

Linguagens e paradigmas orientado a objetos

Análise e Desenvolvimento de Sistemas

Cronograma

1. Introdução à Programação Orientada a Objetos

- Definição e importância da POO.
- Comparação entre programação procedural e POO.

2. Princípios Fundamentais da POO

- Encapsulamento
- Herança
- Polimorfismo
- Abstração

3. Classes e Objeto

- Definição de classe.
- Criação e manipulação de objetos.

4. Sintaxe Básica em Java

- Método MAIN
- Funções
- Tipo de variáveis
- Declaração de variáveis
- Estrutura de decisão
- Estrutura de Repetição

Introdução à Programação Orientada a Objetos - Definição de POO

"A programação orientada a objetos é uma metodologia de design e desenvolvimento de software onde os sistemas são modelados como coleções de objetos que interagem entre si.

Cada objeto representa uma entidade que possui um estado (dados) e um comportamento (operações)." (Grady Booch)

Programação Orientada a Objetos, ou POO, é uma maneira de organizar o código de um programa de forma que ele seja mais fácil de entender e manter, ela utiliza "objetos" e suas interações para projetar e programar aplicações.

Introdução à Programação Orientada a Objetos - Importância da POO

Organização e Estrutura do Código

Reutilização de Código / Modularidade

Facilidade de Manutenção e Extensibilidade

Escalabilidade

Facilidade de Entendimento / Modelagem do Mundo Real

Princípios Fundamentais da POO

Encapsulamento

Encapsulamento é o princípio que consiste em esconder os detalhes internos de um objeto e expor apenas o que é necessário. Isso é feito através do uso de modificadores de acesso, como **private**, **protected**, e **public**.

```
public class Pessoa {  
    private String nome; // Atributo privado  
    private int idade;   // Atributo privado  
  
    // Método público para acessar o atributo nome  
    public String getNome() {  
        return nome;  
    }  
  
    // Método público para modificar o atributo nome  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    // Método público para acessar o atributo idade  
    public int getIdade() {  
        return idade;  
    }  
  
    // Método público para modificar o atributo idade  
    public void setIdade(int idade) {  
        if (idade > 0) { // Validação simples  
            this.idade = idade;  
        }  
    }  
}
```

Herança

Herança é o princípio que permite criar novas classes a partir de classes existentes, reutilizando código e adicionando novas funcionalidades.

A nova classe é chamada de **subclasse** (ou classe derivada), e a classe existente é chamada de **superclasse** (ou classe base).

```
// Superclasse
public class Animal {
    public void comer() {
        System.out.println("Animal está comendo");
    }
}

// Subclasse que herda de Animal
public class Cachorro extends Animal {
    public void latir() {
        System.out.println("Cachorro está latindo");
    }

    // Sobrescrevendo o método comer da superclasse
    @Override
    public void comer() {
        System.out.println("Cachorro está comendo ração");
    }
}

// Uso da herança
public class TesteHeranca {
    public static void main(String[] args) {
        Cachorro meuCachorro = new Cachorro();
        meuCachorro.comer(); // Chama o método sobrescrito da subclasse
        meuCachorro.latir(); // Chama o método da subclasse
    }
}
```

Polimorfismo

Polimorfismo é o princípio que permite que **objetos tenham implementações diferentes** estendendo de uma **classe comum**.

Isso é particularmente útil para permitir que uma única interface manipule diferentes tipos de objetos.

```
public class Animal {
    public void fazerSom() {
        System.out.println("Animal faz som");
    }
}

public class Cachorro extends Animal {
    @Override
    public void fazerSom() {
        System.out.println("Cachorro late");
    }
}

public class Gato extends Animal {
    @Override
    public void fazerSom() {
        System.out.println("Gato mia");
    }
}

// Uso do polimorfismo
public class TestePolimorfismo {
    public static void main(String[] args) {
        Animal meuAnimal1 = new Cachorro();
        Animal meuAnimal2 = new Gato();

        meuAnimal1.fazerSom(); // Chama o método fazerSom da classe Cachorro
        meuAnimal2.fazerSom(); // Chama o método fazerSom da classe Gato
    }
}
```


Abstração

Abstração é o princípio de se **concentrar** nos aspectos essenciais de um objeto, ignorando os detalhes menos relevantes.

Isso é feito através de classes abstratas e interfaces.

```
// Classe abstrata
abstract class Forma {
    // Método abstrato (sem implementação)
    abstract void desenhar();
}

// Subclasse concreta
class Circulo extends Forma {
    @Override
    void desenhar() {
        System.out.println("Desenhar um círculo");
    }
}

// Subclasse concreta
class Quadrado extends Forma {
    @Override
    void desenhar() {
        System.out.println("Desenhar um quadrado");
    }
}

// Uso da abstração
public class TesteAbstracao {
    public static void main(String[] args) {
        Forma forma1 = new Circulo();
        Forma forma2 = new Quadrado();

        forma1.desenhar(); // Chama o método desenhar da classe Circulo
        forma2.desenhar(); // Chama o método desenhar da classe Quadrado
    }
}
```

Classes e Objetos

Definição de Classe

Uma classe é um modelo que define as características e comportamentos de um objeto.

Ela especifica os atributos (dados) e métodos (funções) que os objetos desta classe terão.

Pense em uma classe como a planta de uma casa: ela descreve como a casa será, mas não é a casa em si.

```
public class Casa {  
    // Atributos da classe Casa  
    String cor;  
    int numeroDeQuartos;  
    double area;  
  
    // Método da classe Casa  
    void exibirDetalhes() {  
        System.out.println("Cor: " + cor);  
        System.out.println("Número de Quartos: " + numeroDeQuartos);  
        System.out.println("Área: " + area + " metros quadrados");  
    }  
}
```

Definição da Classe **Casa**:

- **Atributos:** **cor**, **numeroDeQuartos**, **area** são características de uma casa.
- **Método:** **exibirDetalhes()** é uma função que exibe os valores dos atributos da casa.



```
public class Casa {  
    // Atributos da classe Casa  
    String cor;  
    int numeroDeQuartos;  
    double area;  
  
    // Método da classe Casa  
    void exibirDetalhes() {  
        System.out.println("Cor: " + cor);  
        System.out.println("Número de Quartos: " + numeroDeQuartos);  
        System.out.println("Área: " + area + " metros quadrados");  
    }  
}
```

Criação e Manipulação de Objetos

Um objeto é uma instância de uma classe.

Depois de definir uma classe, você pode criar objetos a partir dela e manipular seus atributos e métodos.

Um objeto representa uma entidade específica com seus próprios valores para os atributos definidos na classe.

```
public class TesteCasa {  
    public static void main(String[] args) {  
        // Criação de um objeto da classe Casa  
        Casa minhaCasa = new Casa();  
  
        // Manipulação dos atributos do objeto  
        minhaCasa.cor = "Azul";  
        minhaCasa.numeroDeQuartos = 3;  
        minhaCasa.area = 120.5;  
  
        // Chamada do método para exibir detalhes do objeto  
        minhaCasa.exibirDetalhes();  
    }  
}
```

Declaração do objeto **Casa**:

Criação do Objeto **minhaCasa**:



- Utilizamos a palavra-chave **new** para criar uma nova instância da classe **Casa**.
- **minhaCasa** agora é um objeto da classe **Casa** com seus próprios valores para **cor**, **numeroDeQuartos** e **area**. TODOS não possuem valores (São NULL)

```
public class TesteCasa {  
    public static void main(String[] args) {  
        // Criação de um objeto da classe Casa  
        Casa minhaCasa = new Casa();  
  
        // Manipulação dos atributos do objeto  
        minhaCasa.cor = "Azul";  
        minhaCasa.numeroDeQuartos = 3;  
        minhaCasa.area = 120.5;  
  
        // Chamada do método para exibir detalhes do objeto  
        minhaCasa.exibirDetalhes();  
    }  
}
```

Manipulação dos Atributos e Métodos

- Atribuímos valores aos atributos do objeto **minhaCasa**.
- Chamamos o método **exibirDetalhes()** para exibir os valores dos atributos.

```
public class TesteCasa {  
    public static void main(String[] args) {  
        // Criação de um objeto da classe Casa  
        Casa minhaCasa = new Casa();  
  
        // Manipulação dos atributos do objeto  
        minhaCasa.cor = "Azul";  
        minhaCasa.numeroDeQuartos = 3;  
        minhaCasa.area = 120.5;  
  
        // Chamada do método para exibir detalhes do objeto  
        minhaCasa.exibirDetalhes();  
    }  
}
```

Sintaxe Básica em Java

Método main

Método main

Método **main**

O método main é o ponto de entrada para qualquer aplicação Java.

É onde o programa começa a ser executado

```
public class MinhaClasse {  
    public static void main(String[] args) {  
        System.out.println("Olá, Mundo!"); // Imprime uma mensagem na tela  
    }  
}
```

Método main

```
public class MinhaClasse {  
    public static void main(String[] args) {  
        System.out.println("Olá, Mundo!"); // Imprime uma mensagem na tela  
    }  
}
```

public: Modificador de acesso que permite que o método seja acessível de qualquer lugar.

Método main

```
public class MinhaClasse {  
    public static void main(String[] args) {  
        System.out.println("Olá, Mundo!"); // Imprime uma mensagem na tela  
    }  
}
```

public: Modificador de acesso que permite que o método seja acessível de qualquer lugar.

Método main

```
public class MinhaClasse {  
    public static void main(String[] args) {  
        System.out.println("Olá, Mundo!"); // Imprime uma mensagem na tela  
    }  
}
```

void: Indica que o método não retorna nenhum valor.

Método main

```
public class MinhaClasse {  
    public static void main(String[] args) {  
        System.out.println("Olá, Mundo!"); // Imprime uma mensagem na tela  
    }  
}
```

main: Nome do método especial que a JVM procura para iniciar a execução do programa.

Método main

```
public class MinhaClasse {  
    public static void main(String[] args) {  
        System.out.println("Olá, Mundo!"); // Imprime uma mensagem na tela  
    }  
}
```

String[] args: Parâmetro que permite passar argumentos ao programa pela linha de comando.

Funções

—

Assinatura de Função

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

A assinatura de uma função em Java inclui o tipo de retorno, o nome da função e seus parâmetros.

Assinatura de Função

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

public: Modificador de acesso.

Assinatura de Função

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

int: Tipo de retorno da função.

Assinatura de Função

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

soma: Nome da função.

Assinatura de Função

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

(int a, int b): Parâmetros da função, ambos do tipo int.

Assinatura de Função

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

a + b; Corpo da função, também chamado de implementação

Assinatura de Função

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

return: Indica retorno, o que será devolvido para onde a função foi chamada

Tipos de Variáveis

—

Tipos de Variáveis

Tipo	Tamanho (bits)	Valor Padrão	Faixa de Valores	Exemplo de Uso
`byte`	8	0	-128 a 127	`byte b = 100;`
`short`	16	0	-32.768 a 32.767	`short s = 10000;`
`int`	32	0	-2.147.483.648 a 2.147.483.647	`int i = 123456;`
`long`	64	0L	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807	`long l = 123456789L;`
`float`	32	0.0f	Aproximadamente $\pm 3.40282347E+38F$ (precisão de 6-7 dígitos)	`float f = 3.14f;`
`double`	64	0.0d	Aproximadamente $\pm 1.79769313486231570E+308$ (precisão de 15 dígitos)	`double d = 3.141592653589793;`
`char`	16	'\u0000'	Caracteres Unicode (0 a 65.535)	`char c = 'A';`
`boolean`	1	false	true ou false	`boolean b = true;`
`String`	Variável	null	N/A	`String s = "Olá";`

Tipos de Variáveis - Inteiros

Tipo	Tamanho (bits)	Valor Padrão	Faixa de Valores	Exemplo de Uso
`byte`	8	0	-128 a 127	`byte b = 100;`
`short`	16	0	-32 768 a 32 767	`short s = 10000;`

byte: Um tipo de dado para economizar espaço em grandes arrays. Pode ser usado no lugar de int quando é sabido que os valores estarão dentro da faixa de -128 a 127.

Tipos de Variáveis - Inteiros

Tipo	Tamanho (bits)	Valor Padrão	Faixa de Valores	Exemplo de Uso
`short`	16	0	-32.768 a 32.767	<code>`short s = 10000;`</code>
`int`	32	0	-2.147.483.648 a 2.147.483.647	<code>`int i = 123456;`</code>

short: Também usado para economizar espaço em grandes arrays, pode ser usado em vez de int quando os valores estão na faixa de -32.768 a 32.767.

Tipos de Variáveis - Inteiros

Tipo	Tamanho (bits)	Valor Padrão	Faixa de Valores	Exemplo de Uso
<code>short</code>	16	0	-32.768 a 32.767	<code>short s = 10000;</code>
<code>int</code>	32	0	-2.147.483.648 a 2.147.483.647	<code>int i = 123456;</code>
<code>long</code>	64	0	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807	<code>long l = 123456789L;</code>

int: O tipo de dado inteiro mais comum usado em Java. Representa números inteiros na faixa de -2.147.483.648 a 2.147.483.647.

Tipos de Variáveis - Inteiros

Tipo	Tamanho (bits)	Valor Padrão	Faixa de Valores	Exemplo de Uso
<code>`long`</code>	64	0L	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807	<code>`long l = 123456789L;`</code>

long: Usado quando um valor maior do que o suportado por `int` é necessário. Representa números inteiros na faixa de -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807.

Tipos de Variáveis - Ponto Flutuante:

Tipo	Tamanho (bits)	Valor Padrão	Faixa de Valores	Exemplo de Uso
<code>`float`</code>	32	0.0f	Aproximadamente $\pm 3.40282347E+38F$ (precisão de 6-7 dígitos)	<code>`float f = 3.14f;`</code>

float: Representa números de ponto flutuante de precisão simples. Deve-se usar **f** ao final do número.

Tipos de Variáveis - Ponto Flutuante:

Tipo	Tamanho (bits)	Valor Padrão	Faixa de Valores	Exemplo de Uso
<code>`double`</code>	64	0.0d	Aproximadamente $\pm 1.79769313486231570E+308$ (precisão de 15 dígitos)	<code>`double d = 3.141592653589793;`</code>

double: Representa números de ponto flutuante de precisão dupla e é o tipo padrão para números decimais.

Tipos de Variáveis - Caractere:

Tipo	Tamanho (bits)	Valor Padrão	Faixa de Valores	Exemplo de Uso
<code>`char`</code>	16	<code>'\u0000'</code>	Caracteres Unicode (0 a 65.535)	<code>`char c = 'A';`</code>

char: Armazena um único caractere Unicode. Usa aspas simples (') para definir o valor.

Tipos de Variáveis - Booleano:

Tipo	Tamanho (bits)	Valor Padrão	Faixa de Valores	Exemplo de Uso
<code>`boolean`</code>	1	false	true ou false	<code>`boolean b = true;`</code>
<code>`String`</code>	Variaável	null	N/A	<code>`String s = "Olá!";`</code>

boolean: Armazena valores verdadeiros (true) ou falsos (false).

Tipos de Variáveis - String:

Tipo	Tamanho (bits)	Valor Padrão	Faixa de Valores	Exemplo de Uso
<code>`String`</code>	Variável	null	N/A	<code>`String s = "Olá";`</code>

String: Armazena uma sequência de caracteres. Usa aspas duplas ("). Embora não seja um tipo primitivo, é amplamente utilizado em Java e tem características especiais na linguagem.

Exemplo de uso dos tipos de variáveis

—

```
public class TiposDeDados {  
    public static void main(String[] args) {  
        // Tipos Inteiros  
        byte idade = 30;  
        short ano = 2023;  
        int populacao = 1500000;  
        long distancia = 9876543210L;  
  
        // Tipos de Ponto Flutuante  
        float preco = 19.99f;  
        double pi = 3.141592653589793;  
  
        // Tipo Caractere  
        char letra = 'A';  
    }  
}
```

```
// Tipo Booleano
```

```
boolean isJavaFun = true;
```

```
// Tipo String
```

```
String mensagem = "Olá, Mundo!";
```

```
// Exibição dos Valores
```

```
System.out.println("Idade: " + idade);
```

```
System.out.println("Ano: " + ano);
```

```
System.out.println("População: " + populacao);
```

```
System.out.println("Distância: " + distancia);
```

```
System.out.println("Preço: " + preco);
```

```
System.out.println("Pi: " + pi);
```

```
System.out.println("Letra: " + letra);
```

```
System.out.println("Java é divertido: " + isJavaFun);
```

```
System.out.println("Mensagem: " + mensagem);
```

```
}
```

```
}
```

Entrada e saída de dados

—

Entrada e saída de dados - ENTRADA

```
import java.util.Scanner;


public class EntradaDados {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in); // Criação de um objeto Scanner
        System.out.println("Digite sua idade: ");
        int idade = scanner.nextInt(); // Lê um inteiro do usuário

        System.out.println("Sua idade é: " + idade);
        scanner.close(); // Fecha o scanner
    }
}
```

Para **entrada** e saída de dados, Java utiliza a classe **Scanner** para **entrada** e **System.out** para saída.

Entrada e saída de dados - SAIDA

```
public class SaidaDados {  
    public static void main(String[] args) {  
        System.out.println("Olá, Mundo!"); // Imprime uma linha  
        System.out.print("Este é um exemplo de ");  
        System.out.print("saída de dados. "); // Imprime na mesma linha  
        System.out.printf("Você tem %d anos.\n", 25); // Saída formatada  
    }  
}
```



Para entrada e **saída** de dados, Java utiliza a classe Scanner para entrada e **System.out** para saída.

Estrutura de decisão

—

Estrutura Condicional if

```
if (condição) {  
    // Código a ser executado se a condição for verdadeira  
}
```

A estrutura **if** permite **executar** um bloco de código se uma condição específica for **verdadeira**.

Estrutura Condicional if

```
public class ExemploIf {  
    public static void main(String[] args) {  
        int numero = 10;  
  
        if (numero > 5) {  
            System.out.println("O número é maior que 5");  
        }  
    }  
}
```

A estrutura **if** permite **executar** um bloco de código se uma condição específica for **verdadeira**.

Estrutura Condicional if-else

```
public class ExemploIfElse {  
    public static void main(String[] args) {  
        int numero = 3;  
  
        if (numero > 5) {  
            System.out.println("O número é maior que 5");  
        } else {  
            System.out.println("O número não é maior que 5");  
        }  
    }  
}
```

A estrutura **if-else** permite executar um **bloco** de código se a condição for **verdadeira**, e outro **bloco** se a condição for **falsa**.

Estrutura Condicional if-else

```
if (condição) {  
    // Código a ser executado se a condição for verdadeira  
} else {  
    // Código a ser executado se a condição for falsa  
}
```

A estrutura **if-else** permite executar um **bloco** de código se a condição for **verdadeira**, e outro **bloco** se a condição for **falsa**.

Estrutura de Repetição

—

Laço "for"

```
for (inicialização; condição; incremento/decremento) {  
    // Código a ser repetido  
}
```

O laço **for** é usado quando sabemos antecipadamente quantas vezes queremos executar um bloco de código. A sintaxe do for inclui a inicialização, a **condição de continuidade** e a **atualização do contador**.

Laço "for"

```
public class ExemploFor {  
    public static void main(String[] args) {  
        // Imprime os números de 1 a 5  
        for (int i = 1; i <= 5; i++) {  
            System.out.println("Número: " + i);  
        }  
    }  
}
```

O laço **for** é usado quando sabemos antecipadamente quantas vezes queremos executar um bloco de código. A sintaxe do for inclui a inicialização, a **condição de continuidade** e a **atualização do contador**.

Laço "while"

```
while (condição) {  
    // Código a ser repetido  
}
```

O laço **while** é usado quando não sabemos antecipadamente quantas vezes precisamos repetir um bloco de código. O bloco de código **será executado enquanto a condição** especificada for **verdadeira**.

Laço "while"

```
public class ExemploWhile {  
    public static void main(String[] args) {  
        int contador = 1;  
  
        // Imprime os números de 1 a 5  
        while (contador <= 5) {  
            System.out.println("Número: " + contador);  
            contador++;  
        }  
    }  
}
```

O laço **while** é usado quando não sabemos antecipadamente quantas vezes precisamos repetir um bloco de código. O bloco de código **será executado enquanto a condição** especificada for **verdadeira**.

FIM

Slide e Código fonte:

