

# Linguagem de programação I

## ECM251 - Linguagens de Programação I

**Professor:** Robson Calvetti

**Foco:** Programação em Java, com base na Programação Orientada a Objetos (POO)

---

### Aula 01 – Engenharia de Software, OO e POO

Aula 1 mais detalhada:

#### Engenharia de Software (ES)

É o campo da computação que trata do **projeto, desenvolvimento, manutenção e gerenciamento** de sistemas de software. O objetivo é criar software:

- Funcional
- Confiável
- Escalável
- Fácil de manter
- Que atenda aos requisitos do cliente

#### **Principais áreas da ES:**

- **Análise de Requisitos:** entender o que o software deve fazer
  - **Projeto de Software:** arquitetar o sistema (modularização, camadas, padrões)
  - **Implementação:** codificação propriamente dita
  - **Testes:** garantir que o sistema funciona como esperado
  - **Manutenção e Evolução:** correção de erros e melhorias
- 

#### Paradigmas de Programação

Um paradigma de programação é um **modelo mental para estruturar o código**.

## Exemplos:

- **Imperativo:** sequência de comandos (C, Pascal, etc.)
- **Funcional:** funções puras e recursividade (Haskell, Elixir)
- **Orientado a Objetos (OO):** modelagem com objetos (Java, C++, Python, etc.)

## Orientação a Objetos (OO)

A OO é um **paradigma que simula o mundo real**, estruturando o software em **objetos** que interagem entre si.

## Conceitos Fundamentais da OO:

Conceito	Definição
<b>Classe</b>	Molde ou projeto de um objeto (ex: Pessoa, Carro)
<b>Objeto</b>	Instância real de uma classe (ex: <code>pessoa1</code> , <code>carroAzul</code> )
<b>Atributo</b>	Características ou dados de um objeto (ex: nome, idade)
<b>Método</b>	Ações ou comportamentos (ex: <code>falar()</code> , <code>acelerar()</code> )
<b>Encapsulamento</b>	Esconder os detalhes internos de um objeto e expor apenas o necessário (via métodos <code>get/set</code> )
<b>Herança</b>	Permite que uma classe herde características de outra (ex: <code>Aluno</code> herda de <code>Pessoa</code> )
<b>Polimorfismo</b>	Capacidade de um mesmo método ter comportamentos diferentes dependendo do objeto que o invoca
<b>Abstração</b>	Foco apenas nos detalhes essenciais de um objeto, ocultando os desnecessários

## POO (Programação Orientada a Objetos)

A POO é a **implementação da orientação a objetos** em uma linguagem de programação (ex: Java). O foco principal da POO é **modularizar o software** com base em objetos e facilitar a reutilização, manutenção e legibilidade do código.

## Vantagens da POO:

- Organização e modularidade

- Reutilização de código
- Facilidade de manutenção
- Redução de erros
- Melhor compreensão de sistemas complexos

## Exemplo Básico em Java:

```
java
CopiarEditar
public class Pessoa {
    // Atributos (estado)
    private String nome;
    private int idade;

    // Construtor
    public Pessoa(String nome, int idade) {
        this.nome = nome;
        this.idade = idade;
    }

    // Método (comportamento)
    public void apresentar() {
        System.out.println("Olá, meu nome é " + nome + " e tenho " + idade + " ano
s.");
    }
}
```

## Utilizando a classe:

```
java
CopiarEditar
public class Main {
    public static void main(String[] args) {
```

```
Pessoa p1 = new Pessoa("Felipe", 21);  
p1.apresentar(); // saída: Olá, meu nome é Felipe e tenho 21 anos.  
}  
}
```

### Dica:

POO ≠ apenas usar "classe" e "método". É pensar em modularidade, reutilização e abstração.

## Aula 02 – Conceitos Básicos de Programação

### Aula 2 detalhada

#### **Conceitos Fundamentais**

- **Algoritmo:**

Um **algoritmo** é uma **sequência lógica e finita de passos** usada para resolver um problema. Ele pode ser representado em linguagem natural, pseudocódigo ou fluxogramas.

Exemplo simples:

1. Ler dois números
2. Somar os dois números
3. Mostrar o resultado

- **Programa:**

É a **implementação de um algoritmo** em uma linguagem de programação (como Java). O programa é interpretado ou compilado para que o computador possa executar.

- **Lógica de Programação:**

É a **base do raciocínio computacional**, que envolve o uso correto de sequências, decisões (condições) e repetições (laços).

- **Sistema Computacional:**

Conjunto formado por **hardware + software + usuário**, onde o programa será executado.

---

## Estruturas Básicas de Algoritmos

1. **Sequência:** execução de instruções em ordem.
  2. **Decisão (Condicional):** permite caminhos diferentes com base em condições (ex: `if/else`).
  3. **Repetição (Laço):** permite repetir ações enquanto uma condição for verdadeira ( `for` , `while` ).
- 

## Entrada e Saída em Java

### Entrada com `Scanner` :

```
java
CopiarEditar
import java.util.Scanner;

Scanner sc = new Scanner(System.in);
int x = sc.nextInt(); // Lê um número inteiro digitado pelo usuário
```

### Saída com `System.out.println` :

```
java
CopiarEditar
System.out.println("Valor: " + x); // Imprime o valor de x na tela
```

## ✓ Boas práticas:

- Sempre **importar o Scanner**: `import java.util.Scanner;`
- **Feche o Scanner** com `sc.close();` ao final do uso (exceto quando o Scanner for usado ao longo de todo o programa, como em sistemas interativos).

## ⚙️ Etapas para desenvolver um programa

1. Compreender o problema
2. Planejar o algoritmo
3. Escolher a linguagem de programação
4. Codificar (escrever o programa)
5. Testar e corrigir
6. Validar com base nos requisitos
7. Documentar e entregar

## ✏️ Representações de Algoritmos

- **Narrativa**: Texto descritivo dos passos
- **Pseudocódigo**: Quase-código, estruturado mas sem regras formais de sintaxe
- **Fluxograma**: Representação gráfica com símbolos

## 🎲 Aula 03 – Variáveis e Operadores

Aula 3 mais detalhada

### abc Tipos Primitivos em Java

Tipo	Descrição	Exemplo
<code>int</code>	Números inteiros	<code>int idade = 20;</code>
<code>double</code>	Números com casas decimais	<code>double pi = 3.14;</code>
<code>float</code>	Similar ao <code>double</code> , menor precisão	<code>float f = 2.5f;</code>

<code>char</code>	Caracteres únicos	<code>char letra = 'A';</code>
<code>boolean</code>	Lógico (verdadeiro/falso)	<code>boolean ativo = true;</code>
<code>long</code>	Inteiros maiores	<code>long l = 1000000L;</code>
<code>byte</code>	Inteiro pequeno (8 bits)	<code>byte b = 127;</code>
<code>short</code>	Inteiro pequeno (16 bits)	<code>short s = 32000;</code>

## Operadores do java



### Leitura com `Scanner`

```
import java.util.Scanner;

Scanner sc = new Scanner(System.in);
int idade = sc.nextInt();
String nome = sc.nextLine(); // Para ler texto com espaço
double altura = sc.nextDouble();
sc.close();
```



Dica: Sempre feche o Scanner com `sc.close()`; quando não for mais usá-lo.



### Trabalhando com `String`

- Strings são objetos, não tipos primitivos!
- **Criação:** `String nome = "Felipe";`



### Métodos úteis de `String`:

Método	Descrição	Exemplo
<code>nome.length()</code>	Retorna o número de caracteres	<code>nome.length()</code> → 6
<code>nome.toUpperCase()</code>	Converte para maiúsculas	<code>"felipe".toUpperCase()</code> → "FELIPE"

<code>nome.toLowerCase()</code>	Converte para minúsculas	<code>"FELIPE".toLowerCase()</code> → <code>"felipe"</code>
<code>nome.charAt(0)</code>	Retorna o caractere da posição indicada	<code>nome.charAt(0)</code> → <code>'F'</code>
<code>nome.contains("li")</code>	Verifica se contém uma substring	<code>nome.contains("li")</code> → <code>true</code>
<code>nome.substring(i, j)</code>	Retorna a parte da string entre as posições <code>i</code> e <code>j</code> (exclusivo)	<code>"Felipe".substring(1, 4)</code> → <code>"eli"</code>
<code>nome.substring(i)</code>	Retorna da posição <code>i</code> até o final	<code>"Felipe".substring(3)</code> → <code>"ipe"</code>

⚠ Índices começam em 0! `substring(0, 3)` retorna os 3 primeiros caracteres.

## Classe **Math** (biblioteca matemática)

Método	Descrição	Exemplo
<code>Math.sqrt(x)</code>	Raiz quadrada	<code>Math.sqrt(16)</code> → 4.0
<code>Math.pow(a, b)</code>	Potência ( $a^b$ )	<code>Math.pow(2, 3)</code> → 8.0
<code>Math.abs(x)</code>	Valor absoluto	<code>Math.abs(-5)</code> → 5
<code>Math.max(a, b)</code>	Maior entre dois valores	<code>Math.max(3, 7)</code> → 7
<code>Math.min(a, b)</code>	Menor entre dois valores	<code>Math.min(3, 7)</code> → 3
<code>Math.round(x)</code>	Arredonda para inteiro mais próximo	<code>Math.round(2.8)</code> → 3
<code>Math.floor(x)</code>	Arredonda para baixo (menor inteiro)	<code>Math.floor(2.8)</code> → 2
<code>Math.ceil(x)</code>	Arredonda para cima (maior inteiro)	<code>Math.ceil(2.1)</code> → 3
<code>Math.random()</code>	Retorna um valor entre 0 e 1	<code>Math.random()</code> → 0.0–1.0

## **System.out** – Impressão no Console

### Principais métodos de saída:

Método	Descrição	Exemplo
<code>System.out.print()</code>	Imprime sem pular linha	<code>System.out.print("Olá");</code>



<code>System.out.println()</code>	Imprime com quebra de linha	<code>System.out.println("Olá");</code>
<code>System.out.printf()</code>	Imprime com formatação (placeholders)	<code>System.out.printf("Valor: %d", 10);</code>



Você pode combinar vários especificadores:

```
java
CopiarEditar
String nome = "Ana";
int idade = 22;
System.out.printf("Nome: %s, Idade: %d\n", nome, idade);
```



## Aula 04 – Desvios Condicionais



### Operadores de Comparação (Relacionais)

Utilizados em expressões lógicas para tomar decisões. Sempre retornam um valor

`boolean` ( `true` ou `false` ).

Operador	Significado	Exemplo ( <code>a = 5</code> , <code>b = 10</code> )	Resultado
<code>==</code>	Igual a	<code>a == b</code>	<code>false</code>
<code>!=</code>	Diferente de	<code>a != b</code>	<code>true</code>
<code>&gt;</code>	Maior que	<code>a &gt; b</code>	<code>false</code>
<code>&lt;</code>	Menor que	<code>a &lt; b</code>	<code>true</code>
<code>&gt;=</code>	Maior ou igual	<code>a &gt;= 5</code>	<code>true</code>
<code>&lt;=</code>	Menor ou igual	<code>b &lt;= 10</code>	<code>true</code>



### Estruturas de Decisão



#### If / Else / Else If:

```
if (idade >= 18) {  
    System.out.println("Maior de idade");  
} else if (idade > 12) {  
    System.out.println("Adolescente");  
} else {  
    System.out.println("Criança");  
}
```

## Operador Ternário


Forma compacta do `if/else`, usado para atribuição rápida ou lógica simples.

```
variável = (condição) ? valor_se_verdadeiro : valor_se_falso;
```

### Exemplo:

```
int idade = 20;  
String status = (idade >= 18) ? "Adulto" : "Menor";  
System.out.println(status); // Adulto
```

## Comparação de Strings

 **Nunca use `==` para comparar `Strings`!** Ele compara referências (endereço de memória), não o conteúdo.

 **Use `equals()` :**

```
String nome = "Felipe";  
  
if (nome.equals("Felipe")) {  
    System.out.println("Nome correto!");  
}
```

 **Ignorar maiúsculas/minúsculas:**

```
if (nome.equalsIgnoreCase("felipe")) {  
    System.out.println("Nome válido!");  
}  
if ("12345".equals("12345")) {  
    System.out.println("Número válido");  
}
```

### Dica:

Sempre que precisar verificar igualdade de valores literais ( `String` , `char` , etc), use:

- `equals()` → igual
- `equalsIgnoreCase()` → igual ignorando maiúsculas
- `!nome.equals("algo")` → diferente

## Aula 05 – Laços de Repetição (Loops)

### O que são laços de repetição?

Laços de repetição (ou loops) permitem **executar um mesmo trecho de código várias vezes**, de forma automática, sem precisar repetir o código manualmente.

### Vantagens:

- Evita repetição de código
- Automatiza tarefas
- Permite trabalhar com dados em sequência (ex: vetores, listas)

### Tipos de Laços em Java

#### 1. `while` – Laço com teste no início

Executa o bloco **enquanto a condição for verdadeira**. Se a condição for falsa na primeira verificação, o bloco **nunca será executado**.

```
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;
}
```

#### Usar quando:

→ O número de repetições **não é conhecido** com antecedência.

---

## 2. **do...while** – Laço com teste no final

Executa o bloco **pelo menos uma vez**, e depois verifica a condição.

```
int i = 0;
do {
    System.out.println(i);
    i++;
} while (i < 5);
```

#### Usar quando:

→ É necessário **executar o bloco pelo menos uma vez**, mesmo sem saber se a condição será verdadeira.

---

## 3. **for** – Laço com controle explícito

Ideal quando se sabe o número exato de vezes que algo deve ser repetido.

```
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

#### Usar quando:

→ O número de repetições é **conhecido ou controlado por um contador**.

---

### Diferença entre os laços

Tipo	Testa antes?	Executa pelo menos uma vez?	Ideal para...
<code>while</code>	✅ Sim	❌ Não	Quando a condição pode ser falsa no início
<code>do-while</code>	❌ Não	✅ Sim	Quando precisa executar ao menos uma vez
<code>for</code>	✅ Sim	❌ Não	Quando se sabe quantas vezes deve repetir

## Exemplo prático: soma de 1 a 100

```
int soma = 0;
for (int i = 1; i <= 100; i++) {
    soma += i;
}
System.out.println("Soma: " + soma); // Resultado: 5050
```

## Laço Infinito (cuidado!)

Um loop sem condição de parada trava o programa.

```
while (true) {
    System.out.println("Loop infinito...");
}
```

**! Sempre garanta que a condição do laço vai se tornar falsa em algum momento.**

## Leitura de dados com laço ( `while` + `Scanner` )

```
Scanner sc = new Scanner(System.in);
int num = 0;

while (num != -1) {
    System.out.print("Digite um número (-1 para sair): ");
```

```
num = sc.nextInt();  
}
```

## Contador e Acumulador

- **Contador** → incrementa em cada repetição ( `i++` )
- **Acumulador** → soma valores ao longo do tempo ( `soma += valor` )

```
int contador = 0;  
int acumulador = 0;  
  
while (contador <= 5) {  
    acumulador += contador;  
    contador++;  
}
```

## Boas práticas

- Sempre defina **condições claras de parada**.
- Cuidado com laços infinitos não intencionais.
- Prefira `for` quando usar **contadores**.
- Use nomes descritivos para variáveis ( `i` , `j` , `contador` , `soma` ).
- **Inicialize** variáveis antes do laço.

## Dica extra: `break` e `continue`

- `break` : encerra o laço imediatamente.
- `continue` : pula a iteração atual e vai para a próxima.

```
for (int i = 1; i <= 10; i++) {  
    if (i == 5) continue; // pula o número 5  
    if (i == 8) break;    // para quando chegar em 8  
}
```

```
System.out.println(i);  
}
```

## Aula 06 – Conceitos de Orientação a Objetos

[Aula 6 mais detalhada](#)

### O que é a Programação Orientada a Objetos (POO)?

POO é um **paradigma de programação** que modela o mundo real usando **objetos**, que possuem **estado** (atributos) e **comportamentos** (métodos).

O foco está em **organizar o código em torno de entidades lógicas**, promovendo reuso, clareza e manutenção.

## Conceitos Fundamentais

### Classe

- É uma **estrutura (molde)** que define atributos (variáveis) e métodos (funções) de um objeto.

```
public class Pessoa {  
    String nome;  
    int idade;  
  
    void apresentar() {  
        System.out.println("Olá, meu nome é " + nome);  
    }  
}
```

### Objeto

- Uma **instância** de uma classe, ou seja, a classe em uso real.
- Cada objeto tem seu **próprio estado (atributos com valores diferentes)**.

```
Pessoa p1 = new Pessoa();  
p1.nome = "Felipe";  
p1.idade = 21;  
p1.apresentar();
```

## Estado da Classe

- O **conjunto de valores atuais dos atributos** de um objeto representa seu **estado**.
- Esse estado pode **mudar** com o tempo, à medida que métodos são executados.

## Instanciação de Objetos

- Feita usando a palavra-chave `new`.
- Chama o **construtor** da classe para criar o objeto.

```
Pessoa p2 = new Pessoa(); // instanciando a classe Pessoa
```

## Assinatura de um Método

- Conjunto que define **nome, parâmetros e tipo de retorno** do método.

Exemplo de assinatura:

```
public int somar(int a, int b)
```

- Nome: `somar`
- Parâmetros: `(int a, int b)`
- Tipo de retorno: `int`

## Tipos de Métodos



## ✓ 1. Métodos de Instância

- Chamados em um objeto da classe.
- Podem acessar os atributos e outros métodos da **mesma instância**.

```
public void apresentar() {  
    System.out.println("Olá, eu sou " + nome);  
}
```

## ↺ 2. Métodos Estáticos ( **static** )

- Pertencem à **classe**, não ao objeto.
- Usados sem precisar instanciar a classe.

```
public static double calcularPI() {  
    return 3.1415;  
}
```

```
double pi = MinhaClasse.calcularPI();
```

## 🏗️ 3. Construtores

- São métodos especiais para **criar e inicializar objetos**.
- Têm o **mesmo nome da classe** e **não têm tipo de retorno**.

```
public Pessoa(String nome) {  
    this.nome = nome;  
}
```

## 🔒 4. Getters e Setters (Encapsulamento)

```
private int idade;
```

```
public int getIdade() {  
    return idade;  
}  
  
public void setIdade(int idade) {  
    this.idade = idade;  
}
```

## 5. Métodos com ou sem retorno

- `void` → não retorna nada
- `int`, `double`, `String` → retornam algum valor

```
public void imprimir() {  
    System.out.println("Texto");  
}  
  
public int soma(int a, int b) {  
    return a + b;  
}
```

## Princípios da POO

### 1. Abstração

- Foca nos **detalhes importantes** e esconde a complexidade.

### 2. Encapsulamento

- **Restringe o acesso direto** aos dados internos e os protege com `getters` e `setters`.

### 3. Herança

- Uma classe pode **herdar** de outra para reutilizar comportamentos.

```
class Animal {
    void dormir() {
        System.out.println("Dormindo...");
    }
}

class Cachorro extends Animal {
    void latir() {
        System.out.println("Au au!");
    }
}
```

## 4. Polimorfismo

- Permite que um **mesmo método** tenha **diferentes comportamentos**, dependendo da classe que o implementa.

```
class Animal {
    void emitirSom() {
        System.out.println("Som genérico");
    }
}

class Gato extends Animal {
    void emitirSom() {
        System.out.println("Miau");
    }
}
```