmetros diferentes : cada versão do método deve ter um número diferente de parâmetros, tipos diferentes de parâmetros ou ambos.

O tipo de retorno não é suficiente : métodos sobrecarregados não podem diferir apenas pelo tipo de retorno. Eles devem ter listas de parâmetros diferentes.

Aqui está um exemplo para ilustrar a sobrecarga de métodos:

public class MathUtils {

public static int add(int a, int b) {

return a + b;

}

public static double add(double a, double b) {

return a + b;

}

public static int add(int a, int b, int c) {

return a + b + c;

}

}

Neste exemplo, temos três métodos chamados **add**. O primeiro soma dois inteiros, o segundo soma dois doubles e o terceiro soma três inteiros. Java determina qual método chamar com base nos argumentos passados durante a chamada.

A sobrecarga de métodos torna o código mais legível e reutilizável. Em vez de criar nomes diferentes para métodos que realizam tarefas semelhantes (por exemplo, addInts, addDoubles, addThreeInts), você pode usar um único nome, add, e deixar o compilador escolher o método apropriado com base no contexto.

Lembre-se: a sobrecarga requer diferentes tipos ou números de parâmetros, não apenas diferentes nomes de parâmetros ou tipos de retorno.

**Métodos de Vazio**

Em Java, um **void** método é um método que não retorna nenhum valor. Quando você declara um método como **void**, indica que o método executa uma tarefa ou um conjunto de operações, mas não produz um resultado que precise ser retornado ao chamador. **void** Métodos são usados quando você deseja executar ações como imprimir uma saída, modificar estados de objetos ou executar uma sequência de instruções sem retornar um valor específico.

Aqui está a estrutura básica de um void método:

public static void methodName(parameters) {

// Code to be executed

}

Noções básicas de Arrays

**Declarando Arrays**

Um **array** é uma coleção de itens e pode conter valores de diferentes tipos, como números, strings ou até mesmo outros arrays. Os **arrays** são criados usando colchetes [] e os itens dentro do array são separados por vírgulas.

Aqui está um exemplo de como criar uma **Array** (matriz):

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

Para verificar o comprimento do array, podemos usar o **.length** campo:

int length = numbers.length;

A variável length conterá 5 porque há 5 elementos no array.

Outra maneira de criar um array usando a **new** palavra-chave seguida do tipo e tamanho do array:

int[] numbers = new int[5];

Cria uma matriz de 5 inteiros, todos inicializados em 0

**Acessando Elementos**

Em Java, usamos arrays para armazenar múltiplos valores em uma única variável. Cada valor em um array é chamado de elemento, e cada elemento possui um índice. Os índices começam em 0 até o comprimento do array menos um. Por exemplo, veja o próximo array:

char[] letters = {'a', 'b', 'c', 'd', 'e', 'f', 'g'};

O elemento a está no índice 0

O elemento b está no índice 1

...

O elemento g está no índice 6

Para acessar um elemento de um array, podemos usar seu índice entre colchetes. Por exemplo, para acessar o primeiro elemento de um array chamado letters, usamos **letters[0]**.

Aqui está um exemplo:

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

int element = numbers[2];

A variável element manterá o valor 30 porque acessa o terceiro elemento (que tem um índice de 2).

**Modificando Arrays**

Além de acessar os elementos de um array, você também pode modificá-los. Para modificar um elemento específico em um array, você pode atribuir um novo valor a ele usando seu índice.

Aqui está um exemplo:

String[] my\_array = {"apple", "banana", "cherry"};

my\_array[1] = "orange";

System.out.println(my\_array[0] + ", " + my\_array[1] + ", " + my\_array[2]);

Saída:

apple, orange, cherry

banana foi alterado para um orange

**Métodos de matriz**

Arrays são repletos de diversos métodos (funcionalidades). Para acessar um método, escreva:

Arrays.methodName(arrayName, otherParameters)

Aqui está uma lista dos métodos básicos:

* **fill(array, value)** - preenche a matriz(Array) com um valor específico.
* **toString()** - converte a matriz (Array) em uma string.
* **sort(array)** - classifica a matriz (Array) em ordem crescente.
* **equals(array1, array2)** - compara duas matrizes (Arrays) para determinar se são iguais.

Aqui está um exemplo de como usar o **fill** método com **toString**:

int[] numbers = new int[5];

Arrays.fill(numbers, 10);

System.out.println(Arrays.toString(numbers));

Isso produzirá [10, 10, 10, 10, 10].

Exemplo do sort método:

int[] numbers = {5, 2, 9, 1, 5, 6};

Arrays.sort(numbers);

System.out.println(Arrays.toString(numbers));

Isso produzirá [1, 2, 5, 5, 6, 9].

Use **System.arraycopy()** para copiar elementos de um array para outro. A sintaxe é:

System.arraycopy(sourceArray, sourceStartPosition, destinationArray, destinationStartPosition, length);

Por exemplo:

// Matriz de origem

String[] sourceArray = {"1", "2", "3", "4", "5"};

// Matriz de destino

String[] destinationArray = new String[5];

// Copia elementos de sourceArray para destinationArray

System.arraycopy(sourceArray, 0, destinationArray, 0, 5);

Matriz de destino após a cópia: 1 2 3 4 5

Iterando sobres Arrays

**Usando o For Loop**

Iteração significa percorrer os elementos um por um em uma sequência. Com arrays, podemos acessar cada elemento sistematicamente usando métodos diferentes.

A maneira mais comum de iterar por um array é usando um for loop:

String[] fruits = {"apple", "banana", "orange"};

for (int i = 0; i < fruits.length; i++) {

System.out.println(fruits[i]);

}

Saída:

apple

banana

orange

**For Loop aprimorado**

O loop aprimorado for, também conhecido como loop **for-each**, oferece uma maneira mais simples de iterar por arrays e coleções. Ele lida automaticamente com a indexação e a recuperação de elementos, tornando o código mais legível e menos sujeito a erros.

Aqui está a sintaxe básica de um for loop aprimorado:

for (data\_type element : array\_or\_collection) {

// Code to be executed for each element

}

data\_type: O tipo de elementos na matriz ou coleção.

element: Uma variável que conterá o elemento atual em cada iteração.

array\_or\_collection: A matriz ou coleção sobre a qual você deseja iterar.

Aqui está um exemplo:

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

for (int number : numbers) {

System.out.println(number);

}

Neste exemplo, o loop itera sobre o numbers array. Em cada iteração, o elemento atual é atribuído à number variável e o código dentro do loop é executado. A saída será:

1

2

3

4

5

O loop aprimorado **for** é especialmente útil quando você precisa acessar cada elemento de uma coleção sem modificar a coleção em si. Ele simplifica o código e o torna mais legível.

**Operações comuns de matriz**

Aqui estão algumas operações comuns de matriz(Array):

Encontre a soma de todos os elementos em uma matriz:

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

int sum = 0;

for (int number : numbers) {

sum += number;

}

System.out.println("Sum: " + sum);

Encontre a média dos elementos em uma matriz:

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

int sum = 0;

for (int number : numbers) {

sum += number;

}

double average = (double) sum / numbers.length;

System.out.println("Average: " + average);

Encontre o elemento máximo em uma matriz:

int[] numbers = {1, 5, 2, 9, 3};

int max = numbers[0];

for (int i = 1; i < numbers.length; i++) {

if (numbers[i] > max) {

max = numbers[i];

}

}

System.out.println("Max: " + max);

Encontre o elemento mínimo em uma matriz:

int[] numbers = {1, 5, 2, 9, 3};

int min = numbers[0];

for (int i = 1; i < numbers.length; i++) {

if (numbers[i] < min) {

min = numbers[i];

}

}

System.out.println("Min: " + min);

Operações de String

Java possui um rico conjunto de métodos de string integrados que permitem realizar diversas operações em strings. Esses métodos fornecem funcionalidades para manipular, pesquisar e transformar strings. Aqui estão alguns métodos de string comumente usados:

* **length()**: Retorna o número de caracteres na string.
* **charAt(int index)**: Retorna o caractere no índice especificado (base 0).
* **substring(int beginIndex)**: Retorna uma substring começando do índice especificado até o final da string.
* **substring(int beginIndex, int endIndex)**: Retorna uma substring do beginIndex(inclusivo) para o endIndex(exclusivo).
* **startsWith(String prefix)**: Verifica se a string começa com o prefixo especificado.
* **endsWith(String suffix)**: Verifica se a string termina com o sufixo especificado.
* **toUpperCase()**: Converte a string para maiúsculas.
* **toLowerCase()**: Converte a string para minúsculas.

Aqui está um exemplo que demonstra alguns desses métodos:

String message = "Hello, World!";

int length = message.length();

char firstChar = message.charAt(0);

String sub = message.substring(7);

String sub2 = message.substring(7, 12);

int index = message.indexOf("World");

boolean starts = message.startsWith("Hello");

String lower = message.toLowerCase();

System.out.println("Length: " + length);

System.out.println("First char: " + firstChar);

System.out.println("Substring: " + sub);

System.out.println("Substring 2: " + sub2);

System.out.println("Index of 'World': " + index);

System.out.println("Starts with 'Hello': " + starts);

System.out.println("Lowercase: " + lower);

A divisão de strings permite que você divida uma string em uma matriz, enquanto a junção permite que você combine itens da matriz em uma string.

O **.split()** método divide uma string em uma matriz de substrings com base em um delimitador.

Dividido por espaços em branco:

String text = "apple banana cherry";

String[] fruits = text.split(" ");

System.out.println(Arrays.toString(fruits));

// ["apple", "banana", "cherry"]

Dividir com delimitador específico:

String data = "john,25,new york";

String[] dataArr = data.split(",");

System.out.println(Arrays.toString(dataArr));

// ["john", "25", "new york"]

O **String.join()** método combina elementos de um iterável em uma única string.

Junção básica:

String[] words = {"Hello", "World", "Java"};

String text = String.join(" ", words);

System.out.println(text);

// "Hello World Java"

Junte com separador diferente:

String[] fruits = {"apple", "banana", "cherry"};

String line = String.join(",", fruits);

System.out.println(line);

// "apple,banana,cherry"

**Formatação de Strings**

Em Java, **String.format()** é um método poderoso usado para criar strings formatadas. Ele permite combinar texto com variáveis de forma legível e personalizável. Este método usa especificadores de formato para definir como as variáveis devem ser inseridas na string.

A sintaxe básica de **String.format()** é:

String formattedString = String.format("format\_string", arg1, arg2, ...);

format\_string: Uma string que contém texto e especificadores de formato.

arg1, arg2, ...: As variáveis a serem inseridas na format\_string.

Aqui estão alguns especificadores de formato comuns:

**%s**: Insere uma string.

**%d**: Insere um decimal (inteiro).

**%f**: Insere um número de ponto flutuante.

**%b**: Insere um valor booleano.

**%c**: Insere um caractere.

**%n**: Insere um caractere de quebra de linha.

Você também pode controlar ainda mais a formatação:

**%.2f**: Formata um número de ponto flutuante com 2 casas decimais.

**%10s**: Insere uma string alinhada à direita em um campo de 10 caracteres.

**%-10s**: Insere uma string alinhada à esquerda em um campo de 10 caracteres.

**%03d**: Insere um inteiro preenchido com zeros à esquerda até uma largura de 3 dígitos.

Veja um exemplo:

String nome = "Alice";

int idade = 30;

double preço = 19.99;

String formatado = String.format("Nome: %s, Idade: %d, Preço: %.2f", nome, idade, preço);

System.out.println(formatado);

// Saída: Nome: Alice, Idade: 30, Preço: 19,99

Neste exemplo, **%s** é substituído por nome, **%d** por idade e **%.2f** por preço, formatados com duas casas decimais.

**Comparação de strings**

Em Java, comparar strings não é tão simples quanto comparar tipos de dados primitivos, como inteiros ou caracteres. Você não pode usar o **==** operador para comparar strings quanto à igualdade, pois strings são objetos e **==** compararia seus endereços de memória, não seus conteúdos. Em vez disso, você precisa usar métodos específicos fornecidos pela **String classe** para comparar os caracteres reais dentro das strings.

Aqui estão os principais métodos para comparação de strings:

* **equals(Object other)**: Compara a string com outro objeto (geralmente outra string). Retorna true se as strings tiverem os mesmos caracteres na mesma ordem, false caso contrário. Este método diferencia maiúsculas de minúsculas.
* **equalsIgnoreCase(String other)**: Semelhante a equals, mas ignora maiúsculas e minúsculas. Retorna true se as strings tiverem os mesmos caracteres, independentemente de maiúsculas e minúsculas, false caso contrário.
* **compareTo(String other)**: Compara duas strings lexicograficamente (com base em seus valores Unicode). Retorna 0 se as strings forem iguais, um número negativo se a primeira string for menor que a segunda e um número positivo se a primeira string for maior que a segunda. Este método diferencia maiúsculas de minúsculas.
* **compareToIgnoreCase(String other)**: Semelhante a **compareTo**, mas ignora maiúsculas e minúsculas.

Aqui está um exemplo:

String str1 = "hello";

String str2 = "hello";

String str3 = "HELLO";

String str4 = "world";

boolean eq1 = str1.equals(str2); // true

boolean eq2 = str1.equals(str3); // false

boolean eq3 = str1.equalsIgnoreCase(str3); // true

int cmp1 = str1.compareTo(str2); // 0

int cmp2 = str1.compareTo(str4); // negative value

int cmp3 = str4.compareTo(str1); // positive value

int cmp4 = str1.compareToIgnoreCase(str3); // 0

Extra

**Padrão numérico**

Ao criar padrões como pirâmides, muitas vezes precisamos criar strings com caracteres repetidos. Aqui está uma técnica útil para criar essas strings:

String str = new String(new char[10]).replace("\0", "a");

str vai segurar: "aaaaaaaaaa"

Vamos explicar como isso funciona:

1. **new char[10]** cria uma matriz de 10 caracteres, cada um inicializado como '\0'(caractere nulo).
2. **new String(new char[10])** converte esta matriz em uma sequência de 10 caracteres nulos.
3. **replace("\0", "a")** substitui cada caractere nulo por 'a'

É importante escrever "\0" e não“0”.

**Método trim**

O método **trim()** em Java é utilizado para remover espaços em branco no início e no final de uma string. Espaços em branco incluem espaços, tabulações e caracteres de nova linha. O método retorna uma nova string com os espaços em branco removidos, sem modificar a string original, pois strings em Java são imutáveis.

Ex:

String textoComEspacos = " texto com espaços ";

String textoSemEspacos = textoComEspacos.trim();

System.out.println("'" + textoComEspacos + "'"); // Output: ' texto com espaços '

System.out.println("'" + textoSemEspacos + "'"); // Output: 'texto com espaços'