POO Java
Programação Orientada a Objetos Java

Conteúdo desenvolvido por: Olival Paulino
Canal do Youtube: https://youtube.com/@olivalpaulino
Instagram: https://www.instagram.com/olivalpaulino/

Fala, Dev! Usei o poder da inteligência artificial, para criar um conteúdo diferenciado com o foco em te apresentar todos os tópicos de orientação a objetos usando Java, que acredito que você vai precisar em um futuro próximo. Esse conteúdo é para ser compartilhado com todos que desejam aprender programação orientada a objetos. Tornando o aprendizado colaborativo.

Leia os conteúdos, crie os códigos, adapte-os, teste, use-o para estudar para provas, testes, concursos, e principalmente, para fazer da programação Java a sua profissão, e se tornar um programador(a), um Dev melhor.

Não tenha medo de usar inteligência artificial para tirar as dúvidas que surgirem aqui, e potencializar ainda mais o seus estudos. O meu canal do youtube acima tem vários projetos, conceitos e práticas que você pode desenvolver e fortalecer tudo na prática.

Aproveite e Conheça todos os meus cursos e acelere ainda mais o seu aprendizado:

- Java na Prática – Do Básico ao Avançado: https://pay.kiwify.com.br/55BuhNc
- Projeto Java Full Stack, Sistema de Lanchonete: https://pay.kiwify.com.br/V8tIuJ3
- Projeto Java Back-end, Sistema de Gestão de Atendimentos: https://pay.kiwify.com.br/0vmjNYG
- Java na Web com Spring Boot: https://pay.kiwify.com.br/meoh7va
- Java no Front-end para Desktop: https://pay.kiwify.com.br/TxhrQlt
- Java no Front-end para Web com Spring Boot: https://pay.kiwify.com.br/QcDb7zc

Sucesso!

Sumário:

5. Relacionamentos entre Classes

- **Associação**
  - Unidirecional
  - Bidirecional
- **Agregação**
- **Composição**
- **Dependência**
- **Generalização e Especialização**

6. Classes Especiais e Tipos de Objetos

- **Classes Abstratas**
- **Interfaces**
- **Objetos Imutáveis**
- **Objetos Mutáveis**
- **Objetos Singleton**
- **Objetos de Transferência de Dados (DTOs – Data Transfer Objects)**

7. Exceções e Tratamento de Erros

- **Checked Exceptions**
- **Unchecked Exceptions**
- **Lançamento de Exceções (throw e throws)**
- **Criação de Exceções Personalizadas**

8. Métodos Especiais da Classe Object

- **equals()**
- **hashCode()**
- **toString()**
- **clone()**
- **finalize()**

9. Reflexão e Metaprogramação

- **Reflexão em Java (Reflection API)**
- **Manipulação Dinâmica de Objetos**
- **Criação Dinâmica de Classes**

10. JavaBeans e POJOs (Plain Old Java Objects)

- **Padrão JavaBean**
- **Propriedades com getters e setters**

11. Programação Genérica e Tipos Parametrizados

- **Generics em Classes e Métodos**
- **Wildcards (? extends T, ? super T)**

12. Serialização e Persistência de Objetos

- **Interface Serializable**
- **Escrita e Leitura de Objetos**

13. Design Patterns Relacionados à POO

- **Criacionais** (Singleton, Factory, Builder, Prototype)
- **Estruturais** (Adapter, Composite, Proxy, Decorator)
- **Comportamentais** (Observer, Strategy, Command)

1. Fundamentos da Programação Orientada a Objetos (POO)

A Programação Orientada a Objetos é um **paradigma de programação** baseado no conceito de objetos. Esses objetos representam entidades do mundo real e interagem entre si para resolver problemas.

A POO tem **quatro pilares fundamentais**:

✅ **Abstração**
✅ **Encapsulamento**
✅ **Herança**
✅ **Polimorfismo**

Vamos explorar cada um desses conceitos com detalhes e exemplos em Java.

1.1 Objetos e Classes

Na POO, **um objeto** é uma instância de uma **classe**.

Classe

É um **modelo** ou **molde** para criar objetos. Define os atributos (**dados**) e métodos (**comportamentos**) de um objeto.

Objeto

É uma **instância de uma classe**, criada a partir de seu modelo. Ele contém valores próprios para seus atributos e pode executar métodos definidos na classe.

Exemplo em Java

```
// Definição de uma classe
class Carro {
    String marca;
    int ano;

    void buzinar() {
        System.out.println("Biiiii!");
    }
}

// Criando objetos da classe Carro
public class Main {
    public static void main(String[] args) {
        Carro meuCarro = new Carro();
        meuCarro.marca = "Toyota";
        meuCarro.ano = 2022;
        meuCarro.buzinar();  // Saída: Biiiii!
    }
}
```

📌 **Resumo**: Uma **classe** define o que um objeto pode ter e fazer. Um **objeto** é uma cópia dessa classe em memória.

1.2 Abstração
A **abstração** permite representar um objeto do mundo real destacando **somente os aspectos essenciais** e ignorando detalhes irrelevantes.

Exemplo
Imagine um **carro**. Para um programador, ele precisa saber apenas alguns atributos e comportamentos, como:
✅ **Atributos** → cor, modelo, ano, marca.
✅ **Métodos** → acelerar(), frear(), ligar().

Detalhes como **o funcionamento interno do motor** não são relevantes para a abstração do carro.

Exemplo em Java
```
abstract class Veiculo {
    String modelo;
    int ano;

    // Método abstrato (não implementado aqui)
    abstract void ligar();
}

// Classe concreta implementando o método abstrato
class Moto extends Veiculo {
    @Override
    void ligar() {
        System.out.println("Moto ligada com sucesso!");
    }
}

public class Main {
    public static void main(String[] args) {
        Moto minhaMoto = new Moto();
        minhaMoto.modelo = "Honda";
        minhaMoto.ano = 2021;
        minhaMoto.ligar();  // Saída: Moto ligada com sucesso!
    }
}
```

📌 **Resumo**: A **abstração** foca nos **atributos e métodos essenciais** de um objeto, ignorando detalhes internos.

1.3 Encapsulamento

O **encapsulamento** protege os dados do objeto, impedindo que sejam modificados diretamente. Ele é implementado usando **modificadores de acesso** (private, public, protected).

Benefícios do Encapsulamento

✅ Protege os dados do objeto.
✅ Evita acessos e modificações indevidas.
✅ Facilita a manutenção do código.

Exemplo sem encapsulamento (ruim)

```
class Pessoa {
   String nome; // Qualquer um pode alterar diretamente
}

public class Main {
   public static void main(String[] args) {
      Pessoa p = new Pessoa();
      p.nome = "João";  // Sem controle!
      System.out.println(p.nome);
   }
}
```

Aqui, qualquer código pode alterar o nome sem nenhuma validação.

Exemplo com encapsulamento (correto)

```
class Pessoa {
   private String nome;  // Protegido

   // Método para definir o nome (com validação)
   public void setNome(String nome) {
      if (nome.length() > 2) {
         this.nome = nome;
      } else {
         System.out.println("Nome inválido!");
      }
   }

   // Método para obter o nome
   public String getNome() {
      return nome;
   }
}

public class Main {
   public static void main(String[] args) {
      Pessoa p = new Pessoa();
      p.setNome("João");  // OK
      System.out.println(p.getNome());  // Saída: João

      p.setNome("A");  // Nome inválido!
   }
}
```

📌 **Resumo**: O **encapsulamento** protege os dados e fornece **métodos seguros** para acessá-los.

1.4 Herança
A **herança** permite que uma classe filha **herde** atributos e métodos de uma classe pai.

Vantagens da Herança
✅ Reutiliza código.
✅ Facilita a manutenção.
✅ Evita redundância.

Exemplo de Herança em Java
```
// Classe base (superclasse)
class Animal {
    String nome;

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

// Classe derivada (subclasse)
class Cachorro extends Animal {
    void latir() {
        System.out.println("Au au!");
    }
}

public class Main {
    public static void main(String[] args) {
        Cachorro dog = new Cachorro();
        dog.nome = "Rex";
        dog.dormir();  // Herdado da classe Animal
        dog.latir();   // Método da classe Cachorro
    }
}
```

📌 **Resumo**: A **herança** permite que uma classe herde atributos e métodos de outra classe.

1.5 Polimorfismo
O **polimorfismo** permite que um mesmo método tenha **diferentes comportamentos**, dependendo do contexto.

Tipos de Polimorfismo
- **Polimorfismo de Sobrescrita (Override)**
- **Polimorfismo de Sobrecarga (Overload)**

Exemplo de Sobrescrita (Override)
```
class Animal {
    void emitirSom() {
        System.out.println("Som genérico de animal.");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Animal a = new Gato();
        a.emitirSom();  // Saída: Miau!
    }
}
```

Aqui, o método emitirSom() foi **sobrescrito** na classe Gato.
Exemplo de Sobrecarga (Overload)
```
class Calculadora {
    int somar(int a, int b) {
        return a + b;
    }

    int somar(int a, int b, int c) {
        return a + b + c;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();
        System.out.println(calc.somar(2, 3));       // Saída: 5
        System.out.println(calc.somar(2, 3, 4));   // Saída: 9
    }
}
```

📌 **Resumo**: O **polimorfismo** permite que métodos se comportem **de maneiras diferentes** dependendo do contexto.

Conclusão

Esses são os **quatro fundamentos** da POO:

✅ **Abstração** → Esconder detalhes desnecessários.

✅ **Encapsulamento** → Proteger os dados do objeto.

✅ **Herança** → Reutilizar código de outra classe.

✅ **Polimorfismo** → Permitir comportamentos diferentes para um mesmo método.

Quer aprofundar algum conceito? 🚀

2. Princípios da POO (SOLID)

O SOLID é um conjunto de **cinco princípios** definidos por **Robert C. Martin (Uncle Bob)**, que ajudam a criar **código mais organizado, flexível e de fácil manutenção**.

Esses princípios são essenciais para desenvolver sistemas modulares, reutilizáveis e que seguem boas práticas de design.

✅ **S** – *Single Responsibility Principle* (**Princípio da Responsabilidade Única**)

✅ **O** – *Open/Closed Principle* (**Princípio Aberto/Fechado**)

✅ **L** – *Liskov Substitution Principle* (**Princípio da Substituição de Liskov**)

✅ **I** – *Interface Segregation Principle* (**Princípio da Segregação de Interfaces**)

✅ **D** – *Dependency Inversion Principle* (**Princípio da Inversão de Dependência**)

Agora, vamos explorar cada um deles em detalhes com exemplos práticos em **Java**.

2.1 S – Single Responsibility Principle (SRP)

📌 **Cada classe deve ter apenas uma única responsabilidade e motivo para mudar.**

Se uma classe tem **muitas responsabilidades**, o código se torna difícil de **manter, testar e modificar**.

Exemplo de Violação do SRP (Errado)

```
class Relatorio {
    void gerarRelatorio() {
        System.out.println("Gerando relatório...");
    }

    void salvarEmArquivo() {
        System.out.println("Salvando relatório no disco...");
    }
}
```

🔴 **Problema:** A classe Relatorio está fazendo **duas coisas**:

1️⃣ Gerar o relatório.

2️⃣ Salvar o relatório em arquivo.

Isso **viola o SRP**, pois se a forma de salvar mudar (exemplo: salvar no banco de dados), a classe Relatorio precisaria ser alterada.

Aplicando SRP (Correto)

```
class Relatorio {
    void gerarRelatorio() {
        System.out.println("Gerando relatório...");
    }
}

class RelatorioArquivo {
    void salvarEmArquivo(Relatorio relatorio) {
        System.out.println("Salvando relatório no disco...");
    }
}
```

✅ **Agora cada classe tem uma única responsabilidade!**

- Relatorio → Apenas gera relatórios.
- RelatorioArquivo → Apenas salva relatórios.

2.2 O – Open/Closed Principle (OCP)

📌 **Classes devem estar abertas para extensão, mas fechadas para modificação.**

Ou seja, devemos **evitar modificar código existente**, mas devemos poder adicionar novas funcionalidades **sem alterar o código original**.

Exemplo de Violação do OCP (Errado)

```
class Desconto {
    double calcularDesconto(String tipoCliente, double valor) {
        if (tipoCliente.equals("VIP")) {
            return valor * 0.9; // 10% de desconto
        } else if (tipoCliente.equals("Regular")) {
            return valor * 0.95; // 5% de desconto
        }
        return valor;
    }
}
```

🔴 **Problema:** Sempre que um novo tipo de cliente surgir, **precisamos modificar essa classe**.

Aplicando OCP (Correto)

```
interface Desconto {
    double calcular(double valor);
}

class DescontoVIP implements Desconto {
    public double calcular(double valor) {
        return valor * 0.9;
    }
}

class DescontoRegular implements Desconto {
    public double calcular(double valor) {
        return valor * 0.95;
    }
}

class CalculadoraDesconto {
    double aplicarDesconto(Desconto desconto, double valor) {
        return desconto.calcular(valor);
    }
}
```

✅ Agora, se quisermos adicionar um novo tipo de cliente (exemplo: **"SuperVIP"**), **basta criar uma nova classe** sem modificar o código existente!

2.3 L – Liskov Substitution Principle (LSP)

📌 **Subtipos devem ser substituíveis por seus tipos base sem quebrar o sistema.**

Isso significa que se uma classe herda de outra, ela **deve manter o comportamento esperado**, sem alterar funcionalidades essenciais.

Exemplo de Violação do LSP (Errado)

```
class Retangulo {
    protected int largura;
    protected int altura;

    void setLargura(int largura) { this.largura = largura; }
    void setAltura(int altura) { this.altura = altura; }

    int getArea() {
        return largura * altura;
    }
}

class Quadrado extends Retangulo {
    @Override
    void setLargura(int largura) {
        this.largura = largura;
        this.altura = largura; // FORÇANDO a altura = largura
    }

    @Override
    void setAltura(int altura) {
        this.largura = altura;
        this.altura = altura; // FORÇANDO a largura = altura
    }
}
```

🔴 **Problema:** O Quadrado altera o comportamento de Retangulo, **violando o LSP**.

Aplicando LSP (Correto)

```
interface Forma {
    int getArea();
}

class Retangulo implements Forma {
    protected int largura;
    protected int altura;

    public Retangulo(int largura, int altura) {
        this.largura = largura;
        this.altura = altura;
    }

    public int getArea() {
        return largura * altura;
    }
}

class Quadrado implements Forma {
    private int lado;

    public Quadrado(int lado) {
        this.lado = lado;
    }

    public int getArea() {
        return lado * lado;
    }
}
```

✅ **Agora, cada forma é independente e mantém o comportamento esperado!**

2.4 I – Interface Segregation Principle (ISP)

📌 **Uma classe não deve ser forçada a implementar métodos que não utiliza.**

Exemplo de Violação do ISP (Errado)

```
interface Funcionario {
    void calcularSalario();
    void baterPonto();
    void realizarAtendimento();
}

class Desenvolvedor implements Funcionario {
    public void calcularSalario() { }
    public void baterPonto() { }
    public void realizarAtendimento() { } // ❌ NÃO FAZ SENTIDO!
}
```

🔴 **Problema:** A classe Desenvolvedor é obrigada a implementar um método que **não faz sentido para ela**.

Aplicando ISP (Correto)

```
interface Trabalhador {
    void calcularSalario();
    void baterPonto();
}

interface Atendente {
    void realizarAtendimento();
}

class Desenvolvedor implements Trabalhador {
    public void calcularSalario() { }
    public void baterPonto() { }
}

class SuporteTecnico implements Trabalhador, Atendente {
    public void calcularSalario() { }
    public void baterPonto() { }
    public void realizarAtendimento() { }
}
```

✅ **Agora, cada classe implementa apenas as interfaces que fazem sentido para ela!**

2.5 D – Dependency Inversion Principle (DIP)

📌 **Dependa de abstrações, não de implementações concretas.**

Exemplo de Violação do DIP (Errado)

```
class MySQLDatabase {
    void conectar() { System.out.println("Conectando ao MySQL..."); }
}

class Aplicacao {
    private MySQLDatabase database = new MySQLDatabase();
}
```

🔴 **Problema:** A classe Aplicacao depende diretamente de MySQLDatabase, dificultando mudanças futuras.

Aplicando DIP (Correto)

```
interface Database {
    void conectar();
}

class MySQLDatabase implements Database {
    public void conectar() { System.out.println("Conectando ao MySQL..."); }
}

class Aplicacao {
    private Database database;

    public Aplicacao(Database database) {
        this.database = database;
    }
}
```

✅ **Agora podemos trocar o banco de dados sem alterar a Aplicacao!**

Conclusão

Os princípios SOLID garantem **código mais organizado, escalável e fácil de manter**.
Quer aprofundar algum deles? 🚀

3. Padrões de Projetos (Design Patterns)
Os **Design Patterns** são **soluções reutilizáveis para problemas comuns** no desenvolvimento de software. Eles foram catalogados pelo **Gang of Four (GoF)** no livro *Design Patterns: Elements of Reusable Object-Oriented Software*.

Os padrões são divididos em **três categorias principais**:
✅ **3.1 Padrões Criacionais** – Controlam a criação de objetos.
✅ **3.2 Padrões Estruturais** – Definem como classes e objetos se relacionam.
✅ **3.3 Padrões Comportamentais** – Definem como os objetos interagem entre si.
Agora, vamos explorar cada um deles com exemplos em **Java**! 🚀

3.1 Padrões Criacionais (Creational Patterns)
Os padrões criacionais ajudam na **criação de objetos** de forma flexível e desacoplada.

3.1.1 Singleton
📌 **Garante que uma classe tenha apenas uma única instância e fornece um ponto global de acesso a ela.**
Exemplo de Singleton em Java

```
class Configuracao {
    private static Configuracao instancia;

    private Configuracao() { } // Construtor privado

    public static Configuracao getInstancia() {
        if (instancia == null) {
            instancia = new Configuracao();
        }
        return instancia;
    }
}
```

✅ **Usamos o getInstancia() para garantir que só exista um objeto da classe Configuracao!**

3.1.2 Factory Method
📌 **Cria objetos sem expor a lógica de criação ao cliente.**
Exemplo de Factory Method

```
interface Produto {
    void exibir();
}

class ProdutoA implements Produto {
    public void exibir() { System.out.println("Produto A criado!"); }
}

class ProdutoB implements Produto {
    public void exibir() { System.out.println("Produto B criado!"); }
}

class Fabrica {
    static Produto criarProduto(String tipo) {
        if (tipo.equals("A")) return new ProdutoA();
        if (tipo.equals("B")) return new ProdutoB();
        return null;
    }
}
```

✅ **O cliente chama Fabrica.criarProduto("A"), sem saber qual classe concreta será usada.**

3.2 Padrões Estruturais (Structural Patterns)
Os padrões estruturais organizam **a composição entre classes e objetos**.

3.2.1 Adapter
📌 **Permite que classes incompatíveis trabalhem juntas.**

Exemplo de Adapter
```
class TomadaAntiga {
    void conectar() { System.out.println("Conectado à tomada antiga!"); }
}

interface NovaTomada {
    void conectarNovo();
}

class Adaptador implements NovaTomada {
    private TomadaAntiga tomadaAntiga;

    public Adaptador(TomadaAntiga tomadaAntiga) {
        this.tomadaAntiga = tomadaAntiga;
    }

    public void conectarNovo() {
        tomadaAntiga.conectar();
    }
}
```

✅ **Agora um sistema novo pode usar a TomadaAntiga sem precisar modificar seu código!**

3.2.2 Composite
📌 **Permite tratar um grupo de objetos da mesma maneira que um único objeto.**

Exemplo de Composite
```
interface Componente {
    void mostrar();
}

class Arquivo implements Componente {
    private String nome;

    public Arquivo(String nome) { this.nome = nome; }

    public void mostrar() { System.out.println("Arquivo: " + nome); }
}

class Pasta implements Componente {
    private String nome;
    private List<Componente> filhos = new ArrayList<>();

    public Pasta(String nome) { this.nome = nome; }
    public void adicionar(Componente c) { filhos.add(c); }

    public void mostrar() {
        System.out.println("Pasta: " + nome);
        for (Componente c : filhos) c.mostrar();
    }
}
```
✅ **Criamos uma árvore de arquivos e pastas, onde ambos podem ser manipulados da mesma forma!**

3.3 Padrões Comportamentais (Behavioral Patterns)
Os padrões comportamentais definem **como os objetos interagem e trocam mensagens**.

3.3.1 Strategy
📌 **Permite escolher algoritmos em tempo de execução.**

Exemplo de Strategy
```
interface EstrategiaPagamento {
    void pagar(double valor);
}

class PagamentoCartao implements EstrategiaPagamento {
    public void pagar(double valor) { System.out.println("Pago R$" + valor + " com Cartão."); }
}

class PagamentoPix implements EstrategiaPagamento {
    public void pagar(double valor) { System.out.println("Pago R$" + valor + " com PIX."); }
}

class Checkout {
    private EstrategiaPagamento estrategia;

    public void setEstrategia(EstrategiaPagamento estrategia) {
        this.estrategia = estrategia;
    }

    public void processarPagamento(double valor) {
        estrategia.pagar(valor);
    }
}
```

✅ **Podemos trocar a estratégia de pagamento em tempo de execução!**

3.3.2 Observer

📌 **Define uma dependência 1 para N entre objetos, onde um objeto notifica outros quando seu estado muda.**

Exemplo de Observer

```
interface Observador {
    void atualizar(String mensagem);
}

class Usuario implements Observador {
    private String nome;

    public Usuario(String nome) { this.nome = nome; }

    public void atualizar(String mensagem) {
        System.out.println(nome + " recebeu a notificação: " + mensagem);
    }
}

class Notificador {
    private List<Observador> observadores = new ArrayList<>();

    public void adicionarObservador(Observador o) {
        observadores.add(o);
    }

    public void notificar(String mensagem) {
        for (Observador o : observadores) {
            o.atualizar(mensagem);
        }
    }
}
```

✅ **Quando algo acontece no Notificador, todos os usuários são avisados automaticamente!**

Conclusão

Os **Design Patterns** ajudam a construir **sistemas modulares, flexíveis e reutilizáveis**. Eles resolvem problemas comuns e tornam o código mais organizado.

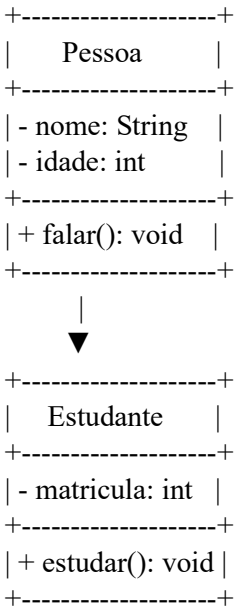
Se quiser aprofundar em algum padrão específico, me avise! 🚀

4. UML (Unified Modeling Language)

A **UML (Linguagem de Modelagem Unificada)** é um **padrão para modelar sistemas orientados a objetos**.
Ela fornece **diagramas visuais** que ajudam a projetar, visualizar e documentar a estrutura e o comportamento de um sistema.

A UML é composta por **14 diagramas**, organizados em duas categorias principais:
✅ **4.1 Diagramas Estruturais** – Representam a estrutura do sistema.
✅ **4.2 Diagramas Comportamentais** – Representam o comportamento e a interação entre os objetos.
Vamos explorar **cada diagrama** em detalhes! 🚀

4.1 Diagramas Estruturais
Os diagramas estruturais descrevem **a organização dos elementos do sistema**.
4.1.1 Diagrama de Classes
📌 **Mostra a estrutura do sistema, incluindo classes, atributos, métodos e relacionamentos.**
Exemplo de Diagrama de Classes UML



✅ **Explicação:**
- Pessoa tem os atributos nome e idade.
- Estudante herda (extends) Pessoa e adiciona matricula.

Exemplo Java baseado no diagrama

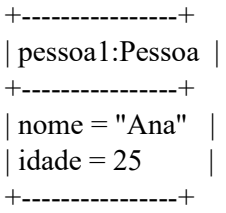
```
class Pessoa {
    protected String nome;
    protected int idade;

    public void falar() {
        System.out.println("Falando...");
    }
}

class Estudante extends Pessoa {
    private int matricula;

    public void estudar() {
        System.out.println("Estudando...");
    }
}
```

🚀 **O diagrama de classes serve como um mapa para o código!**

4.1.2 Diagrama de Objetos

📌 **Representa instâncias de classes em um determinado momento.**

Exemplo UML

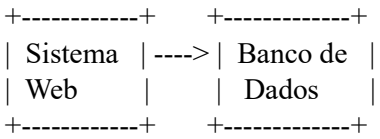


✅ **Representa um objeto pessoa1 da classe Pessoa com valores reais.**

4.1.3 Diagrama de Componentes

📌 **Mostra os módulos do sistema e suas dependências.**

Exemplo UML

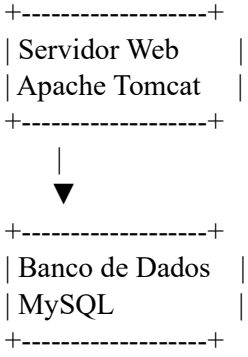


✅ **Usado para representar a arquitetura do sistema!**

4.1.4 Diagrama de Implantação

📌 **Mostra como o sistema é implantado em hardware (servidores, redes, etc.).**

Exemplo UML



✅ **Indica onde cada componente do sistema será executado!**

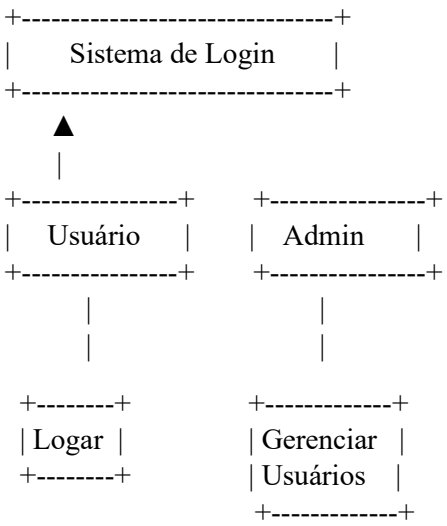
4.2 Diagramas Comportamentais

Os diagramas comportamentais representam **como o sistema funciona e interage com os usuários e outros sistemas**.

4.2.1 Diagrama de Casos de Uso

📌 **Representa funcionalidades do sistema e os atores envolvidos.**

Exemplo UML

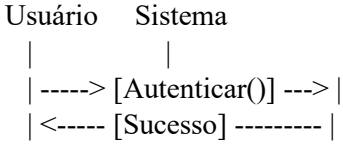


✅ **Mostra que o "Usuário" pode fazer login, e o "Admin" pode gerenciar usuários.**

4.2.2 Diagrama de Sequência

📌 **Mostra a interação entre objetos ao longo do tempo.**

Exemplo UML



✅ **Mostra a troca de mensagens entre Usuário e Sistema durante a autenticação.**

Exemplo Java

```
class Sistema {
    boolean autenticar(String usuario, String senha) {
        return "admin".equals(usuario) && "1234".equals(senha);
    }
}
```

4.2.3 Diagrama de Atividades

📌 **Mostra o fluxo de atividades no sistema.**

Exemplo UML

Início → Inserir Dados → Validar Login → [OK?] → Acesso Permitido → Fim
| Não |
↓
Erro de Login

✅ **Indica que, se a validação falhar, o usuário recebe um erro.**

4.2.4 Diagrama de Estados

📌 **Representa os estados pelos quais um objeto passa durante seu ciclo de vida.**

Exemplo UML

[Criado] → [Ativo] → [Bloqueado] → [Encerrado]

✅ **Útil para modelar processos como status de pedidos ou usuários.**

Conclusão

A UML ajuda a **planejar, visualizar e documentar** sistemas antes mesmo de escrever código.
Se quiser aprofundar em algum diagrama, me avise! 🚀

5. Princípios SOLID
Os princípios **SOLID** são um conjunto de **boas práticas de design de software** para tornar o código **mais modular, flexível e fácil de manter**.

O nome **SOLID** vem das iniciais de cinco princípios:
✅ **5.1 - SRP (Single Responsibility Principle) - Princípio da Responsabilidade Única**
✅ **5.2 - OCP (Open/Closed Principle) - Princípio Aberto/Fechado**
✅ **5.3 - LSP (Liskov Substitution Principle) - Princípio da Substituição de Liskov**
✅ **5.4 - ISP (Interface Segregation Principle) - Princípio da Segregação de Interfaces**
✅ **5.5 - DIP (Dependency Inversion Principle) - Princípio da Inversão de Dependência**
Cada um desses princípios resolve um problema comum no design de software. Vamos detalhar um por um! 🚀

5.1 - SRP (Single Responsibility Principle)
📌 **Uma classe deve ter apenas uma razão para mudar.**
Isso significa que **cada classe deve ter apenas uma responsabilidade**. Se uma classe faz muitas coisas diferentes, pode ser difícil de manter e modificar.
❌ **Código errado (violação do SRP)**
```
class Relatorio {
    public void gerarRelatorio() {
        System.out.println("Gerando relatório...");
    }

    public void salvarNoBanco() {
        System.out.println("Salvando no banco de dados...");
    }
}
```

🚨 **Problema:** A classe Relatorio tem **duas responsabilidades**:
1. Gerar o relatório
2. Salvar no banco de dados

Se precisarmos mudar a forma como o relatório é salvo, teremos que modificar essa classe, violando o **SRP**.

✅ **Código corrigido (seguindo o SRP)**
```
class GeradorRelatorio {
    public void gerarRelatorio() {
        System.out.println("Gerando relatório...");
    }
}

class RelatorioDAO {
    public void salvarNoBanco() {
        System.out.println("Salvando no banco de dados...");
    }
}
```

✅ **Agora cada classe tem apenas uma responsabilidade!**

5.2 - OCP (Open/Closed Principle)

📌 **Uma classe deve estar aberta para extensão, mas fechada para modificação.**

Isso significa que podemos **adicionar novas funcionalidades sem modificar o código existente**.

❌ **Código errado (violação do OCP)**

```
class CalculadoraSalario {
    public double calcularSalario(String cargo, double salarioBase) {
        if (cargo.equals("DESENVOLVEDOR")) {
            return salarioBase * 1.2;
        } else if (cargo.equals("GERENTE")) {
            return salarioBase * 1.5;
        }
        return salarioBase;
    }
}
```

🚨 **Problema:** Se precisarmos adicionar um novo cargo, teremos que **modificar** essa classe, o que pode gerar bugs.

✅ **Código corrigido (seguindo o OCP)**

```
interface RegraDeCalculo {
    double calcular(double salarioBase);
}

class RegraDesenvolvedor implements RegraDeCalculo {
    public double calcular(double salarioBase) {
        return salarioBase * 1.2;
    }
}

class RegraGerente implements RegraDeCalculo {
    public double calcular(double salarioBase) {
        return salarioBase * 1.5;
    }
}

class CalculadoraSalario {
    public double calcularSalario(RegraDeCalculo regra, double salarioBase) {
        return regra.calcular(salarioBase);
    }
}
```

✅ **Agora, para adicionar um novo cargo, basta criar uma nova classe sem modificar o código existente!**

5.3 - LSP (Liskov Substitution Principle)

📌 **Se uma classe S é uma subclasse de T, então os objetos de T podem ser substituídos por objetos de S sem afetar o funcionamento do programa.**

Em outras palavras, **uma subclasse deve poder substituir a superclasse sem quebrar o código**.

❌ **Código errado (violação do LSP)**

```
class Ave {
    public void voar() {
        System.out.println("Estou voando!");
    }
}

class Pinguim extends Ave { }
```

🚨 **Problema:** O **pinguim** não voa! Mas como ele herda de Ave, ele herda o método voar(), o que não faz sentido.
✅ **Código corrigido (seguindo o LSP)**

```
class Ave { }

class AveQueVoa extends Ave {
   public void voar() {
      System.out.println("Estou voando!");
   }
}

class Pinguim extends Ave { }
class Andorinha extends AveQueVoa { }
```

✅ **Agora, apenas as aves que realmente voam têm o método voar().**

5.4 - ISP (Interface Segregation Principle)
📌 **Uma classe não deve ser forçada a implementar métodos que não usa.**

Isso significa que **interfaces grandes devem ser divididas em interfaces menores**.

❌ **Código errado (violação do ISP)**

```
interface Funcionario {
   void calcularSalario();
   void baterPonto();
   void receberComissao();
}

class Desenvolvedor implements Funcionario {
   public void calcularSalario() { /* ok */ }
   public void baterPonto() { /* ok */ }
   public void receberComissao() { /* NÃO USA! */ }
}
```

🚨 **Problema:** Nem todos os funcionários recebem comissão, mas são obrigados a implementar receberComissao().
✅ **Código corrigido (seguindo o ISP)**

```
interface Funcionario {
   void calcularSalario();
   void baterPonto();
}

interface Comissionado {
   void receberComissao();
}

class Desenvolvedor implements Funcionario {
   public void calcularSalario() { /* ok */ }
   public void baterPonto() { /* ok */ }
}

class Vendedor implements Funcionario, Comissionado {
   public void calcularSalario() { /* ok */ }
   public void baterPonto() { /* ok */ }
   public void receberComissao() { /* ok */ }
}
```

✅ **Agora cada classe implementa apenas os métodos que realmente usa.**

5.5 - DIP (Dependency Inversion Principle)

📌 **Módulos de alto nível não devem depender de módulos de baixo nível, ambos devem depender de abstrações.**

Isso significa que devemos **evitar dependências diretas de classes concretas**, e preferir depender de interfaces ou classes abstratas.

❌ **Código errado (violação do DIP)**

```
class Motor {
    public void ligar() {
        System.out.println("Motor ligado!");
    }
}

class Carro {
    private Motor motor = new Motor(); // 🚨 DEPENDÊNCIA DIRETA

    public void ligarCarro() {
        motor.ligar();
    }
}
```

🚨 **Problema:** Se quisermos trocar o motor por um MotorEletrico, teremos que modificar Carro.

✅ **Código corrigido (seguindo o DIP)**

```
interface Motor {
    void ligar();
}

class MotorCombustao implements Motor {
    public void ligar() {
        System.out.println("Motor a combustão ligado!");
    }
}

class MotorEletrico implements Motor {
    public void ligar() {
        System.out.println("Motor elétrico ligado!");
    }
}

class Carro {
    private Motor motor;

    public Carro(Motor motor) {
        this.motor = motor;
    }

    public void ligarCarro() {
        motor.ligar();
    }
}
```

✅ **Agora podemos trocar o motor sem alterar a classe Carro!**

Conclusão

Os princípios **SOLID** ajudam a criar sistemas mais organizados, flexíveis e fáceis de manter.
Se quiser aprofundar em algum princípio, me avise! 🚀

6. Padrões de Projeto (Design Patterns)

Os **padrões de projeto** são soluções **reutilizáveis** para problemas comuns no desenvolvimento de software.
Eles ajudam a **organizar o código**, **reduzir acoplamento** e **melhorar a manutenção** do sistema.

Os padrões de projeto são divididos em **três categorias principais**:

✅ **6.1 - Padrões Criacionais** → Focados na criação de objetos.

✅ **6.2 - Padrões Estruturais** → Focados na organização das classes e objetos.

✅ **6.3 - Padrões Comportamentais** → Focados na interação entre objetos.

Vamos explorar cada categoria e seus principais padrões! 🚀

6.1 - Padrões Criacionais

Os padrões criacionais ajudam a **criar objetos de maneira eficiente**, evitando instâncias desnecessárias e garantindo flexibilidade.

6.1.1 - Singleton

📌 **Garante que uma classe tenha apenas uma instância no sistema e fornece um ponto global de acesso a ela.**

❌ **Código errado (sem Singleton)**

```
class ConexaoBanco {
    public ConexaoBanco() {
        System.out.println("Nova conexão criada.");
    }
}
```

🧢 **Problema:** Toda vez que criarmos um novo objeto ConexaoBanco, será uma nova conexão.

✅ **Código corrigido (Singleton)**

```
class ConexaoBanco {
    private static ConexaoBanco instancia;

    private ConexaoBanco() { } // Construtor privado

    public static ConexaoBanco getInstance() {
        if (instancia == null) {
            instancia = new ConexaoBanco();
        }
        return instancia;
    }
}
```

✅ **Agora o sistema sempre usa a mesma instância, economizando recursos!**

6.1.2 - Factory Method

📌 **Cria objetos sem expor a lógica de criação diretamente.**

❌ **Código errado (sem Factory)**

```
class Carro {
    public Carro(String modelo) {
        System.out.println("Carro modelo: " + modelo);
    }
}
```

🚨 **Problema:** Se quisermos criar diferentes tipos de carros, teremos que modificar essa classe diretamente.

✅ **Código corrigido (com Factory)**

```
class CarroFactory {
    public static Carro criarCarro(String modelo) {
        return new Carro(modelo);
    }
}
```

✅ **Agora podemos criar carros sem modificar a classe Carro!**

6.2 - Padrões Estruturais

Os padrões estruturais ajudam a **organizar classes e objetos** para que o código fique mais flexível e reutilizável.

6.2.1 - Adapter

📌 **Permite que duas classes incompatíveis trabalhem juntas.**

❌ **Código errado (sem Adapter)**

```
class TomadaBrasil {
    public void conectar() {
        System.out.println("Conectado na tomada brasileira!");
    }
}

class TomadaEUA {
    public void plugIn() {
        System.out.println("Plugged into US outlet!");
    }
}
```

🚨 **Problema:** Os métodos conectar() e plugIn() são diferentes, então não podemos usá-los de forma intercambiável.

✅ **Código corrigido (com Adapter)**

```
class AdaptadorTomada extends TomadaBrasil {
    private TomadaEUA tomadaEUA;

    public AdaptadorTomada(TomadaEUA tomadaEUA) {
        this.tomadaEUA = tomadaEUA;
    }

    @Override
    public void conectar() {
        tomadaEUA.plugIn();
    }
}
```

✅ **Agora podemos conectar dispositivos dos EUA em uma tomada do Brasil!**

6.2.2 - Composite

📌 **Permite tratar um grupo de objetos como um único objeto.**

✅ **Código corrigido (com Composite)**

```
interface Componente {
    void exibir();
}

class Arquivo implements Componente {
    private String nome;

    public Arquivo(String nome) {
        this.nome = nome;
    }

    public void exibir() {
        System.out.println("Arquivo: " + nome);
    }
}

class Pasta implements Componente {
    private List<Componente> componentes = new ArrayList<>();

    public void adicionar(Componente c) {
        componentes.add(c);
    }

    public void exibir() {
        for (Componente c : componentes) {
            c.exibir();
        }
    }
}
```

✅ **Agora podemos tratar arquivos e pastas como um único tipo (Componente)!**

6.3 - Padrões Comportamentais

Os padrões comportamentais **definem como os objetos interagem** entre si.

6.3.1 - Observer

📌 **Permite que objetos sejam notificados quando outro objeto mudar de estado.**

✅ **Código corrigido (com Observer)**

```
interface Observador {
    void atualizar(String mensagem);
}

class Usuario implements Observador {
    private String nome;

    public Usuario(String nome) {
        this.nome = nome;
    }

    public void atualizar(String mensagem) {
        System.out.println(nome + " recebeu: " + mensagem);
    }
}

class Canal {
    private List<Observador> observadores = new ArrayList<>();

    public void adicionarObservador(Observador o) {
        observadores.add(o);
    }

    public void notificar(String mensagem) {
        for (Observador o : observadores) {
            o.atualizar(mensagem);
        }
    }
}
```

✅ **Agora podemos notificar automaticamente os usuários quando algo acontecer no Canal!**

6.3.2 - Strategy

📌 **Permite alterar o comportamento de um objeto em tempo de execução.**

✅ **Código corrigido (com Strategy)**

```
interface EstrategiaPagamento {
    void pagar(double valor);
}

class PagamentoCartao implements EstrategiaPagamento {
    public void pagar(double valor) {
        System.out.println("Pagando R$" + valor + " com cartão.");
    }
}

class PagamentoPix implements EstrategiaPagamento {
    public void pagar(double valor) {
        System.out.println("Pagando R$" + valor + " via PIX.");
    }
}

class Pagamento {
    private EstrategiaPagamento estrategia;

    public void setEstrategia(EstrategiaPagamento estrategia) {
        this.estrategia = estrategia;
    }

    public void executarPagamento(double valor) {
        estrategia.pagar(valor);
    }
}
```

✅ **Agora podemos trocar a forma de pagamento sem modificar o código da classe Pagamento!**

Conclusão

Os **Padrões de Projeto** ajudam a criar software **mais modular, reutilizável e flexível**.
Se quiser aprofundar em algum padrão, me avise! 🚀

📌 7. Princípios SOLID

Os princípios **SOLID** foram criados por **Robert C. Martin (Uncle Bob)** e são um conjunto de **cinco diretrizes** que ajudam a escrever um código **mais limpo, coeso e desacoplado**.

Cada letra da palavra **SOLID** representa um princípio:

✅ **S - Single Responsibility Principle (SRP)** → Princípio da Responsabilidade Única
✅ **O - Open/Closed Principle (OCP)** → Princípio do Aberto/Fechado
✅ **L - Liskov Substitution Principle (LSP)** → Princípio da Substituição de Liskov
✅ **I - Interface Segregation Principle (ISP)** → Princípio da Segregação de Interfaces
✅ **D - Dependency Inversion Principle (DIP)** → Princípio da Inversão de Dependência

Agora, vamos explorar **cada princípio** com **detalhes, exemplos errados e correções**! 🚀

📌 7.1 - Single Responsibility Principle (SRP)

Uma classe deve ter apenas uma única responsabilidade.
Ou seja, deve haver **apenas um motivo para uma classe ser modificada**.

❌ Código errado (violando SRP)

```
class Relatorio {
    public void gerarRelatorio() {
        System.out.println("Gerando relatório...");
    }

    public void salvarNoBanco() {
        System.out.println("Salvando no banco...");
    }

    public void enviarPorEmail() {
        System.out.println("Enviando por e-mail...");
    }
}
```

🚨 Problema:

A classe **Relatorio** está fazendo **três coisas diferentes**:

1. Gerando um relatório
2. Salvando no banco
3. Enviando e-mail

Isso quebra o **SRP**, pois se a regra de envio de e-mail mudar, temos que alterar essa classe, mesmo que o relatório não tenha mudado.

✅ Código corrigido (respeitando SRP)

```
class GeradorRelatorio {
    public void gerar() {
        System.out.println("Gerando relatório...");
    }
}

class BancoDeDados {
    public void salvar(String dados) {
        System.out.println("Salvando no banco: " + dados);
    }
}

class EmailService {
    public void enviar(String destinatario, String conteudo) {
        System.out.println("Enviando e-mail para " + destinatario);
    }
}
```

✅ Agora cada classe tem apenas uma responsabilidade!

📌 7.2 - Open/Closed Principle (OCP)

Uma classe deve estar aberta para extensão, mas fechada para modificação.
Isso significa que **podemos adicionar novas funcionalidades sem modificar o código original**.

❌ Código errado (violando OCP)

```
class Calculadora {
    public double calcular(double a, double b, String operacao) {
        if (operacao.equals("+")) {
            return a + b;
        } else if (operacao.equals("-")) {
            return a - b;
        } else if (operacao.equals("*")) {
            return a * b;
        } else if (operacao.equals("/")) {
            return a / b;
        }
        throw new IllegalArgumentException("Operação inválida!");
    }
}
```

🚨 Problema:

Se precisarmos adicionar uma nova operação (por exemplo, potência ^), temos que **modificar** essa classe, quebrando o princípio OCP.

✅ Código corrigido (respeitando OCP com Polimorfismo)

```
interface Operacao {
    double executar(double a, double b);
}

class Soma implements Operacao {
    public double executar(double a, double b) {
        return a + b;
    }
}

class Subtracao implements Operacao {
    public double executar(double a, double b) {
        return a - b;
    }
}

class Calculadora {
    public double calcular(Operacao operacao, double a, double b) {
        return operacao.executar(a, b);
    }
}
```

✅ Agora podemos adicionar novas operações sem modificar a classe Calculadora!

📌 7.3 - Liskov Substitution Principle (LSP)
Uma subclasse deve poder substituir a superclasse sem alterar o comportamento esperado.

❌ Código errado (violando LSP)
```
class Ave {
    public void voar() {
        System.out.println("Voando...");
    }
}

class Pinguim extends Ave {
    // Pinguins não voam!
}
```

🚨 Problema:
Se chamarmos pinguim.voar(), ele herdou um comportamento **que não faz sentido**!

✅ Código corrigido (respeitando LSP)
```
interface Ave {
    void emitirSom();
}

interface AveVoadora extends Ave {
    void voar();
}

class Pardal implements AveVoadora {
    public void voar() {
        System.out.println("Voando...");
    }

    public void emitirSom() {
        System.out.println("Piu piu!");
    }
}

class Pinguim implements Ave {
    public void emitirSom() {
        System.out.println("Quack quack!");
    }
}
```

✅ Agora respeitamos o LSP e não temos aves que "voam" sem poder voar!

📌 7.4 - Interface Segregation Principle (ISP)

Uma interface não deve forçar a implementação de métodos que a classe não precisa.

❌ Código errado (violando ISP)

```
interface Funcionario {
    void calcularSalario();
    void baterPonto();
    void dirigir();
}

class Programador implements Funcionario {
    public void calcularSalario() { }
    public void baterPonto() { }
    public void dirigir() { } // Programador não precisa dirigir!
}
```

🚨 **Problema:**

A interface Funcionario está forçando um **Programador** a implementar dirigir(), algo que não faz sentido.

✅ Código corrigido (respeitando ISP)

```
interface Funcionario {
    void calcularSalario();
    void baterPonto();
}

interface Motorista {
    void dirigir();
}

class Programador implements Funcionario {
    public void calcularSalario() { }
    public void baterPonto() { }
}

class Entregador implements Funcionario, Motorista {
    public void calcularSalario() { }
    public void baterPonto() { }
    public void dirigir() { }
}
```

✅ **Agora cada classe implementa apenas o que faz sentido!**

📌 7.5 - Dependency Inversion Principle (DIP)

Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.

❌ Código errado (violando DIP)

```
class Teclado {
    public void conectar() {
        System.out.println("Teclado conectado!");
    }
}

class Computador {
    private Teclado teclado = new Teclado(); // Forte acoplamento!

    public void iniciar() {
        teclado.conectar();
    }
}
```

🚨 Problema:

A classe Computador depende diretamente de Teclado, tornando difícil mudar para outro tipo de entrada (como um **mouse**).

✅ Código corrigido (respeitando DIP)

```
interface DispositivoEntrada {
    void conectar();
}

class Teclado implements DispositivoEntrada {
    public void conectar() {
        System.out.println("Teclado conectado!");
    }
}

class Computador {
    private DispositivoEntrada dispositivo;

    public Computador(DispositivoEntrada dispositivo) {
        this.dispositivo = dispositivo;
    }

    public void iniciar() {
        dispositivo.conectar();
    }
}
```

✅ Agora podemos trocar Teclado por Mouse sem modificar Computador!

📌 Conclusão

Os princípios **SOLID** tornam o código **mais organizado, modular e flexível**. Aplicá-los corretamente melhora a **manutenção**, **reutilização** e **escalabilidade** do software! 🚀

📌 8. Relacionamentos entre Classes

Os relacionamentos entre classes definem **como os objetos interagem e se conectam** dentro de um sistema orientado a objetos. Eles são fundamentais para criar um **design limpo, coeso e reutilizável**.

⚡ Tipos de Relacionamento entre Classes

Os principais tipos de relacionamento entre classes são:

✅ **1. Associação** (Simples, Bidirecional, Unidirecional)
✅ **2. Agregação**
✅ **3. Composição**
✅ **4. Herança**
✅ **5. Dependência**

Agora, vamos explorar **cada um deles** com explicações detalhadas e exemplos práticos em Java! 🚀

📌 8.1 - Associação

A associação representa um vínculo entre duas classes.

Ela pode ser:

- **Unidirecional** → Apenas uma classe conhece a outra.
- **Bidirecional** → Ambas as classes se conhecem.

❌ Exemplo de Associação Unidirecional

```
class Cliente {
    private String nome;

    public Cliente(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }
}

class Pedido {
    private Cliente cliente; // Pedido conhece Cliente

    public Pedido(Cliente cliente) {
        this.cliente = cliente;
    }

    public void exibirPedido() {
        System.out.println("Pedido feito por: " + cliente.getNome());
    }
}

public class Main {
    public static void main(String[] args) {
        Cliente cliente = new Cliente("Carlos");
        Pedido pedido = new Pedido(cliente);
        pedido.exibirPedido();
    }
}
```

🔷 Explicação:

- A classe Pedido conhece Cliente, mas Cliente **não conhece** Pedido.
- Essa é uma associação **unidirecional**.

✅ **Exemplo de Associação Bidirecional**

```
class Cliente {
    private String nome;
    private Pedido pedido; // Cliente agora conhece Pedido

    public Cliente(String nome) {
        this.nome = nome;
    }

    public void setPedido(Pedido pedido) {
        this.pedido = pedido;
    }

    public String getNome() {
        return nome;
    }
}

class Pedido {
    private Cliente cliente;

    public Pedido(Cliente cliente) {
        this.cliente = cliente;
        cliente.setPedido(this); // Mantendo bidirecionalidade
    }

    public void exibirPedido() {
        System.out.println("Pedido feito por: " + cliente.getNome());
    }
}
```

🔷 **Explicação:**

- **Agora Cliente e Pedido se conhecem.**
- Se precisarmos do pedido a partir do cliente, podemos acessar cliente.getPedido().

📌 8.2 - Agregação

A agregação é uma relação "tem um", onde uma classe pode existir independentemente da outra.

💡 **Exemplo:** Um **"Curso"** tem vários **"Alunos"**, mas se o curso for encerrado, os alunos continuam existindo.

✅ Exemplo de Agregação

```
import java.util.ArrayList;
import java.util.List;

class Aluno {
    private String nome;

    public Aluno(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }
}

class Curso {
    private String nome;
    private List<Aluno> alunos = new ArrayList<>();

    public Curso(String nome) {
        this.nome = nome;
    }

    public void adicionarAluno(Aluno aluno) {
        alunos.add(aluno);
    }

    public void listarAlunos() {
        System.out.println("Alunos do curso " + nome + ":");
        for (Aluno aluno : alunos) {
            System.out.println(aluno.getNome());
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Aluno aluno1 = new Aluno("Ana");
        Aluno aluno2 = new Aluno("Lucas");

        Curso curso = new Curso("Java Completo");
        curso.adicionarAluno(aluno1);
        curso.adicionarAluno(aluno2);

        curso.listarAlunos();
    }
}
```

🔷 Explicação:

- Curso **tem** vários Aluno, mas Aluno pode existir sem Curso.
- Se o curso for excluído, os alunos **não desaparecem**.

📌 8.3 - Composição
A composição é uma relação "parte de", onde uma classe depende totalmente da outra para existir.

💡 **Exemplo:** Um **"Carro"** tem um **"Motor"**. Se o carro for destruído, o motor também deixa de existir.

✅ Exemplo de Composição
```
class Motor {
    private String tipo;

    public Motor(String tipo) {
        this.tipo = tipo;
    }

    public void ligar() {
        System.out.println("Motor " + tipo + " ligado!");
    }
}

class Carro {
    private String modelo;
    private Motor motor; // Composição: o Motor é parte do Carro

    public Carro(String modelo, String tipoMotor) {
        this.modelo = modelo;
        this.motor = new Motor(tipoMotor); // Motor é criado dentro do Carro
    }

    public void ligarCarro() {
        System.out.println("Ligando o carro " + modelo);
        motor.ligar();
    }
}

public class Main {
    public static void main(String[] args) {
        Carro carro = new Carro("Fusca", "1.6");
        carro.ligarCarro();
    }
}
```

🔷 **Explicação:**
- Motor **só existe** se houver um Carro.
- Se Carro for destruído, Motor **também é destruído**.

📌 8.4 - Herança
Herança é uma relação "é um".

💡 **Exemplo:** Um **"Cachorro" é um** tipo de **"Animal"**.

✅ Exemplo de Herança
```
class Animal {
    public void fazerSom() {
        System.out.println("Som genérico de animal");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Animal meuCachorro = new Cachorro();
        meuCachorro.fazerSom(); // Saída: "Au au!"
    }
}
```

🔷 **Explicação:**
- Cachorro **herda** os comportamentos de Animal.
- Podemos sobrescrever (@Override) métodos para **especializar o comportamento**.

📌 8.5 - Dependência
Uma classe depende temporariamente de outra para executar uma ação.

💡 **Exemplo:** Um **"Cliente"** usa um **"Pagamento"** para pagar um pedido.

✅ Exemplo de Dependência
```
class Pagamento {
    public void processarPagamento() {
        System.out.println("Pagamento processado!");
    }
}

class Cliente {
    public void pagar(Pagamento pagamento) {
        pagamento.processarPagamento();
    }
}

public class Main {
    public static void main(String[] args) {
        Cliente cliente = new Cliente();
        Pagamento pagamento = new Pagamento();

        cliente.pagar(pagamento);
    }
}
```

🔷 Explicação:
- Cliente usa Pagamento, mas **não mantém uma referência fixa**.

📌 Conclusão
Os relacionamentos entre classes são **essenciais** para modelar sistemas de forma eficiente. Cada tipo de relação tem **um propósito específico** e deve ser usado corretamente para garantir **um código modular, reutilizável e flexível**! 🚀

📌 9. Polimorfismo

O **polimorfismo** é um dos pilares fundamentais da **programação orientada a objetos (POO)**. Ele permite que **um mesmo método ou comportamento seja implementado de diferentes formas em diferentes classes**.

🎯 O que é o Polimorfismo?

A palavra "polimorfismo" vem do grego e significa **"muitas formas"**. Na prática, significa que **um mesmo método pode se comportar de maneiras diferentes dependendo do contexto**.

🔷 Tipos de Polimorfismo

O polimorfismo pode ser dividido em **duas categorias principais**:

- **Polimorfismo de Sobrecarga (Compile-time ou Estático)**
- **Polimorfismo de Substituição (Override ou Runtime ou Dinâmico)**

📌 9.1 - Polimorfismo de Sobrecarga (Overloading)

Permite que uma **classe tenha mais de um método com o mesmo nome**, desde que tenham **assinaturas diferentes**.

💡 A assinatura de um método inclui:

✅ Nome do método
✅ Quantidade de parâmetros
✅ Tipo dos parâmetros
✅ Ordem dos parâmetros

Importante: O tipo de retorno **não** é considerado para diferenciar métodos sobrecarregados!

✅ Exemplo de Sobrecarga de Métodos

```
class Calculadora {
    // Método para somar dois números inteiros
    public int somar(int a, int b) {
        return a + b;
    }

    // Método para somar três números inteiros (mesmo nome, parâmetros diferentes)
    public int somar(int a, int b, int c) {
        return a + b + c;
    }

    // Método para somar dois números decimais (tipos diferentes)
    public double somar(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();

        System.out.println(calc.somar(2, 3));      // Chama o método que soma dois inteiros
        System.out.println(calc.somar(2, 3, 4));   // Chama o método que soma três inteiros
        System.out.println(calc.somar(2.5, 3.5));  // Chama o método que soma dois doubles
    }
}
```

🔷 Explicação:

- O método somar tem **três versões diferentes**, cada uma com uma assinatura única.
- O compilador decide **qual versão chamar com base nos argumentos passados**.
- Esse tipo de polimorfismo ocorre **em tempo de compilação** (**compile-time polymorphism**).

📌 9.2 - Polimorfismo de Substituição (Override)

Permite que uma **subclasse modifique um método herdado de sua superclasse** para fornecer **uma implementação específica**.

💡 Regras para sobrescrever métodos

✅ A assinatura do método deve ser **exatamente a mesma** (nome e parâmetros idênticos).
✅ O tipo de retorno **pode ser o mesmo ou um subtipo covariante**.
✅ O nível de visibilidade **não pode ser reduzido** (por exemplo, um método public na superclasse não pode ser private na subclasse).
✅ Deve ser usado o **@Override** para indicar a sobrescrita.

✅ Exemplo de Sobrescrita de Métodos (Override)

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

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

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

public class Main {
    public static void main(String[] args) {
        Animal meuAnimal1 = new Cachorro();
        Animal meuAnimal2 = new Gato();

        meuAnimal1.fazerSom();  // Saída: O cachorro late: Au Au!
        meuAnimal2.fazerSom();  // Saída: O gato mia: Miau!
    }
}
```

🔷 Explicação:

- Cachorro e Gato **herdam de Animal** e sobrescrevem o método fazerSom().
- Mesmo que as variáveis meuAnimal1 e meuAnimal2 sejam do tipo Animal, o **método específico da subclasse é chamado**.
- Isso acontece porque, em **tempo de execução**, o Java verifica o tipo real do objeto.
- Esse tipo de polimorfismo ocorre **em tempo de execução** (**runtime polymorphism**).

📌 9.3 - Polimorfismo com Interfaces

O polimorfismo também pode ser usado com **interfaces**, permitindo que diferentes classes **implementem a mesma interface de formas distintas**.

✅ Exemplo com Interface

```
interface Pagamento {
    void realizarPagamento(double valor);
}

class CartaoCredito implements Pagamento {
    @Override
    public void realizarPagamento(double valor) {
        System.out.println("Pagamento de R$" + valor + " realizado via Cartão de Crédito");
    }
}

class Pix implements Pagamento {
    @Override
    public void realizarPagamento(double valor) {
        System.out.println("Pagamento de R$" + valor + " realizado via PIX");
    }
}

public class Main {
    public static void main(String[] args) {
        Pagamento pagamento1 = new CartaoCredito();
        Pagamento pagamento2 = new Pix();

        pagamento1.realizarPagamento(100.0);
        pagamento2.realizarPagamento(50.0);
    }
}
```

🔷 Explicação:

- CartaoCredito e Pix implementam a interface Pagamento.
- Podemos criar uma variável Pagamento e atribuir a ela qualquer classe que implemente essa interface.
- Isso permite flexibilidade e modularidade no código.

📌 9.4 - Benefícios do Polimorfismo

✅ **Reutilização de Código** → Evita duplicação de código ao permitir que diferentes classes compartilhem métodos comuns.
✅ **Facilidade de Manutenção** → Podemos modificar o comportamento das subclasses sem alterar o código da superclasse.
✅ **Extensibilidade** → O sistema pode ser facilmente expandido com novas funcionalidades sem afetar as classes existentes.
✅ **Flexibilidade** → Permite escrever código mais genérico e adaptável.

📌 Conclusão

O polimorfismo é **uma das características mais poderosas da POO**! Ele permite criar **código mais modular, reutilizável e flexível**, garantindo que métodos possam **se comportar de maneiras diferentes dependendo do contexto**.

💡 Resumo dos Tipos de Polimorfismo:

Tipo	Quando acontece?	Como funciona?
Sobrecarga (Overloading)	Em tempo de compilação	Mesmos nomes de métodos com parâmetros diferentes
Sobrescrita (Override)	Em tempo de execução	Subclasse redefine um método da superclasse

O **uso correto do polimorfismo** torna o código **mais elegante e adaptável**! 🚀

📌 10. Encapsulamento

O **encapsulamento** é um dos pilares da **programação orientada a objetos (POO)** e tem como principal objetivo **ocultar os detalhes internos de implementação de um objeto**, expondo apenas o necessário para sua interação externa.

Esse conceito garante **segurança, organização e modularidade** no código, permitindo que os dados internos de um objeto **não sejam acessados ou modificados diretamente**, mas sim através de métodos controlados.

🎯 O que é Encapsulamento?

O encapsulamento é o **princípio de restringir o acesso direto aos dados internos de um objeto e permitir a interação apenas por meio de métodos específicos**.

Em Java, ele é implementado utilizando **modificadores de acesso** (private, protected, public) e métodos **getter e setter** para controlar a leitura e escrita dos atributos.

✅ Vantagens do Encapsulamento:

- ✔ Proteção dos dados internos contra acessos indevidos.
- ✔ Maior controle sobre como os atributos são alterados.
- ✔ Redução do acoplamento entre classes.
- ✔ Facilidade na manutenção e evolução do código.

📌 10.1 - Modificadores de Acesso em Java

Os **modificadores de acesso** definem **o nível de visibilidade dos atributos e métodos** dentro de uma classe.

Modificador	Acesso dentro da classe	Acesso dentro do pacote	Acesso por subclasses	Acesso por outras classes
private	✅ Sim	❌ Não	❌ Não	❌ Não
default (sem modificador)	✅ Sim	✅ Sim	❌ Não	❌ Não
protected	✅ Sim	✅ Sim	✅ Sim	❌ Não
public	✅ Sim	✅ Sim	✅ Sim	✅ Sim

📌 10.2 - Criando uma Classe Encapsulada

Para garantir o encapsulamento, usamos:

🔹 **Atributos privados (private)** para impedir acesso direto.

🔹 **Métodos públicos (public)** para acessar e modificar os atributos (getters e setters).

✅ Exemplo de Encapsulamento

```
class ContaBancaria {
    // Atributos privados (não podem ser acessados diretamente)
    private String titular;
    private double saldo;

    // Construtor
    public ContaBancaria(String titular, double saldoInicial) {
        this.titular = titular;
        this.saldo = saldoInicial;
    }

    // Método getter para obter o titular da conta
    public String getTitular() {
        return titular;
    }

    // Método setter para alterar o titular da conta
    public void setTitular(String titular) {
        this.titular = titular;
    }

    // Método getter para obter o saldo
    public double getSaldo() {
        return saldo;
    }

    // Método para depositar dinheiro
    public void depositar(double valor) {
        if (valor > 0) {
            saldo += valor;
            System.out.println("Depósito de R$" + valor + " realizado com sucesso.");
        } else {
            System.out.println("Valor inválido para depósito.");
        }
    }

    // Método para sacar dinheiro
    public boolean sacar(double valor) {
        if (valor > 0 && valor <= saldo) {
            saldo -= valor;
            System.out.println("Saque de R$" + valor + " realizado com sucesso.");
            return true;
        } else {
            System.out.println("Saldo insuficiente ou valor inválido.");
            return false;
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Criando um objeto da classe ContaBancaria
        ContaBancaria conta = new ContaBancaria("João", 1000);

        // Acessando os dados via métodos públicos
        System.out.println("Titular da conta: " + conta.getTitular());
        System.out.println("Saldo inicial: R$" + conta.getSaldo());

        // Realizando operações
        conta.depositar(500);
        conta.sacar(300);
        conta.sacar(1500); // Tentativa de saque inválida

        // Exibindo saldo atualizado
        System.out.println("Saldo final: R$" + conta.getSaldo());
    }
}
```

🔹 **Explicação:**

- **Os atributos são privados (private)**, impedindo acesso direto.
- **Os métodos get e set controlam o acesso aos atributos.**
- O saldo **não pode ser alterado diretamente**, apenas pelos métodos depositar() e sacar().
- **O código fica mais seguro e controlado**, pois o usuário não pode inserir valores inválidos diretamente.

📌 10.3 - Encapsulamento e protected

O modificador protected permite que **subclasses (herança) tenham acesso aos atributos protegidos**, mas impede acesso direto por classes externas.

✅ Exemplo com protected

```
class Veiculo {
    protected String marca;

    public Veiculo(String marca) {
        this.marca = marca;
    }

    public void exibirMarca() {
        System.out.println("Marca: " + marca);
    }
}

class Carro extends Veiculo {
    private String modelo;

    public Carro(String marca, String modelo) {
        super(marca);
        this.modelo = modelo;
    }

    public void exibirModelo() {
        System.out.println("Modelo: " + modelo);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Carro carro = new Carro("Toyota", "Corolla");
        carro.exibirMarca(); // Acessa atributo protected via método
        carro.exibirModelo();
    }
}
```

🔹 **Explicação:**

- marca é protected, então a subclasse Carro pode acessá-lo.
- Se marca fosse private, Carro não poderia acessá-lo diretamente.

📌 **10.4 - Benefícios do Encapsulamento**

✅ **Segurança** → Os atributos não podem ser modificados indevidamente.

✅ **Facilidade de manutenção** → As regras de negócio podem ser modificadas sem impacto externo.

✅ **Ocultação da implementação** → Os detalhes internos do objeto são escondidos.

✅ **Redução do acoplamento** → As classes dependem menos umas das outras.

📌 **Conclusão**

O **encapsulamento** é **fundamental na POO**, pois protege os dados, melhora a organização e facilita a manutenção do código.

💡 **Resumo dos conceitos principais:**

🔹 **Atributos privados (private)** para restringir acesso direto.

🔹 **Métodos públicos (getters e setters)** para controlar leitura e escrita.

🔹 **Modificadores de acesso (private, protected, public)** definem a visibilidade dos membros da classe.

🔹 **O encapsulamento aumenta a segurança e modularidade do código.**

O **uso correto do encapsulamento** torna o sistema **mais robusto, seguro e organizado**! 🚀

📌 11. Herança

A **herança** é um dos pilares fundamentais da **programação orientada a objetos (POO)**. Ela permite que **uma classe (subclasse ou classe filha) herde atributos e métodos de outra classe (superclasse ou classe mãe)**, promovendo **reutilização de código** e **organização hierárquica**.

🎯 O que é Herança?

A herança estabelece uma **relação "é um"** entre classes.
Ou seja, se temos uma **classe Animal**, podemos ter subclasses como Cachorro e Gato, pois ambos são animais.

Em Java, usamos a **palavra-chave extends** para criar herança.

✅ Benefícios da Herança:

✔ **Reutilização de código** → Reduz a duplicação de código, pois as subclasses reaproveitam a implementação da superclasse.
✔ **Organização** → Estrutura hierárquica clara, facilitando o entendimento do sistema.
✔ **Extensibilidade** → Facilita a adição de novos comportamentos sem alterar código já existente.

📌 11.1 - Como Implementar Herança em Java?

Usamos a **palavra-chave extends** para indicar que uma classe herda de outra.

✅ Exemplo Básico de Herança

```
// Superclasse (Classe Pai)
class Animal {
    String nome;

    public void fazerSom() {
        System.out.println("O animal faz um som.");
    }
}

// Subclasse (Classe Filha)
class Cachorro extends Animal {
    public void abanarRabo() {
        System.out.println("O cachorro está abanando o rabo.");
    }
}

public class Main {
    public static void main(String[] args) {
        Cachorro dog = new Cachorro();
        dog.nome = "Rex";  // Herdado da classe Animal
        System.out.println("Nome: " + dog.nome);

        dog.fazerSom();    // Método herdado da classe Animal
        dog.abanarRabo();  // Método específico da classe Cachorro
    }
}
```

🔷 Explicação:

- A classe Animal tem um atributo nome e um método fazerSom().
- A classe Cachorro herda (extends Animal), então pode usar os métodos e atributos da superclasse.
- Além disso, Cachorro tem seu próprio método abanarRabo().

📌 11.2 - Construtores e Herança

A **subclasse não herda automaticamente o construtor** da superclasse, mas pode chamá-lo usando super().

✅ Exemplo com Construtor

```
class Animal {
    String nome;

    // Construtor da Superclasse
    public Animal(String nome) {
        this.nome = nome;
    }

    public void fazerSom() {
        System.out.println(nome + " faz um som.");
    }
}

// Subclasse
class Cachorro extends Animal {
    String raca;

    // Construtor da Subclasse chamando o da Superclasse
    public Cachorro(String nome, String raca) {
        super(nome);  // Chama o construtor da superclasse
        this.raca = raca;
    }

    public void abanarRabo() {
        System.out.println(nome + " está abanando o rabo.");
    }
}

public class Main {
    public static void main(String[] args) {
        Cachorro dog = new Cachorro("Rex", "Labrador");
        dog.fazerSom();
        dog.abanarRabo();
    }
}
```

🔷 Explicação:

- super(nome) chama o **construtor da superclasse Animal**.
- Assim, a subclasse Cachorro **não precisa reescrever código para inicializar nome**.

📌 11.3 - Herança e Modificadores de Acesso

Modificador	Pode ser herdado?	Pode ser acessado diretamente pela subclasse?
private	❌ Não	❌ Não
default	✅ Sim	❌ Não (exceto se estiver no mesmo pacote)
protected	✅ Sim	✅ Sim
public	✅ Sim	✅ Sim

Se um atributo ou método for private, a subclasse **não pode acessá-lo diretamente**, mas pode usar métodos public ou protected da superclasse.

✅ Exemplo de Herança com protected

```
class Animal {
    protected String nome;

    public Animal(String nome) {
        this.nome = nome;
    }

    public void exibirNome() {
        System.out.println("Nome: " + nome);
    }
}

class Gato extends Animal {
    public Gato(String nome) {
        super(nome);
    }

    public void miar() {
        System.out.println(nome + " está miando: Miau!");
    }
}

public class Main {
    public static void main(String[] args) {
        Gato gato = new Gato("Mingau");
        gato.exibirNome();
        gato.miar();
    }
}
```

🔹 Explicação:

- nome está como protected, então Gato pode acessá-lo diretamente.

📌 11.4 - Sobrescrita de Métodos (Override)

A subclasse pode **modificar o comportamento** de um método herdado usando **@Override**.

✅ Exemplo de Sobrescrita (@Override)

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

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

public class Main {
    public static void main(String[] args) {
        Animal meuAnimal = new Cachorro();
        meuAnimal.fazerSom(); // Chama o método sobrescrito de Cachorro
    }
}
```

🔷 Explicação:

- O método fazerSom() foi sobrescrito em Cachorro.
- Mesmo que meuAnimal seja do tipo Animal, ele chama a versão de Cachorro.

📌 11.5 - Herança Múltipla

Em Java, **não existe herança múltipla de classes** (uma classe não pode herdar de duas classes ao mesmo tempo). Isso evita o problema do "diamante".

💡 Mas Java suporta herança múltipla de interfaces!

✅ Exemplo de Herança Múltipla com Interfaces

```
interface Mamifero {
    void amamentar();
}

interface Carnivoro {
    void comerCarne();
}

class Leao implements Mamifero, Carnivoro {
    @Override
    public void amamentar() {
        System.out.println("O leão amamenta seus filhotes.");
    }

    @Override
    public void comerCarne() {
        System.out.println("O leão come carne.");
    }
}

public class Main {
    public static void main(String[] args) {
        Leao leao = new Leao();
        leao.amamentar();
        leao.comerCarne();
    }
}
```

🔷 Explicação:

- Leao implementa **duas interfaces (Mamifero e Carnivoro)**.
- Como interfaces não possuem implementação, **não há conflito de código**.

📌 11.6 - Benefícios da Herança

✅ **Evita código duplicado** → Métodos e atributos comuns são centralizados na superclasse.
✅ **Facilita manutenção** → Alterações na superclasse refletem automaticamente nas subclasses.
✅ **Melhora a organização** → Permite um design mais limpo e modular.
✅ **Facilita extensibilidade** → Podemos criar novas classes rapidamente.

📌 Conclusão

A **herança** permite **reutilizar código** e organizar o sistema em uma **hierarquia lógica**.

💡 Resumo:

- A subclasse **herda** atributos e métodos da superclasse (extends).
- O **construtor da superclasse** pode ser chamado com super().
- Métodos podem ser **sobrescritos** (@Override).
- Java **não suporta herança múltipla de classes**, mas suporta herança de **múltiplas interfaces**.

Usada corretamente, a **herança torna o código mais eficiente, reutilizável e fácil de manter!** 🚀

📌 12. Polimorfismo

O **polimorfismo** é um dos quatro pilares da Programação Orientada a Objetos (POO), junto com **abstração, encapsulamento e herança**. Ele permite que um **mesmo método possa ter diferentes comportamentos**, tornando o código mais **flexível, reutilizável e extensível**.

🎯 O que é Polimorfismo?

A palavra **polimorfismo** vem do grego e significa **"muitas formas"**.
Em Java, isso significa que um objeto pode **se comportar de diferentes maneiras dependendo do contexto**.

- O polimorfismo pode ser classificado em dois tipos:
  Polimorfismo de Sobrecarga (Compile-time Polymorphism)
- **Polimorfismo de Sobrescrita (Runtime Polymorphism)**

📌 12.1 - Polimorfismo de Sobrecarga (Method Overloading)

Ocorre quando **métodos com o mesmo nome são definidos dentro da mesma classe**, mas com **diferentes assinaturas (quantidade e tipo de parâmetros).**

💡 É resolvido em tempo de compilação.

✅ Exemplo de Sobrecarga de Método

```
class Calculadora {
    // Método 1: Soma de dois inteiros
    public int somar(int a, int b) {
        return a + b;
    }

    // Método 2: Soma de três inteiros (mesmo nome, mas com 3 parâmetros)
    public int somar(int a, int b, int c) {
        return a + b + c;
    }

    // Método 3: Soma de dois números decimais (float)
    public float somar(float a, float b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();

        System.out.println(calc.somar(5, 10));       // Chama o método com dois inteiros
        System.out.println(calc.somar(5, 10, 20));   // Chama o método com três inteiros
        System.out.println(calc.somar(5.5f, 2.5f));  // Chama o método com floats
    }
}
```

🔷 Explicação:

- Os métodos somar possuem o **mesmo nome**, mas **assinaturas diferentes**.
- O compilador escolhe automaticamente **qual método chamar com base nos parâmetros passados**.

📌 Vantagens da Sobrecarga:

✔ Permite criar várias versões do mesmo método.
✔ Melhora a **legibilidade e organização** do código.
✔ **Evita nomes de métodos diferentes** para funções semelhantes.

📌 12.2 - Polimorfismo de Sobrescrita (Method Overriding)

Ocorre quando um método **de uma superclasse é redefinido (sobrescrito) em uma subclasse**.

💡 É resolvido em tempo de execução.

✅ Exemplo de Sobrescrita de Método

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

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

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

public class Main {
    public static void main(String[] args) {
        Animal meuAnimal = new Cachorro();
        meuAnimal.fazerSom();  // Chama o método sobrescrito de Cachorro

        meuAnimal = new Gato();
        meuAnimal.fazerSom();  // Chama o método sobrescrito de Gato
    }
}
```

🔷 Explicação:

- A superclasse Animal tem um método fazerSom().
- Cachorro e Gato sobrescrevem esse método com sua própria implementação.
- **Mesmo que meuAnimal seja do tipo Animal, ele se comporta como Cachorro ou Gato em tempo de execução.**

📌 Regras da Sobrescrita:

✔ O método na subclasse **deve ter o mesmo nome, tipo de retorno e parâmetros**.

✔ Deve ter a mesma ou **maior visibilidade** (exemplo: protected → public).

✔ Métodos private e static **não podem ser sobrescritos**.

📌 12.3 - Polimorfismo com Classes Abstratas
Se uma classe contém **métodos abstratos**, as subclasses devem sobrescrevê-los.

✅ Exemplo com Classe Abstrata
```
abstract class Animal {
    abstract void fazerSom();  // Método abstrato (sem implementação)
}

class Cavalo extends Animal {
    @Override
    public void fazerSom() {
        System.out.println("O cavalo relincha: Iiirrrrí!");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal meuCavalo = new Cavalo();
        meuCavalo.fazerSom(); // Chama a implementação de Cavalo
    }
}
```

🔷 Explicação:
- Animal é uma classe abstrata, então **não pode ser instanciada diretamente**.
- Cavalo **é obrigado a implementar** o método fazerSom().

📌 12.4 - Polimorfismo com Interfaces
Em Java, uma **interface** define um conjunto de métodos que as classes devem implementar.
O polimorfismo permite que um mesmo método tenha diferentes implementações em classes distintas.

✅ Exemplo com Interface
```
interface Forma {
    void desenhar();
}

class Circulo implements Forma {
    @Override
    public void desenhar() {
        System.out.println("Desenhando um Círculo.");
    }
}

class Quadrado implements Forma {
    @Override
    public void desenhar() {
        System.out.println("Desenhando um Quadrado.");
    }
}

public class Main {
    public static void main(String[] args) {
        Forma minhaForma = new Circulo();
        minhaForma.desenhar();  // Chama o método desenhar() de Circulo

        minhaForma = new Quadrado();
        minhaForma.desenhar();  // Chama o método desenhar() de Quadrado
    }
}
```

🔹 **Explicação:**

- Forma é uma interface com o método desenhar().
- Circulo e Quadrado implementam essa interface com suas versões do método.
- **A variável minhaForma pode referenciar diferentes implementações.**

📌 **12.5 - Benefícios do Polimorfismo**

✅ **Código mais flexível** → Podemos trocar implementações facilmente.
✅ **Reutilização** → Métodos podem ser reaproveitados em diferentes classes.
✅ **Extensibilidade** → Podemos adicionar novas funcionalidades sem modificar código existente.
✅ **Facilita manutenção** → Torna o código mais organizado e compreensível.

📌 **Conclusão**

O **polimorfismo** é essencial para criar programas modulares e reutilizáveis.

💡 **Resumo:**

- **Polimorfismo de Sobrecarga (Overloading)** → Mesmo nome, **diferentes assinaturas** (resolvido em tempo de compilação).
- **Polimorfismo de Sobrescrita (Overriding)** → Mesmo nome, **mesma assinatura, mas implementação diferente** (resolvido em tempo de execução).
- **Pode ser aplicado em classes, classes abstratas e interfaces.**

O **polimorfismo permite que um código seja reutilizado sem perder flexibilidade**, tornando o design do sistema mais **organizado, escalável e fácil de manter!** 🚀

📌 13. Classe Abstrata

A **classe abstrata** é um conceito fundamental da Programação Orientada a Objetos (POO) que permite a criação de uma estrutura de código mais flexível e reutilizável. Ela serve como um **modelo base para outras classes**, mas **não pode ser instanciada diretamente**. Seu principal objetivo é definir **um comportamento comum** para classes derivadas.

🎯 O que é uma Classe Abstrata?

Uma **classe abstrata** é uma classe que não pode ser instanciada diretamente e pode conter **métodos abstratos** (sem implementação) e métodos concretos (com implementação).
Ela **força as subclasses** a fornecerem implementações específicas para seus métodos abstratos.

📌 Regras das Classes Abstratas em Java

- ✔ Pode conter métodos **abstratos** (sem corpo) e métodos **concretos** (com corpo).
- ✔ **Não pode ser instanciada diretamente**.
- ✔ **Pode ter construtores**, mas apenas para ser chamado pelas subclasses.
- ✔ Pode conter **atributos e métodos comuns** para todas as subclasses.
- ✔ **Pode implementar interfaces e ter métodos estáticos**.

📌 13.1 - Exemplo Simples de Classe Abstrata

```
abstract class Animal {
    String nome;

    public Animal(String nome) {
        this.nome = nome;
    }

    // Método abstrato (sem implementação)
    abstract void emitirSom();

    // Método concreto (com implementação)
    public void dormir() {
        System.out.println(nome + " está dormindo.");
    }
}

class Cachorro extends Animal {
    public Cachorro(String nome) {
        super(nome);
    }

    // Implementação do método abstrato
    @Override
    void emitirSom() {
        System.out.println(nome + " late: Au Au!");
    }
}

public class Main {
    public static void main(String[] args) {
        // Animal animal = new Animal("Genérico"); // ERRO: não pode instanciar uma classe abstrata
        Animal meuCachorro = new Cachorro("Rex");
        meuCachorro.emitirSom(); // Saída: Rex late: Au Au!
        meuCachorro.dormir();    // Saída: Rex está dormindo.
    }
}
```

🔷 **Explicação do código:**

- A classe Animal é **abstrata** e contém um método abstrato emitirSom().
- Cachorro **herda de Animal** e implementa emitirSom().
- A classe Animal **não pode ser instanciada**, mas podemos usar uma variável do tipo Animal para armazenar um objeto Cachorro.
- **Isso permite o uso do polimorfismo**, onde podemos chamar emitirSom() sem precisar saber exatamente qual classe concreta está sendo usada.

📌 13.2 - Métodos Abstratos em uma Classe Abstrata

Um **método abstrato** é um método que **não possui implementação na classe abstrata** e deve ser implementado por qualquer subclasse concreta.

```
abstract class Veiculo {
    abstract void acelerar();
}

class Carro extends Veiculo {
    @Override
    void acelerar() {
        System.out.println("O carro acelera a 100km/h.");
    }
}

class Moto extends Veiculo {
    @Override
    void acelerar() {
        System.out.println("A moto acelera a 80km/h.");
    }
}

public class Main {
    public static void main(String[] args) {
        Veiculo meuCarro = new Carro();
        meuCarro.acelerar(); // Saída: O carro acelera a 100km/h.

        Veiculo minhaMoto = new Moto();
        minhaMoto.acelerar(); // Saída: A moto acelera a 80km/h.
    }
}
```

📌 **Regra importante:**
Se uma subclasse **não implementar** um método abstrato da superclasse, ela também **deve ser declarada como abstrata**.

📌 13.3 - Classes Abstratas com Métodos Concretos

Além de métodos abstratos, uma classe abstrata pode conter métodos concretos (com implementação).
Isso é útil quando queremos fornecer uma implementação padrão que pode ser **usada diretamente ou sobrescrita pelas subclasses**.

```
abstract class Computador {
    abstract void ligar();

    // Método concreto
    public void exibirMarca() {
        System.out.println("Este computador é da marca Genérica.");
    }
}

class Notebook extends Computador {
    @Override
    void ligar() {
        System.out.println("O notebook está ligando...");
    }
}

public class Main {
    public static void main(String[] args) {
        Notebook meuNotebook = new Notebook();
        meuNotebook.ligar();        // Saída: O notebook está ligando...
        meuNotebook.exibirMarca();  // Saída: Este computador é da marca Genérica.
    }
}
```

📌 13.4 - Classe Abstrata vs Interface

Característica	Classe Abstrata	Interface
Métodos concretos	✅ Sim	❌ Não (apenas a partir do Java 8 com métodos default)
Métodos abstratos	✅ Sim	✅ Sim
Atributos	✅ Sim (com modificadores de acesso)	❌ Apenas constantes (static final)
Herança múltipla	❌ Não	✅ Sim (uma classe pode implementar várias interfaces)
Construtor	✅ Sim	❌ Não

📌 **Quando usar cada um?**

✔ **Use classes abstratas** quando houver uma **relação hierárquica clara** (exemplo: Animal → Cachorro).

✔ **Use interfaces** quando houver **múltiplos comportamentos** que podem ser compartilhados entre classes sem relação direta.

📌 13.5 - Exemplo Prático: Sistema de Pagamentos

Vamos supor que estamos construindo um sistema de pagamentos que pode ter **Cartão de Crédito** e **Pix**.

```
abstract class Pagamento {
    double valor;

    public Pagamento(double valor) {
        this.valor = valor;
    }

    abstract void realizarPagamento();

    public void imprimirRecibo() {
        System.out.println("Pagamento de R$" + valor + " realizado com sucesso.");
    }
}

class PagamentoCartao extends Pagamento {
    public PagamentoCartao(double valor) {
        super(valor);
    }

    @Override
    void realizarPagamento() {
        System.out.println("Pagamento via Cartão de Crédito: R$" + valor);
    }
}

class PagamentoPix extends Pagamento {
    public PagamentoPix(double valor) {
        super(valor);
    }

    @Override
    void realizarPagamento() {
        System.out.println("Pagamento via Pix: R$" + valor);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Pagamento pagamento1 = new PagamentoCartao(100);
        pagamento1.realizarPagamento();
        pagamento1.imprimirRecibo();

        Pagamento pagamento2 = new PagamentoPix(50);
        pagamento2.realizarPagamento();
        pagamento2.imprimirRecibo();
    }
}
```

🔷 **Explicação do código:**

- Pagamento é uma **classe abstrata** que define um método abstrato realizarPagamento().
- PagamentoCartao e PagamentoPix são **subclasses concretas** que fornecem implementações específicas.
- O método imprimirRecibo() é **compartilhado entre todas as subclasses**.
- O uso do **polimorfismo** permite que pagamento1 e pagamento2 sejam tratados de forma genérica, sem precisar saber exatamente qual método de pagamento está sendo usado.

📌 **Conclusão**

As classes abstratas são uma ferramenta poderosa para estruturar e reutilizar código em Java.

📌 **Principais pontos:**
✔ **Forçam as subclasses a implementar métodos específicos.**
✔ **Permitem compartilhar atributos e métodos comuns.**
✔ **Ajudam a organizar o código seguindo um modelo hierárquico.**
✔ **São ideais quando há um comportamento base, mas a implementação varia.**

🎯 **Em resumo:** Se você precisa de um esqueleto comum para várias classes, mas quer que elas implementem seus próprios comportamentos específicos, **use classes abstratas!** 🚀

Fala, Dev! Parabéns por ter chegado até nos estudos. Espero que esse material te faça crescer como profissional e você atinja seus objetivos. Aproveito o momento e o convido para conhecer minhas redes sociais onde terá acesso a mais conteúdos como esse:

Conteúdo desenvolvido por: Olival Paulino
Canal do Youtube: https://youtube.com/@olivalpaulino
Instagram: https://www.instagram.com/olivalpaulino/

Aproveito também e deixo o link de todos os meus cursos, para que você os conheça e possa acelerar ainda mais os seus estudos, seja na teoria, na prática, criando projetos, e aplicando todos os conceitos principais na prática, seja criando aplicações desktop ou web, ou ainda, aplicações back-end ou full stack java. Conheça-os abaixo:

- Java na Prática – Do Básico ao Avançado: https://pay.kiwify.com.br/55BuhNc
- Projeto Java Full Stack, Sistema de Lanchonete: https://pay.kiwify.com.br/V8tIuJ3
- Projeto Java Back-end, Sistema de Gestão de Atendimentos: https://pay.kiwify.com.br/0vmjNYG
- Java na Web com Spring Boot: https://pay.kiwify.com.br/meoh7va
- Java no Front-end para Desktop: https://pay.kiwify.com.br/TxhrQlt
- Java no Front-end para Web com Spring Boot: https://pay.kiwify.com.br/QcDb7zc

Sucesso nos estudos e carreira profissional.