Classes e Objectos

PROGRAMAÇÃO ORIENTADO A OBJECTOS(POO)

Aulas finais de java! Presente de Despedida.☺

APRESENTAÇÃO DO CONTEÚDO

O que iremos ver:

- O que é Programação Orientada a Objetos (POO)
- Classes e Objetos
- Atributos e Métodos
- Construtores
- Os 4 Pilares da POO:
  - Encapsulamento
  - Herança
  - Polimorfismo
  - Abstração
- Exercícios propostos

PROGRAMAÇÃO ORIENTADO A OBJECTOS(POO)

O QUE É PROGRAMAÇÃO ORIENTADA A OBJETOS?

Programação Orientada a Objetos (POO) é uma forma de programar que organiza o código em "objetos" que representam coisas do mundo real.

Por que POO é importante?

Antes da POO, os programas eram escritos de forma **procedural** (sequencial)/**estrutural**, o que tornava difícil:

- Organizar código grande
- Reutilizar código
- Manter e atualizar programas

POO resolve isso organizando código em objetos que:

- Representam coisas reais (Carro, Pessoa, Conta Bancária)
- Agrupam dados relacionados
- Escondem complexidade
- Facilitam reutilização

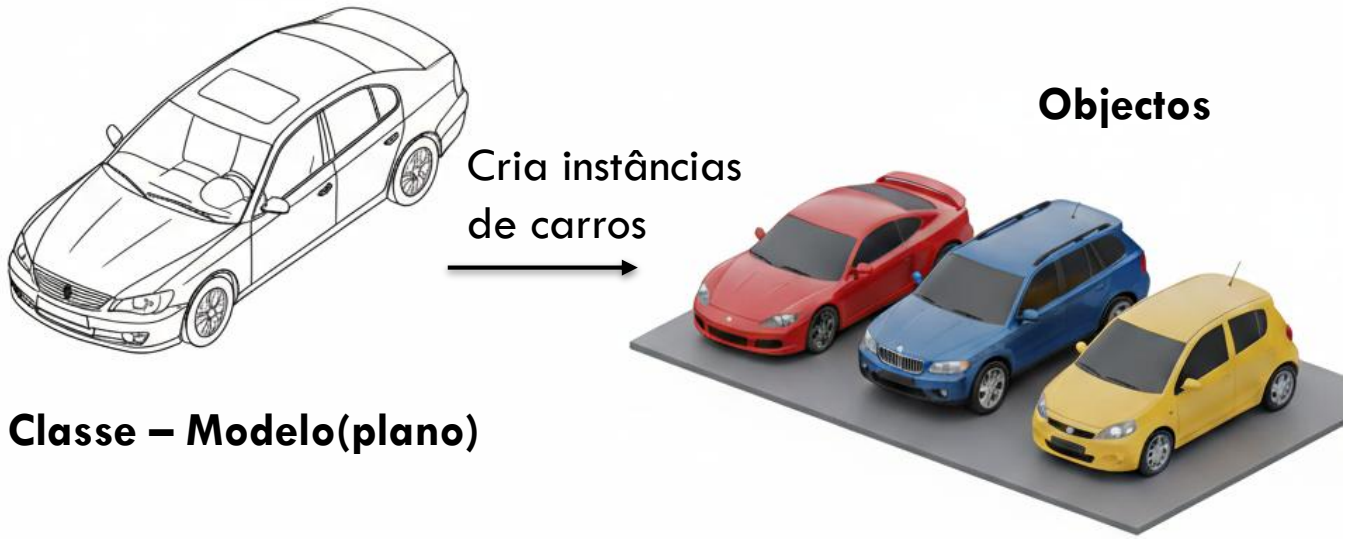
CLASSES E OBJECTOS

O que é uma Classe?

Uma **classe** é um **modelo** ou **plano** para criar objetos. É como uma planta de casa.

O que é um Objeto?

Um **objeto** é uma **instância** de uma classe. É a casa construída a partir da planta.



Exemplo(classe Carro):

```
public class Carro {
    // ATRIBUTOS - Características do carro
    String marca;
    String modelo;
    int ano;
    String cor;
    // MÉTODOS - Ações que o carro pode fazer
    void ligar() {
        System.out.println("Carro ligado!");
    }
    void acelerar() {
        System.out.println("Vrummm! Acelerando...");
    }
    void frear() {
        System.out.println("Freando...");
    }
}
```

Exemplo de uso:

```
public class Principal {

    public static void main(String[] args) {

        Carro meuCarro = new Carro(); // Carro 1

        meuCarro.marca = "Toyota";

        meuCarro.modelo = "Corolla";

        meuCarro.ano = 2020;

        meuCarro.cor = "Prata";

        Carro seuCarro = new Carro(); // Carro 2

        seuCarro.marca = "Honda";

        seuCarro.modelo = "Civic";

        seuCarro.ano = 2021;

        seuCarro.cor = "Preto";

        // Usando os objetos

        System.out.println("Meu carro: " + meuCarro.marca + " " + meuCarro.modelo);

        meuCarro.ligar();

        meuCarro.acelerar();

        System.out.println("Seu carro: " + seuCarro.marca + " " + seuCarro.modelo);

        seuCarro.ligar();

    }

}
```

ENTENDENDO A CRIAÇÃO DE OBJECTOS

Exemplo:

```
Carro meuCarro = new Carro();
```

- `Carro` - tipo do objeto (a classe)
- `meuCarro` - nome da variável (referência ao objeto)
- `new` - palavra-chave que cria o objeto na memória
- `Carro()` - construtor que inicializa o objeto

É como pedir um carro na fábrica. A classe Carro é o modelo, e `new Carro()` fabrica um carro físico para você.

CONSTRUTORES

CONSTRUTORES

Um **construtor** é um método especial usado para inicializar objetos quando são criados.

Características do Construtor:

- Tem o mesmo nome da classe
- Não tem tipo de retorno (nem void)
- É chamado automaticamente com **new**

CONSTRUTORES(EXEMPLO)

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

    // CONSTRUTOR - inicializa o objeto
    public Pessoa(String nome, int idade) {
        this.nome = nome;
        this.idade = idade;
    }
}
```

```
public class Principal {
    public static void main(String[] args) {
        // Criando objetos já com valores iniciais
        Pessoa pessoa1 = new Pessoa("João", 25);
        Pessoa pessoa2 = new Pessoa("Maria", 30);

        pessoa1.apresentar();
        pessoa2.apresentar();
    }
}
```

OS 4 PILARES DA POO

A POO tem **4 conceitos fundamentais** que tornam o código melhor:

- **Encapsulamento** - Proteger dados
- **Herança** - Reutilizar código
- **Polimorfismo** - Flexibilidade
- **Abstração** - Simplificar complexidade

ENCAPSULAMENTO

Encapsulamento é o conceito de **esconder** os detalhes internos e **proteger** os dados de uma classe.

Por que encapsular?

Imagine um carro: você acelera com o pedal, mas não precisa (nem deve) mexer diretamente no motor. O encapsulamento protege o "motor" do código.

Como encapsular:

- Tornar atributos **private**
- Criar métodos **public** para acessar (getters e setters)

EXEMPLO **SEM** ENCAPSULAMENTO

```
public class ContaBancaria {
    public double saldo;  // Qualquer um pode mexer!
}

public class Main {
    public static void main(String[] args) {
        ContaBancaria conta = new ContaBancaria();
        conta.saldo = 1000;

        // PERIGO! Alguém pode fazer isso:
        conta.saldo = -5000;  // Saldo negativo! 😱
    }
}
```

Exemplo **com** Encapsulamento

```
public class ContaBancaria {
    // Atributo PRIVADO - protegido
    private double saldo;
    public ContaBancaria(double saldoInicial) {
        if (saldoInicial >= 0) {
            this.saldo = saldoInicial;
        }
    }
    // GETTER - para ler o saldo
    public double getSaldo() {
        return saldo;
    }
    // Método controlado para depositar
    public void depositar(double valor) {
        if (valor > 0) {
            saldo += valor;
            System.out.println("Depósito de " + valor + " realizado!");
        } else {
            System.out.println("Valor inválido!");
        }
    }
    // Método controlado para levantar
    public void levantar(double valor) {
        if (valor > 0 && valor <= saldo) {
            saldo -= valor;
            System.out.println(“Levantamento de " + valor + "
realizado!");
        } else {
            System.out.println(“Levantamento impossível!");
        }
    }
}
```

EXEMPLO COM ENCAPSULAMENTO - CONTINUAÇÃO

```
public class Main {
    public static void main(String[] args) {
        ContaBancaria conta = new ContaBancaria(1000);

        // conta.saldo = -5000;  // ERRO! saldo é private
        System.out.println("Saldo: " + conta.getSaldo());
        conta.depositar(500);
        conta.levantar(200);
        conta.levantar(2000);  // Vai falhar - saldo insuficiente
        System.out.println("Saldo final: " + conta.getSaldo());
    }
}
```

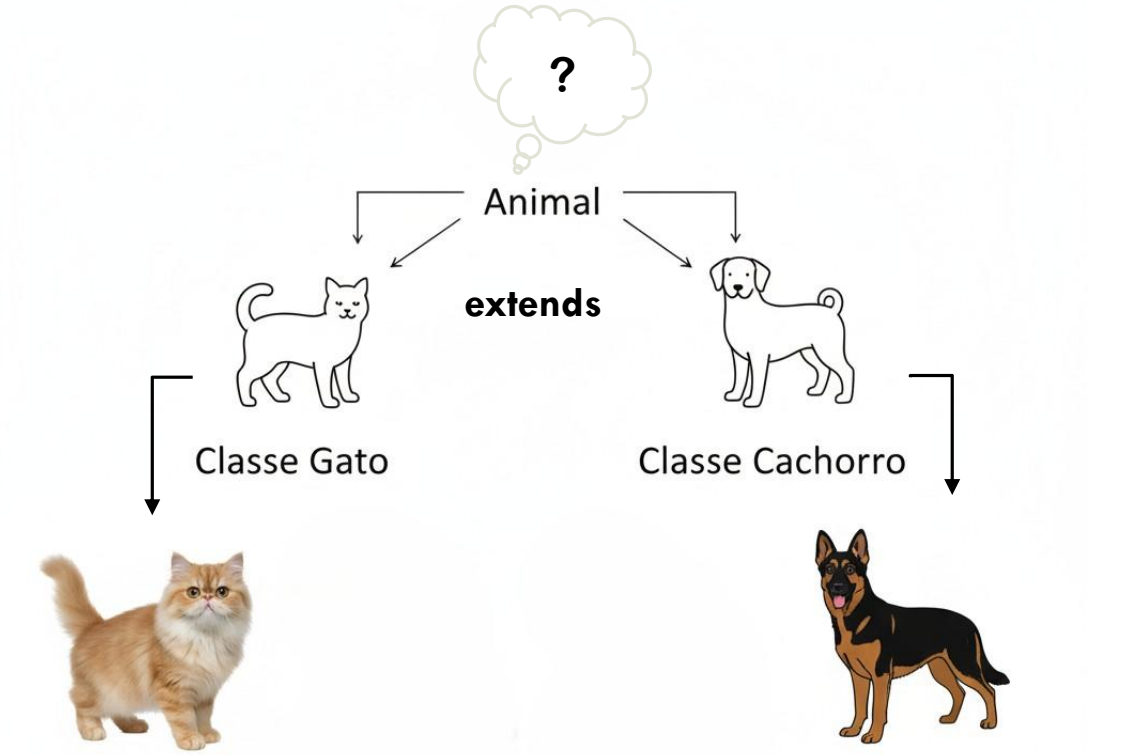
HERANÇA

Herança permite que uma classe **herde atributos e métodos** de outra classe, promovendo reutilização de código.

Terminologia:

- **Superclasse (Classe Pai):** classe que é herdada
- **Subclasse (Classe Filha):** classe que herda
- **Palavra-chave:** extends

EXEMPLO DE HERANÇA



Instância de Gato

Instância de Cachorro

EXEMPLO DE HERANÇA

```
// SUPERCLASSE - Classe pai
public class Animal {
    protected String nome;
    protected int idade;
    public Animal(String nome, int idade) {
        this.nome = nome;
        this.idade = idade;
    }
    public void comer() {
        System.out.println(nome + " está comendo.");
    }
    public void dormir() {
        System.out.println(nome + " está dormindo.");
    }
}
```

```
// SUBCLASSE 1 - Cachorro herda de Animal
 public class Cachorro extends Animal {
     private String raca;
     public Cachorro(String nome, int idade, String raca) {
         super(nome, idade);  // Chama construtor da superclasse
         this.raca = raca;
     }
     // Método específico de Cachorro
     public void latir() {
         System.out.println(nome + " está latindo: Au Au!");
     }
     // Sobrescrevendo método da superclasse
     @Override
     public void comer() {
         System.out.println(nome + " está comendo ração.");
     }
 }
```

EXEMPLO DE HERANÇA - CONTINUAÇÃO

```java
// SUBCLASSE 2 - Gato herda de Animal
public class Gato extends Animal {
    private String pelagem;
    public Gato(String nome, int idade, String pelagem) {
        super(nome, idade);
        this.pelagem = pelagem;
    }
    // Método específico de Gato
    public void miar() {
        System.out.println(nome + " está miando: Miau!");
    }
    @Override
    public void comer() {
        System.out.println(nome + " está comendo peixe.");
    }
}
```

```java
// Programa principal
public class Main {
    public static void main(String[] args) {
        Cachorro dog = new Cachorro("Rex", 3, "Labrador");
        Gato cat = new Gato("Mimi", 2, "Laranja");
        // Métodos herdados de Animal
        dog.comer();    // está comendo ração (sobrescrito)
        dog.dormir();   // está dormindo (herdado)
        dog.latir();    // Au Au! (específico)
        System.out.println();
        cat.comer();    // está comendo peixe (sobrescrito)
        cat.dormir();   // está dormindo (herdado)
        cat.miar();     // Miau! (específico)
    }
}
```

POLIMOSFIRMO

Polimorfismo significa "muitas formas". É a capacidade de um objeto se comportar de **diferentes maneiras**.

Tipos de Polimorfismo:

1. **Polimorfismo de Sobrescrita (Override)**

Já vimos: subclasse redefine método da superclasse.

2. **Polimorfismo de Referência**

Uma variável da superclasse pode referenciar objetos das subclasses.

EXEMPLO DE POLIMORFISMO

Temos a classe animal

```
public class Animal {
    protected String nome;

    public Animal(String nome) {
        this.nome = nome;
    }
    public void emitirSom() {
        System.out.println("Animal fazendo som...");
    }
}
```

Temos a classe Cachorro

```
public class Cachorro extends Animal {
    public Cachorro(String nome) {
        super(nome);
    }
    @Override
    public void emitirSom() {
        System.out.println(nome + ": Au Au!");
    }
}
```

Exemplo de Polimorfismo - Continuação

```java
public class Passaro extends Animal {
    public Passaro(String nome) {
        super(nome);
    }
    @Override
    public void emitirSom() {
        System.out.println(nome + ": Piu Piu!");
    }
}
```

```java
// POLIMORFISMO EM AÇÃO!
public class Main {
    public static void main(String[] args) {
        // Array de Animal pode conter qualquer subclasse
        Animal[] animais = new Animal[3];
        animais[0] = new Cachorro("Rex");
        animais[1] = new Gato("Mimi");
        animais[2] = new Passaro("Piu");
        // Cada animal emite seu próprio som!
        System.out.println("=== Fazenda de Animais ===");
        for (Animal animal : animais) {
            animal.emitirSom();  // Polimorfismo!
        }
    }
    // Método que aceita qualquer Animal
    public static void fazerAnimalEmitirSom(Animal animal) {
        animal.emitirSom();  // Funciona para qualquer subclasse!
    }
}
```

ABSTRAÇÃO

Abstração é o conceito de **simplificar** complexidade, mostrando apenas o essencial e escondendo detalhes.

Classes Abstratas

Uma **classe abstrata** é uma classe que **não pode ser instanciada** diretamente. Serve como modelo para outras classes.

EXEMPLO DE ABSTRAÇÃO

```
// Classe ABSTRATA - não pode criar objetos dela
public abstract class Forma {
    protected String cor;
    public Forma(String cor) {
        this.cor = cor;
    }
    // Método abstrato - subclasses DEVEM implementar
    public abstract double calcularArea();
    // Método concreto - pode ter implementação
    public void exibirCor() {
        System.out.println("Cor: " + cor);
    }
}
```

```
public class Circulo extends Forma {
    private double raio;

    public Circulo(String cor, double raio) {
        super(cor);
        this.raio = raio;
    }
    @Override
    public double calcularArea() {
        return Math.PI * raio * raio;
    }
}
```

EXEMPLO DE ABSTRAÇÃO - CONTINUAÇÃO

```
public class Retangulo extends Forma {
    private double largura;
    private double altura;

    public Retangulo(String cor, double largura, double altura) {
        super(cor);
        this.largura = largura;
        this.altura = altura;
    }
    @Override
    public double calcularArea() {
        return largura * altura;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Forma f = new Forma("azul");  // ERRO! Classe abstrata
        Forma circulo = new Circulo("Vermelho", 5);
        Forma retangulo = new Retangulo("Azul", 4, 6);
        System.out.println(circulo.calcularArea());
        circulo.exibirCor();
        System.out.println(retangulo.calcularArea());
        retangulo.exibirCor();
    }
}
```

FIM

Feito por:

- Anselmo Nhamage