TÓPICO 03 - INTRODUÇÃO AO JAVA

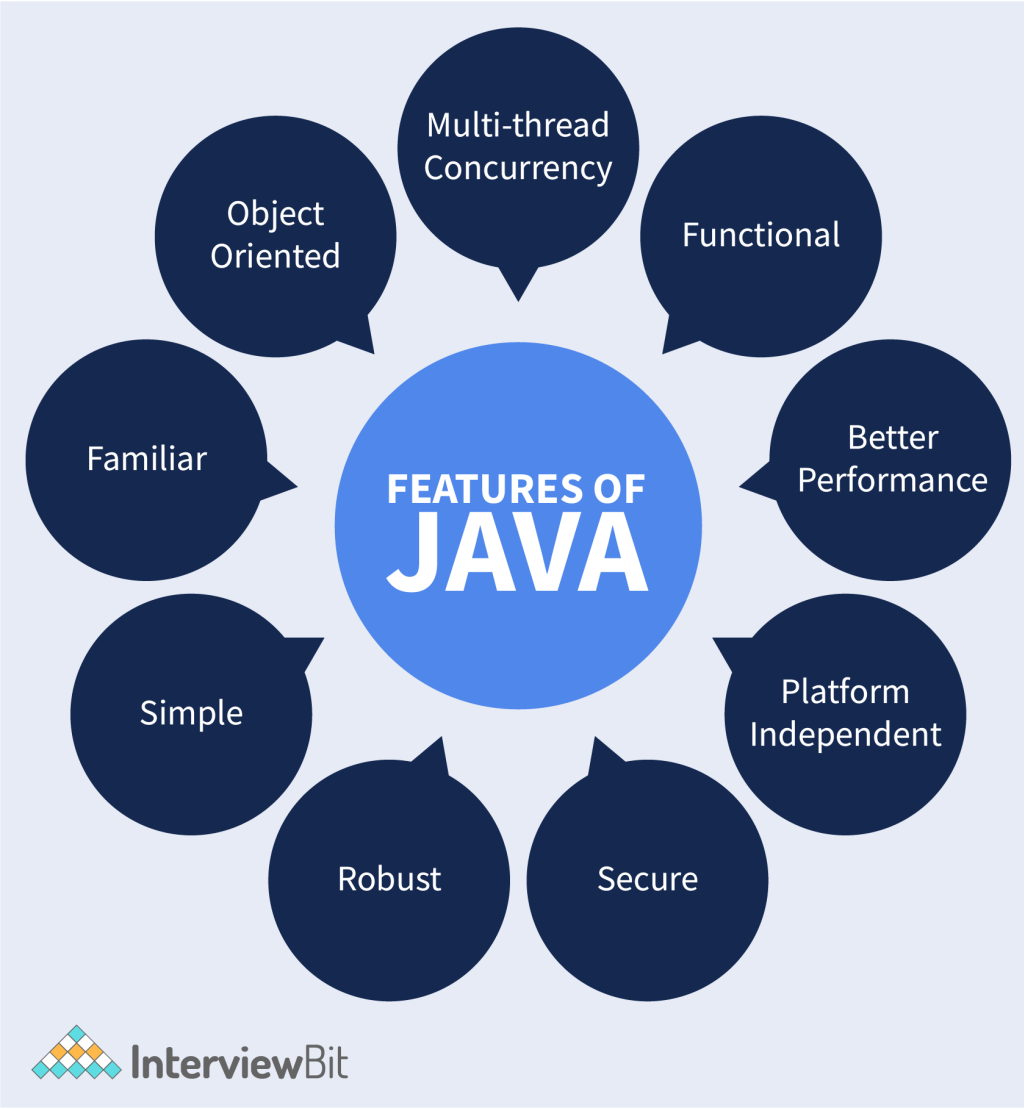
Backend - Professor Ramon Venson - SATC 2025.2

Plataforma Java

- Criada em 1995
- Desenvolvida pela ***Sun Microsystems*** até 2010
- Propriedade da ***Oracle Corporation***
- Conjunto de softwares e padrões
  - Linguagem
  - Kits de desenvolvimento
  - Máquina Virtual

Linguagem Java

- Criada por James Gosling (1991)
- Inspirada em C++
- Foi inicialmente chamada de Oak
- Foco em portabilidade, desempenho e segurança



Características da Linguagem

- Fortemente Tipada
- Baseada em classes e orientada a objetos
- Portável (escreva uma vez, rode em qualquer lugar)
- Gerenciamento automático de memória
- Ampla biblioteca padrão
- Grande comunidade e ecossistema

Linguagem Fortemente Tipada

Variáveis devem ser declaradas com o tipo de dado que será armazenado. Tipos de dados podem ser primitivos (`int` , `float` , `bool`) ou objetos (`String` , `Pessoa`).

```
int idade = 20;
String nome = "Ramon";
```

Ao contrário de linguagens fracamente tipadas (como JavaScript e Python), não é possível alterar o tipo de dado de uma variável após sua declaração. Isso ajuda a evitar erros e torna o código mais previsível.

Tipos de Dados Primitivos

Tipo	Descrição	Exemplo
`byte`	Número inteiro de 8 bits (-128 a 127)	`byte idade = 25;`
`int`	Número inteiro de 32 bits (-2^{31} a $2^{31}-1$)	`int populacao = 1000000;`
`short`	Número inteiro de 16 bits (-32.768 a 32.767)	`short ano = 2025;`
`long`	Número inteiro de 64 bits (-2^{63} a $2^{63}-1$)	`long distancia = 9460730472580800L;`

Tipo	Descrição	Exemplo
`float`	Número decimal de precisão simples (32 bits)	`float preco = 19.99f;`
`double`	Número decimal de precisão dupla (64 bits)	`double pi = 3.14159265359;`
`boolean`	Valor lógico (true ou false)	`boolean ativo = true;`
`char`	Caractere Unicode de 16 bits	`char letra = 'A';`

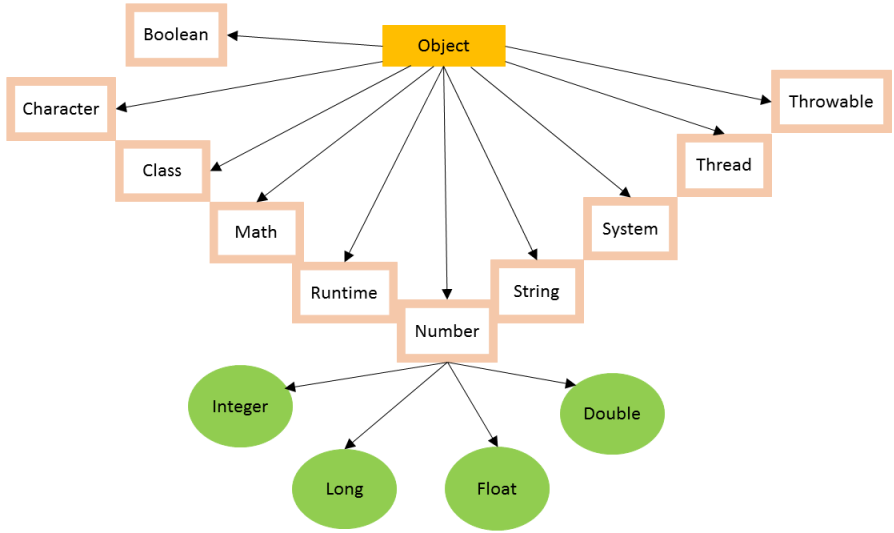
Tipos de Dados Não Primitivos

Tipos de dados não primitivos são **objetos** que podem conter múltiplos valores e métodos.

```
String nome = "Ramon";

List<String> frutas = new ArrayList<>();
frutas.add("Maçã");

Map<String, Integer> idadePorNome = new HashMap<>();
idadePorNome.put("Tim", 60);
```



Baseado em Classes

Objetos são criados a partir da estrutura das classes, usamos uma estrutura de classe com propriedades (variáveis) e métodos (funcionalidades).

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

    public void andar() {
        Mundo.mover(this, 10, 0);
    }

    public void falar(String texto) {
        System.out.println(texto);
    }
}
```

Instanciando uma Classe

Objetos são criados a partir da estrutura das classes, usamos o operador `new` para criar um novo objeto na memória.

```
Pessoa pessoa = new Pessoa();
pessoa.nome = "Ramon";
pessoa.andar();
```

Java Development Kit (JDK)

O JDK inclui:

- compilador - `javac`
- interpretador - `java`
- debug - `jdb`
- empacotamento - `jar`
- documentação - `javadoc`
- bibliotecas padrão - coleções, I/O, rede, etc.

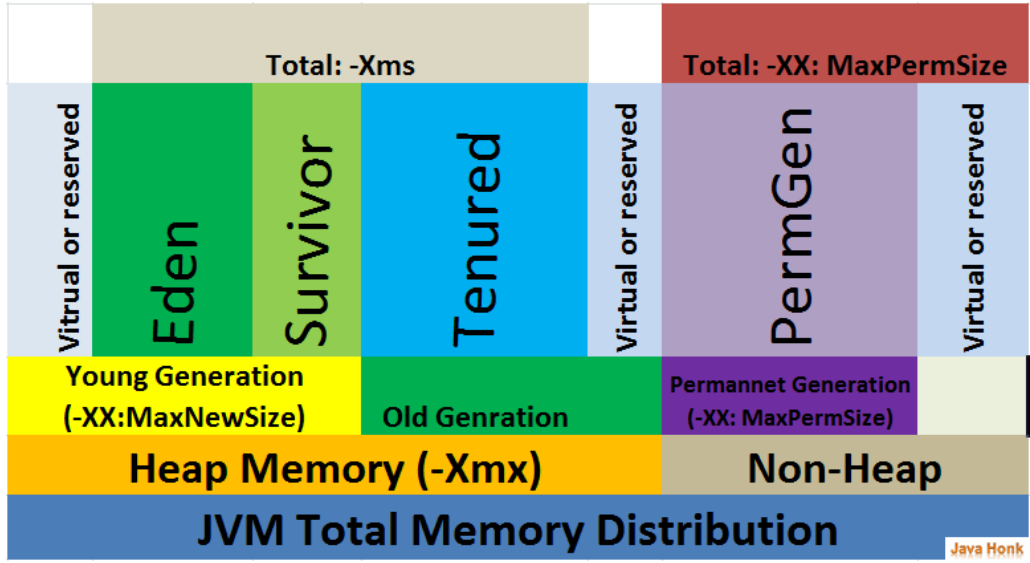
Java Virtual Machine (JVM)

- Código fonte aberto em 2007
- Roda programas compilados
- Ambiente completo é chamado de Java Runtime Environment
  - Inclui bibliotecas e a JVM
- Especificação com várias implementações:
  - **Hotspot**: desempenho robusto
  - **GraalVM**: múltiplas linguagens
  - **OpenJ9**: baixo consumo de memória e inicialização rápida

Gerenciamento de Memória

Java Heap Space

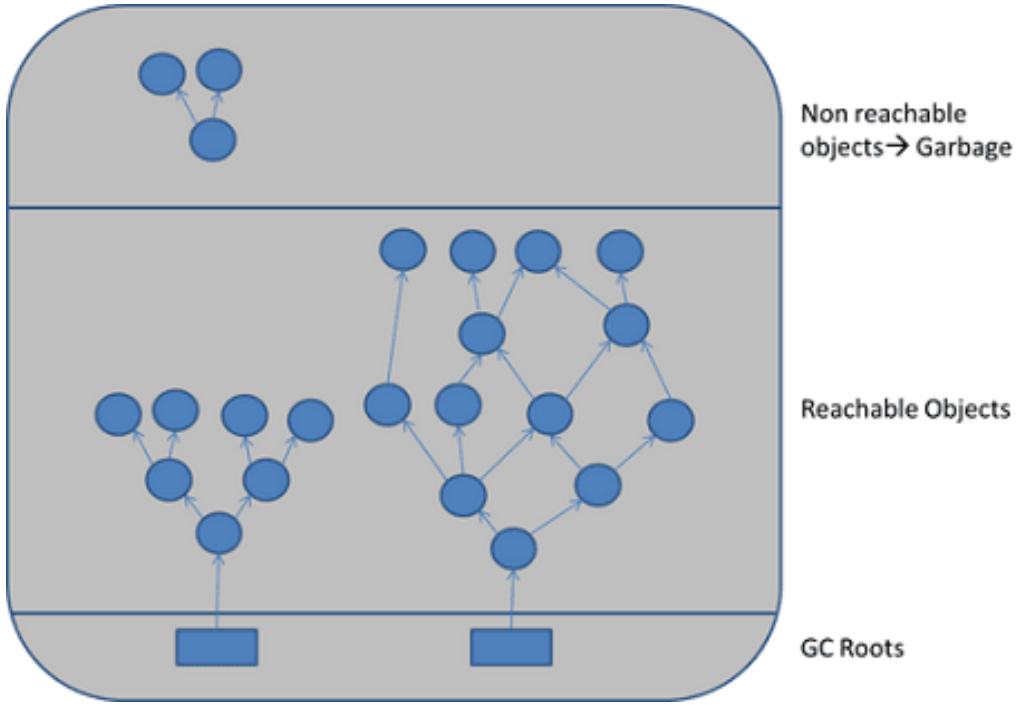
A Java Heap Space é a área de memória onde os objetos são alocados dinamicamente. É gerenciada pela JVM e é onde ocorre a coleta de lixo.



Garbage Collection

Ao invés de alocar e desalocar memória manualmente, Java utiliza um processo automático chamado **Garbage Collection**.

- Garbage Collector deleta memória não **referenciada**
- Processo automático
- Libera o programador de alocar e desalocar recursos da memória
- Não é possível forçar a coleta de lixo do ponto de vista do programador



```
Object meuObjeto = new Object(); // novo objeto é alocado na memoria
meuObjeto = null; // Objeto desalocado da memória e aguarda deleção
```

Como seria na linguagem `C`:

```
int *ptr; // Inicia um ponteiro
ptr = (int*) malloc(sizeof(int)); // aloca um ponteiro de memória do tamanho de um inteiro
free(ptr1); // libera area de memória
```

Uso de memória

tipo	estimativa	capacidade
`byte`	1 byte	$2^8 = 256$
`short`	2 bytes	$2^{16} = 65536$
`int`	4 bytes	$2^{32} = \sim 4$bilhões
`float`	4 bytes	$2^{32} = \sim 4$bilhões
`double`	8 bytes	2^{64}
`Object`	pelo menos 16 bytes (12 bytes para cabeçalho)	`Java Heap Size`

> Referências possuem geralmente 4 bytes

Sintaxe

A sintaxe do Java é inspirada em C/C++ e possui algumas particularidades:

- Sensível a maiúsculas e minúsculas
- Declaração de variáveis tipadas
- Uso de `{}` para definir blocos de código
- Uso de `;` para terminar instruções
- Comentários com `//` ou `/* ... */`
- Convenção de nomenclatura PascalCase para classes e métodos e camelCase para variáveis

Declarações

Declarações de variáveis são realizadas de maneira **tipada** (quando é necessário informar que tipo de dado vamos armazenar em uma variável)

Para declarações de variáveis dentro de classes, podemos utilizar também o modificador de acesso (`public` , `protected` , `private`). À essas variáveis damos o nome de **Atributos**.

```
public int numero;
public String nome;
protected boolean estaChovendo;
private String[] chamada;
```

Métodos

Métodos, que são como funções em outras linguagens, também possuem modificadores de acesso

```
public void imprimir() {

}

public String retornaNome(String nome) {
    return nome;
}

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

Booleanos

Representa tipos de dados que só podem assumir dois valores (`true` e `false`).
Tipicamente ocupa um byte de memória.

```
boolean estaChovendo = true;
boolean vaiChover = estaChovendo || (probabilidadeChuva > 90);
```

Utiliza-se de operadores de comparação (como `>` , `<` e `=`) operadores booleanos:

- `||` - OU
- `&&` - E
- `!` - NEGAÇÃO

Strings

Strings são sequencias de caracteres utilizados para representar texto. São representadas pelas `"` (Aspas duplas) e podem ser concatenadas (unidas) usando o operador `+` (soma).

```
String nome = "Wesley"
String sobrenome = "Safadão"
String artista = nome + " " + sobrenome;
```

Strings são imutáveis, significando que após geradas não serão modificadas.

Métodos de Strings

Assumindo que acabamos de criar uma nova string chamada `texto`, podemos utilizar os seguntes métodos:

- `texto.length` - retorna o tamanho do texto
- `texto.equals("teste")` - compara se o conteúdo de texto é igual a `teste`
- `texto.toLowerCase()` - retorna o conteúdo de texto em caixa baixa
- `texto.toUpperCase()` - retorna o conteúdo de texto em caixa alta
- `texto.replace("a", "b")` - substitui todos os caracteres `a` por `b`
- `texto.split("x")` - quebra a string em várias strings usando a letra `x`

Arrays

Arrays são estruturas capazes de armazenar múltiplos valores de um mesmo tipo sob uma mesma variável de referência.

```
// declara um novo vetor com 6 números inteiros
int[] numerosMegaSena = new int[6];
// numerosMegaSena ==> int[6] { 0, 0, 0, 0, 0, 0 }
```

Arrays em java possuem tamanho fixo definitivo ao serem instanciados. A primeira posição de um vetor sempre será 0.

```
numerosMegaSena[0]; // acessa a primeira posição do vetor anterior
numerosMegaSena[5]; // acessa a última posição do vetor anterior
numerosMegaSena[6]; // indexOutOfBounds - fora de posição
```

Também podemos utilizar as chaves para gerar um novo vetor com valores pré-definidos:

```
int[] numerosMegaSena = {4, 11, 19, 25, 33, 42};
```

Vetores de qualquer tipo de dado podem ser gerados, incluindo os não primitivos:

```
Object[] dados = {"matrix", 10, true};
```

> Os valores acima são respectivamente String, Integer e Boolean, que são, por definição, todos herdeiros da classe Object.

Percorrendo vetores

```
String[] linhas = {"Içara", "Maracajá", "Araranguá", "Cocal do Sul"};
```

```
for (int i = 0; i < linhas.length; i++) {
    System.out.println(linhas[i]);
}
```

ou

```
for (String l : linhas) {
  System.out.println(l);
}
```

Métodos de Arrays

Assumindo que acabamos de criar um novo array chamado chamado `lista`, podemos utilizar os seguntes métodos:

- `lista.length` - retorna o tamanho do vetor
- `lista.equals(lista2)` - compara se o vetor `lista` é igual ao vetor `lista2`. Vetores são iguais se possuem mesmo tamanho, valores iguais e na mesma ordem.
- `Arrays.toString(lista)` - imprime o conteúdo do vetor em forma de texto
- `Arrays.fill(dados, 1)` - preenche todos as posições do vetor com o valor
- `Arrays.sort(dados)` - Ordena o vetor por ordem numerica ou lexicográfica

Console

Para imprimir no console:

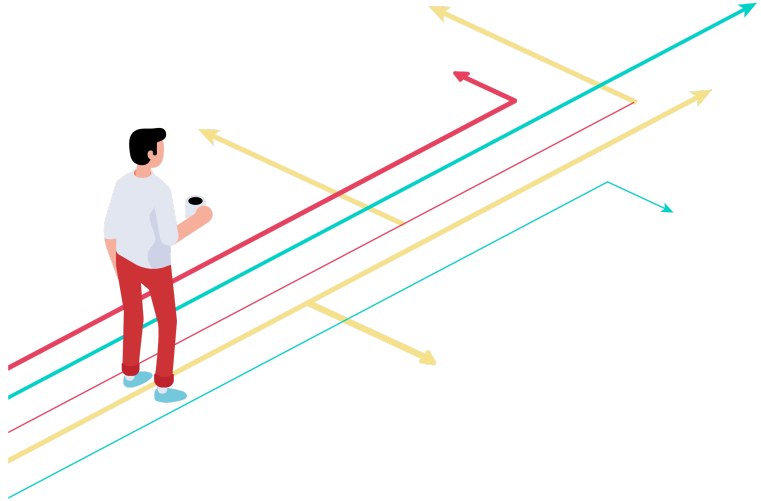
```
System.out.println("Hello World!");
```

Para ler dados do console, especialmente do que é digitado pelo usuário:

```
Scanner scanner = new Scanner(System.in);
String nome = scanner.nextLine();
```

Estrutura de Decisão

Estruturas de decisão são responsáveis por definir o fluxo de execução de um código. Essas decisões geralmente criam diferentes rotas para aplicação e devem ser pensados com cuidado pois aumentam o número de testes necessários para um código (test covering)



IF-ELSE

Utilizado para definir o fluxo do código. Testes booleanos (`true` ou `false`) são usados como condição para executar ou não um bloco de código.

```
if (condicaoBooleana) {
    executeIssoSeVerdadeiro()
}
```

```
if (condicaoBooleana) {
    executeIssoSeVerdadeiro()
} else {
    executeIssoSeFalso()
}
```

Operador Ternario

Tem como objetivo retornar um valor a partir de uma condicional, em uma única operação;

```
int valorFinal = if (condicaoBooleana) ? valorSeVerdadeiro : valorSeFalso;
```

Estrutura de Repetição

Estruturas de repetição permitem executar blocos de código diversas vezes, geralmente com parâmetros diferentes à cada iteração.

As duas principais estruturas que utilizamos (em diversas linguagens) são:

- for
- while

for tradicional

Usado geralmente para iterar sobre uma quantidade definida de operações. É composto por declaração, condição booleana e uma operação pós-loop, porém todos os valores são opcionais.

```
int quantidadeDeIteracoes = 5;
for (int i = 0; i < quantidadeDeIteracoes; i++) {
    // faça isso
}
```

for-each

Usado para iterar sobre coleções iteráveis (especialmente vetores).

```
String[] cidades = {"içara", "forquilhinha", "maracajá"};
for (String c : cidades) {
    System.out.println(c);
}
```

while

Usado geralmente para iterar quando a condição booleana deve ser manipulada de dentro do loop. Similar ao for-loop tradicional.

```
while(condicaoBooleana) {
   // repete enquanto a condição booleana for verdadeira
}
```

Imports e Packages

Packages são usados no Java para organizar diferentes conjuntos de classes. Cada classe possui seu package definido logo no início do documento.

```
// package [nome_do_pacote]
package com.organizacao.modelos;
```

Para referenciar uma classe ou um conjunto delas (packages), é preciso declarar suas importações diretamente no arquivo onde serão utilizadas.

```
// import [nome_do_pacote]
import com.organizacao.modelos.Pessoa;
import com.organizacao.servicos.*;
// ...

public static void main(String[] args) {
    Pessoa pessoa = new Pessoa();
}
```

Sem a declaração de `import` é necessário especificar o caminho completo do package de uma classe.

```
// sem imports
// ...

public static void main(String[] args) {
    com.organizacao.modelos.Pessoa pessoa = new com.organizacao.modelos.Pessoa();
}
```

Porque usar packages?

- Encapsula um conjunto de classes
- Evita conflitos de nome e garantem encapsulamento e proteção

Entradas e Saídas

O terminal é uma interface de linha de comando que permite interagir com o sistema operacional. Em Java, podemos ler e escrever no terminal usando a classe `System`.

Saída no Terminal

Para imprimir no terminal, usamos o método `println` da classe `System.out`.

```
System.out.println("Olá, mundo!");
```

Entrada do Terminal

Para ler entradas do terminal, usamos a classe `Scanner`, que permite capturar dados digitados pelo usuário.

```
import java.util.Scanner;

public class ExemploEntrada {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Digite seu nome: ");
        String nome = scanner.nextLine();
        System.out.println("Olá, " + nome + "!");
        scanner.close();
    }
}
```

Métodos do `Scanner`

O `Scanner` é uma classe que facilita a leitura de entradas do usuário. Alguns dos métodos mais comuns incluem:

- `nextLine()` - Lê uma linha inteira de texto.
- `nextInt()` - Lê um número inteiro.
- `nextDouble()` - Lê um número decimal.
- `nextBoolean()` - Lê um valor booleano (`true` ou `false`).
- `next()` - Lê a próxima palavra (até o próximo espaço).

Problemas com `Scanner`

Quebra de Linha

Ao ler uma entrada com os métodos `nextInt` ou `nextDouble`, o `Scanner` não consome o caractere de nova linha (`\n`) após a leitura, o que pode causar problemas ao ler entradas subsequentes.

```java
import java.util.Scanner;

public class ExemploScanner {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Digite um número: ");
        int numero = scanner.nextInt(); // Lê um número inteiro
        System.out.println("Você digitou: " + numero);
        // Problema: o próximo nextLine() pode ser pulado
        System.out.print("Digite seu nome: ");
        String nome = scanner.nextLine(); // Pode pular a leitura
        System.out.println("Olá, " + nome + "!");
        scanner.close();
    }
}
```

Para resolver esse problema, podemos adicionar um `scanner.nextLine()` após a leitura do número para consumir o caractere de nova linha.

Localização no `Scanner`

O `Scanner` usa, por padrão, a localização do sistema operacional para interpretar números e datas.

Ao usar o método `nextDouble()`, por exemplo, o `Scanner` espera que o separador decimal seja o mesmo usado na localização do sistema.

Em ***PT-BR***, o separador decimal é a vírgula (`,`), enquanto em EN-US é o ponto (`.`).

Referências

Na linguagem Java, objetos são manipulados por referências, que são como ponteiros para a memória onde o objeto está armazenado. Isso significa que quando você atribui um objeto a uma variável, você está na verdade atribuindo uma referência ao objeto, não o objeto em si.

```
Pessoa pessoa1 = new Pessoa();
Pessoa pessoa2 = pessoa1; // pessoa2 agora referencia o mesmo objeto que pessoa1
pessoa2.nome = "Harry Potter"; // altera o nome do objeto referenciado por pessoa1
System.out.println(pessoa1.nome); // Imprime "Harry Potter"
```

Passagem de Parâmetros

Quando passamos objetos como parâmetros para métodos, estamos passando a referência do objeto, não uma cópia dele. Isso significa que alterações feitas no objeto dentro do método afetam o objeto original.

```
public void alterarNome(Pessoa pessoa) {
    pessoa.nome = "Novo Nome";
}

public static void main(String[] args) {
    Pessoa pessoa = new Pessoa();
    pessoa.nome = "Antigo Nome";
    alterarNome(pessoa);
    System.out.println(pessoa.nome); // Imprime "Novo Nome"
}
```

Passagem de Parâmetros Primitivos

Isso não acontece com tipos primitivos, como `int`, `float`, etc. Quando passamos um tipo primitivo para um método, estamos passando uma cópia do valor, não uma referência.

```
public void alterarValor(int valor) {
    valor = 10; // altera apenas a cópia local
}

public static void main(String[] args) {
    int numero = 5;
    alterarValor(numero);
    System.out.println(numero); // Imprime 5, não 10
}
```

Comparando Objetos

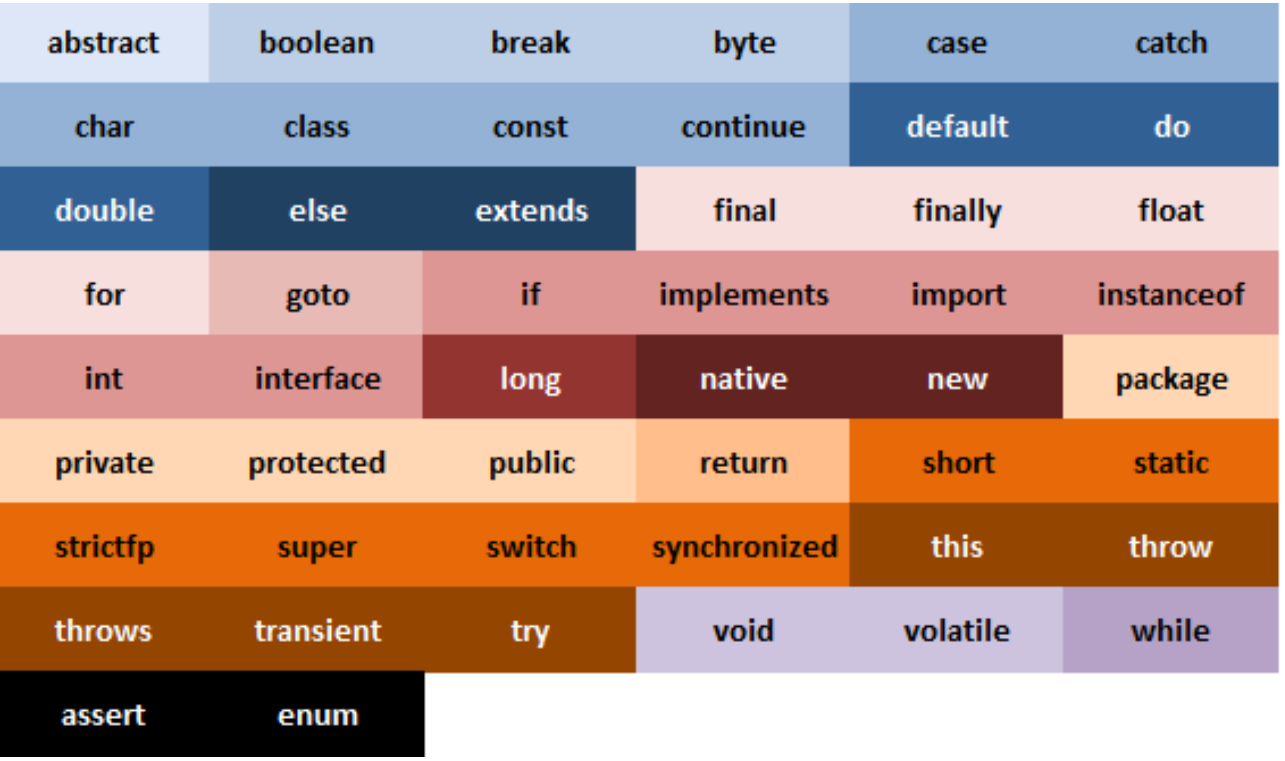
Quando comparamos objetos em Java, usamos o método `equals()` para verificar se dois objetos são iguais em termos de conteúdo. O operador `==` verifica se as referências dos objetos são iguais (ou seja, se apontam para o mesmo local na memória).

```
Pessoa pessoa1 = new Pessoa();
Pessoa pessoa2 = new Pessoa();
pessoa1.nome = "Harry";
pessoa2.nome = "Harry";
System.out.println(pessoa1 == pessoa2); // false, pois são referências diferentes
System.out.println(pessoa1.equals(pessoa2)); // false
```

Wrappers

Os tipos primitivos em Java possuem classes wrappers que permitem tratá-los como objetos. Por exemplo, `int` é envolvido pela classe `Integer`, `boolean` por `Boolean`, etc. Essas classes fornecem métodos úteis para manipulação e conversão de tipos.

```
Integer numero = Integer.valueOf(10); // Cria um objeto Integer
Boolean verdadeiro = Boolean.TRUE; // Cria um objeto Boolean
String numeroComoString = Integer.toString(numero); // Converte Integer para String
```



Palavras Reservadas

Java possui um conjunto de palavras reservadas que não podem ser usadas como identificadores (nomes de variáveis, classes, métodos, etc.). Essas palavras têm significados especiais na linguagem e são usadas para definir a estrutura e o comportamento do código.

Escopo

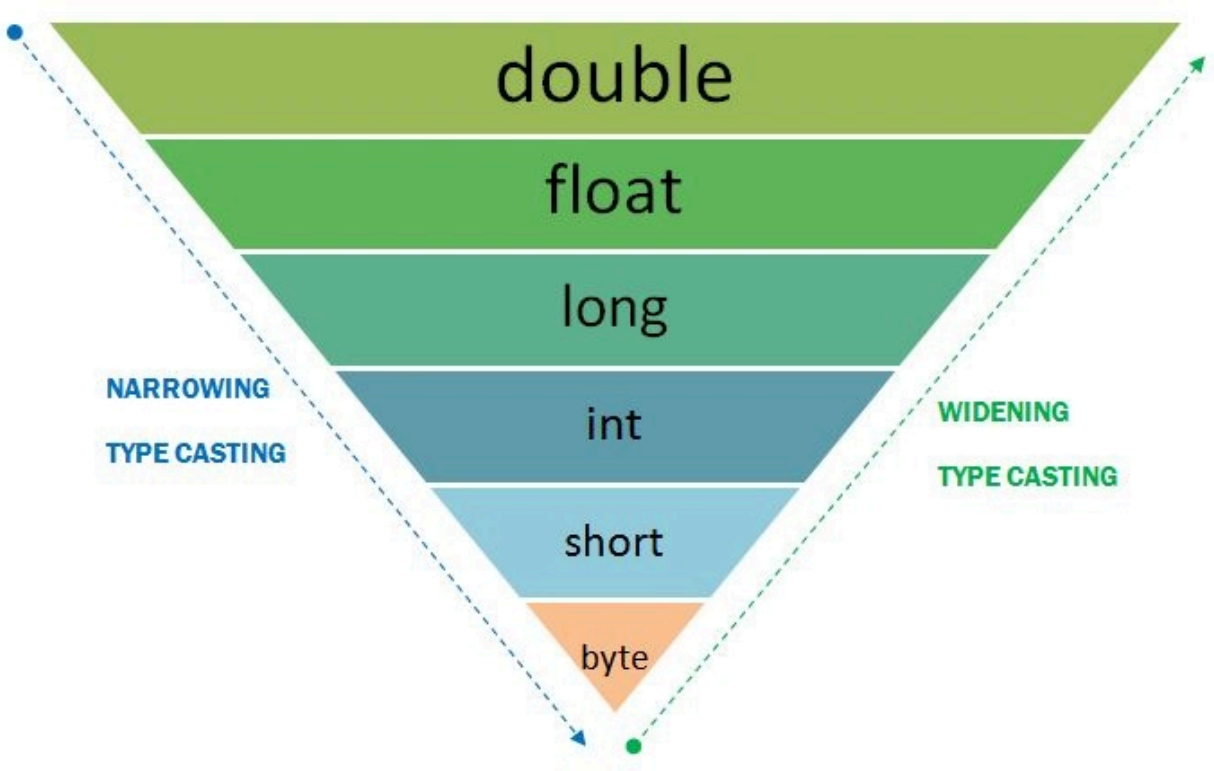
O escopo em Java refere-se à visibilidade e ao tempo de vida de variáveis, métodos e classes. Existem diferentes níveis de escopo:

- **Escopo de Classe**: Variáveis e métodos declarados dentro de uma classe, acessíveis por instâncias da classe.
- **Escopo de Método**: Variáveis declaradas dentro de um método, acessíveis apenas dentro desse método.
- **Escopo de Bloco**: Variáveis declaradas dentro de um bloco de código (como um `if`, `for`, etc.), acessíveis apenas dentro desse bloco.

```java
public class ContaBancaria {
    double saldo = 1000.0; // Escopo de Classe: saldo disponível na conta

    public void sacar(double valor) {
        double taxa = 2.5; // Escopo de Método: taxa de saque válida apenas neste método
        if (valor > 0 && valor + taxa <= saldo) { // Escopo de Bloco
            double saldoAposSaque = saldo - valor - taxa; // Variável só acessível dentro deste bloco
            System.out.println("Saque realizado! Saldo após saque: " + saldoAposSaque);
        }
        // System.out.println(saldoAposSaque); // Erro: saldoAposSaque não é acessível aqui
        System.out.println("Taxa aplicada: " + taxa);
    }

    public void consultarSaldo() {
        // System.out.println(taxa); // Erro: taxa não é acessível aqui
        System.out.println("Saldo atual: " + saldo);
    }
}
```



Type Casting

Casting é o processo de converter um tipo de dado em outro. Em Java, isso pode ser feito de forma explícita ou implícita.

Casting Implícito

Quando o tipo de dado é convertido automaticamente pelo compilador, geralmente de um tipo menor para um maior (por exemplo, de `int` para `long`).

```
int numeroInt = 10;
long numeroLong = numeroInt; // Casting implícito
```

Casting Explícito

Quando o programador especifica a conversão de um tipo de dado para outro, geralmente de um tipo maior para um menor (por exemplo, de `double` para `int`). Isso pode resultar em perda de dados.

```
double numeroDouble = 10.5;
int numeroInt = (int) numeroDouble; // Casting explícito
```

O que aprendemos hoje

- O que é a plataforma Java e seus componentes
- A sintaxe básica para a linguagem java
- Estruturas da linguagem e seu funcionamento