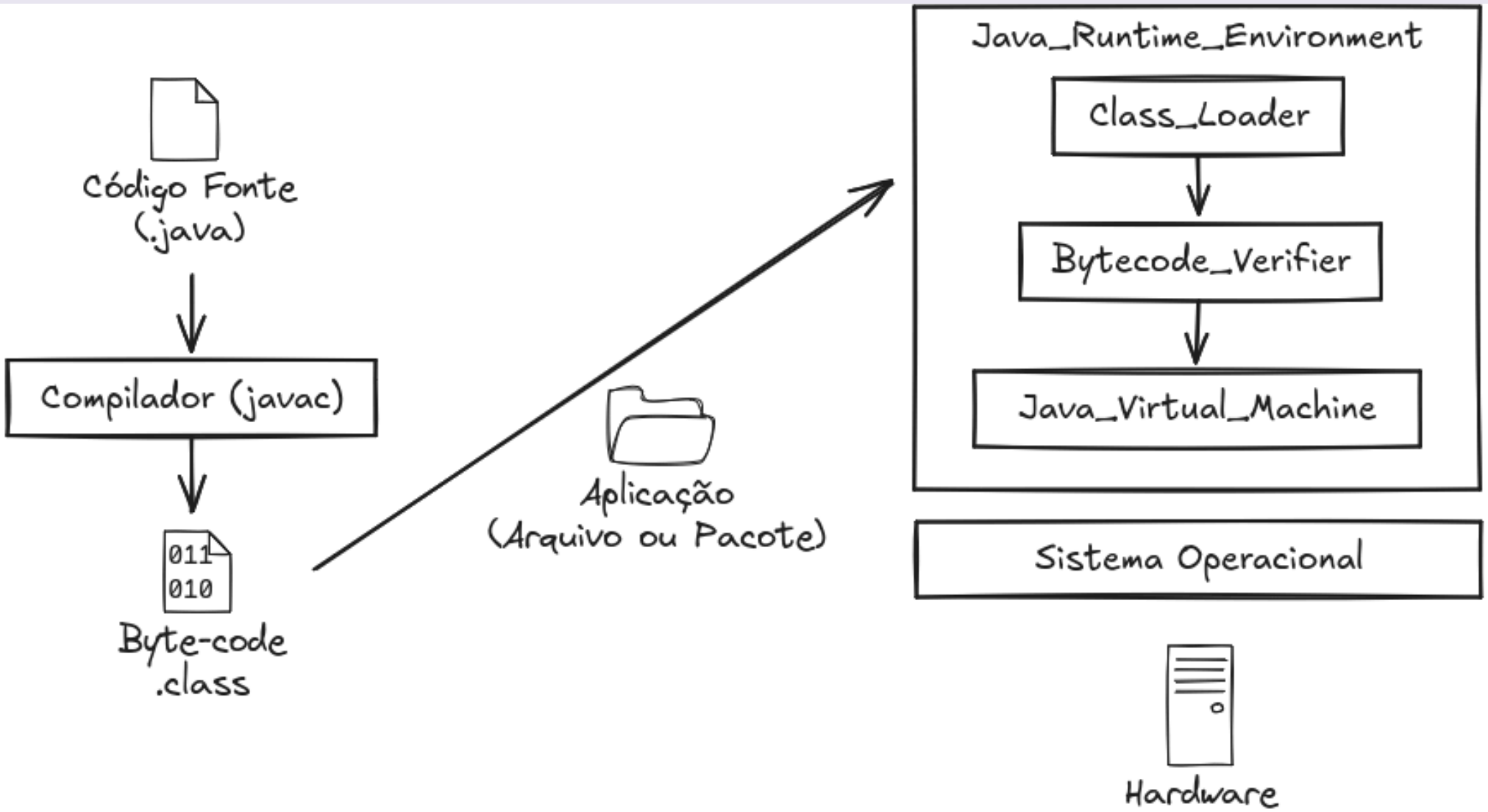
# TÓPICO 03 - INTRODUÇÃO AO JAVA

Backend - Professor Ramon Venson - SATC 2026.1

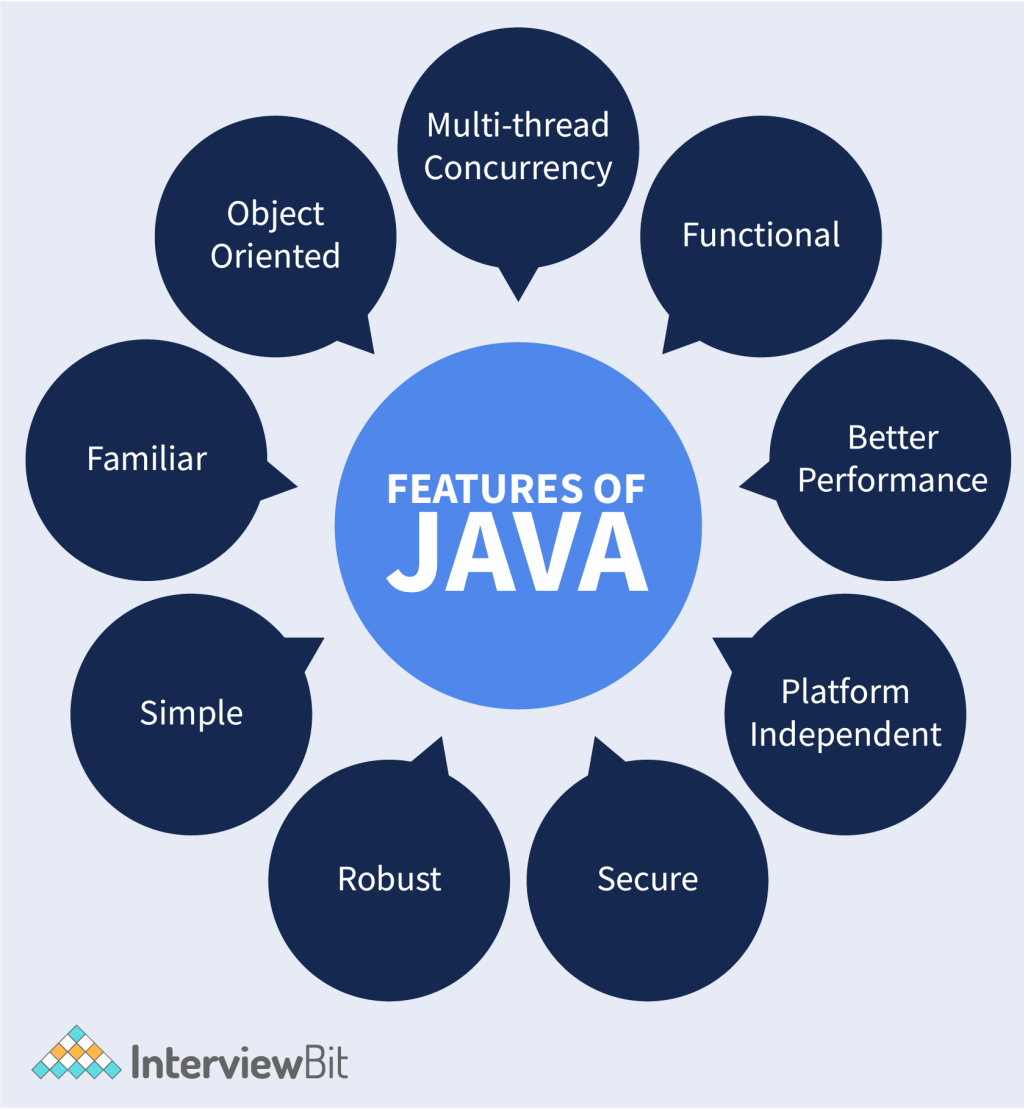

# Plataforma Java

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


Código Fonte
(.java)
Compilador (javac)
011
010
Byte-code
.class
Aplicação
(Arquivo ou Pacote)
Java_Runtime_Environment
Class_Loader
Bytecode_Verifier
Java_Virtual_Machine
Sistema Operacional
Hardware

# 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`, `boolean`) 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';` |

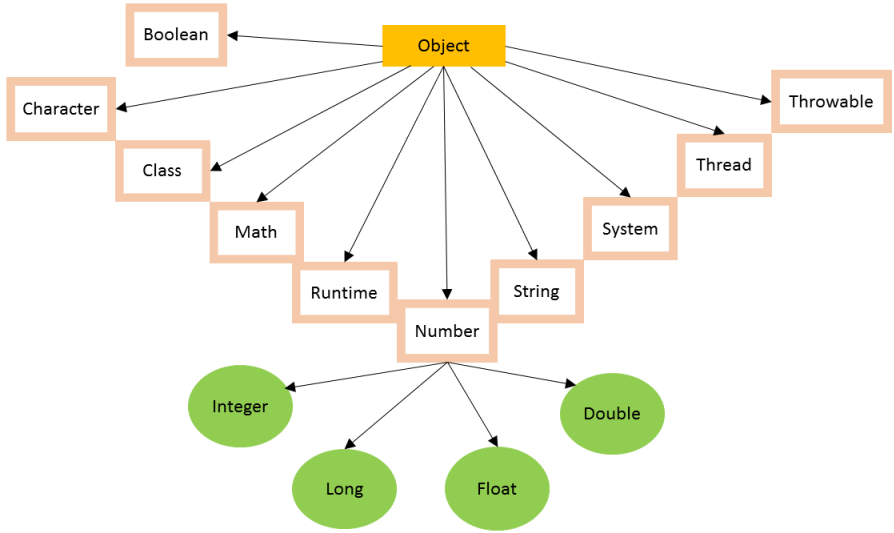
# Tipos de Dados Não Primitivos

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.

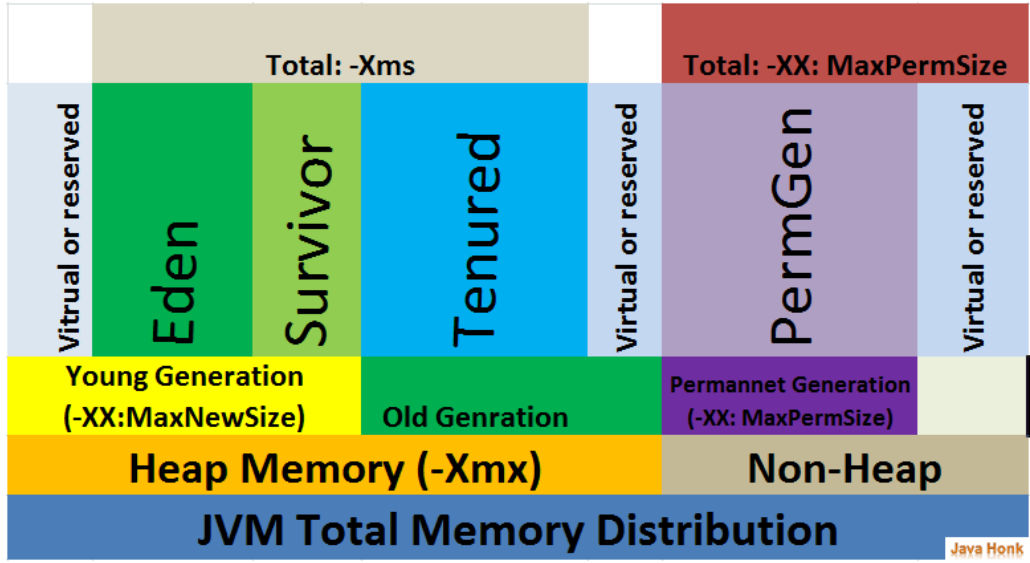
# Java Virtual Machine (JVM)


- Código-fonte aberto em 2007
- Roda programas compilados
- O 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

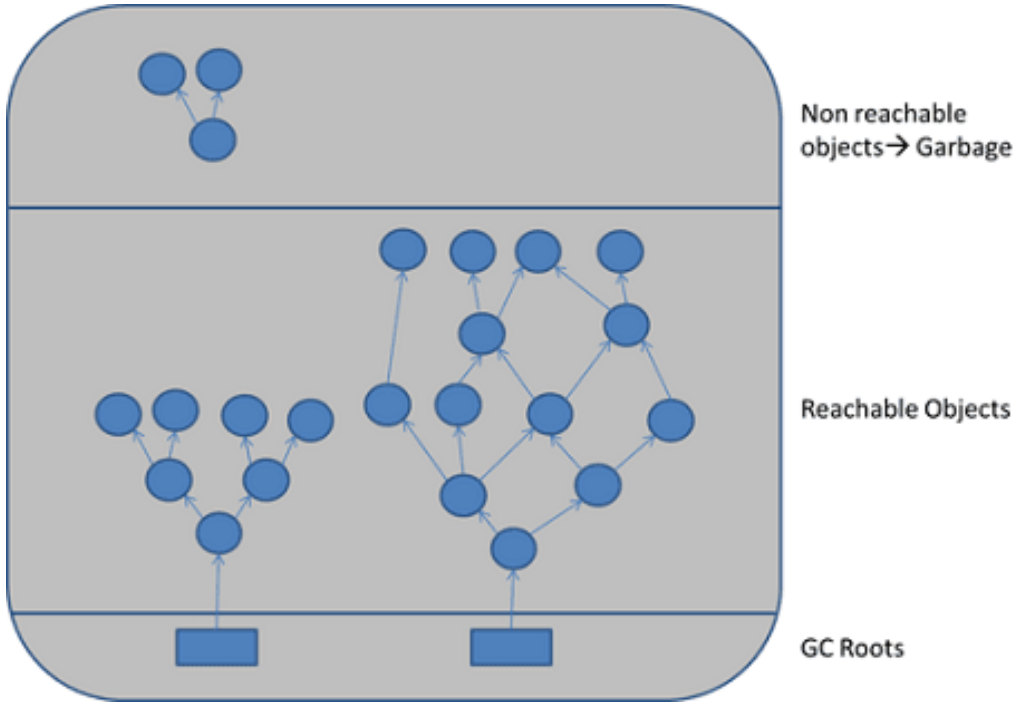
A Java Heap Space é a área de memória onde os objetos são alocados dinamicamente. É gerenciada pela JVM e é onde ocorre a coleta de lixo.



# Garbage Collection

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

- O Garbage Collector remove da memória objetos não **referenciados**
- 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



```java
Object meuObjeto = new Object(); // novo objeto é alocado na memória
meuObjeto = null; // objeto fica sem referência e aguarda deleção pelo GC
```

Como seria na linguagem `c`:

```c
int *ptr; // inicia um ponteiro
ptr = (int*) malloc(sizeof(int)); // aloca uma área de memória do tamanho de um inteiro
free(ptr); // libera área 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** (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`). A 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

Representam tipos de dados que só podem assumir dois valores (`true` e `false`).

Tipicamente ocupam um byte de memória.

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

Utilizam operadores de comparação (como `>`, `<` e `==`) e operadores booleanos:

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

# Strings

Strings são sequências de caracteres utilizadas para representar texto. São representadas por `"` (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 criadas, não serão modificadas.

## Métodos de Strings

Assumindo que acabamos de criar uma nova string chamada `texto`, podemos utilizar os seguintes 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 definido 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 `lista`, podemos utilizar os seguintes 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(lista, 1)` - preenche todas as posições do vetor com o valor `1`
- `Arrays.sort(lista)` - ordena o vetor por ordem numérica ou lexicográfica

# Console

Para imprimir no console:

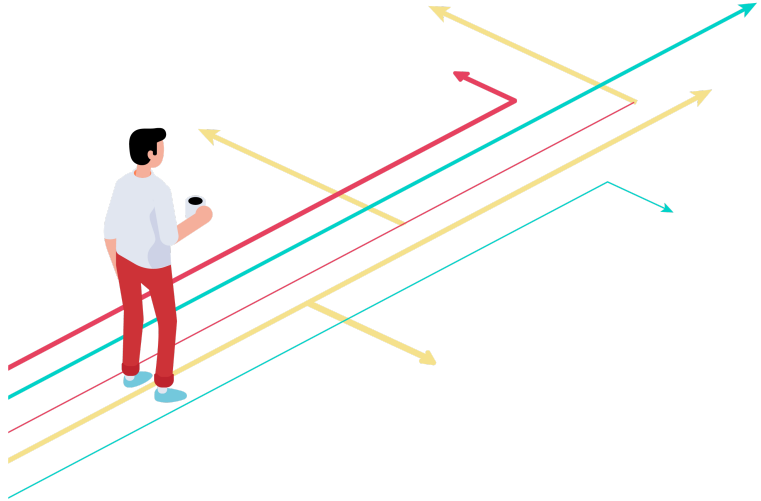
```
System.out.println("Hello World!");
```

Para ler dados do console, especialmente o 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 a aplicação e devem ser pensadas 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 Ternário

Tem como objetivo retornar um valor a partir de uma condicional, em uma única expressão:

```
int valorFinal = condicaoBooleana ? valorSeVerdadeiro : valorSeFalso;
```

# Estrutura de Repetição

Estruturas de repetição permitem executar blocos de código diversas vezes, geralmente com parâmetros diferentes a 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();
}
```

Por que usar packages?

- Encapsulam um conjunto de classes
- Evitam 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
        scanner.nextLine(); // consome a quebra de linha pendente
        System.out.print("Digite seu nome: ");
        String nome = scanner.nextLine(); // agora lê corretamente
        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"
```