FUNDAMENTOS DE PROGRAMAÇÃO ORIENTADA A OBJETOS

Material de Estudo Sequencial - Java

📚 GUIA DE ESTUDO

Este material foi estruturado de forma **progressiva** e **sequencial**. Cada tópico constrói sobre o conhecimento anterior.

📖 **Como usar este material:**

1. **Leia na ordem apresentada** - cada conceito depende do anterior
2. **Pratique os exemplos** - digite e execute os códigos
3. **Teste seu entendimento** - faça os exercícios mentais
4. **Conecte os conceitos** - veja como tudo se relaciona

🏛️ OS 4 PILARES DA PROGRAMAÇÃO ORIENTADA A OBJETOS

Antes de mergulharmos nos conceitos específicos, é fundamental entender que a **Programação Orientada a Objetos** é sustentada por **4 pilares fundamentais**. Estes pilares trabalham juntos para criar um paradigma de programação poderoso e flexível.

🔒 1. ENCAPSULAMENTO

"Esconder os detalhes internos e fornecer uma interface controlada"

- **O que é:** Proteção dos dados internos de um objeto
- **Como:** Atributos `private` + métodos `public` (getters/setters)
- **Por que:** Segurança, validação, controle de acesso
- **Analogia:** Caixa-forte - você não vê o mecanismo interno, apenas usa a interface (teclado, tela)

```
// Exemplo de encapsulamento
public class ContaBancaria {
    private double saldo; // PROTEGIDO

    public void depositar(double valor) { // INTERFACE PÚBLICA
        if (valor > 0) {
            saldo += valor; // CONTROLE INTERNO
        }
    }
}
```

👨‍👩‍👧‍👦 2. HERANÇA

"Reutilizar código através do relacionamento 'é um tipo de'"

- **O que é:** Uma classe filha herda características da classe pai
- **Como:** Palavra-chave `extends` e `super`
- **Por que:** Reutilização de código, hierarquias naturais
- **Analogia:** DNA - filhos herdam características dos pais, mas podem ter particularidades

```
// Exemplo de herança
public class Animal {
    protected String nome;
    public void dormir() { /* comportamento comum */ }
}

public class Cachorro extends Animal { // HERDA de Animal
    public void latir() { /* comportamento específico */ }
}
```

🎭 3. POLIMORFISMO

"Objetos diferentes respondem à mesma mensagem de formas específicas"

- **O que é:** Mesmo método, comportamentos diferentes
- **Como:** Sobrescrita (`@Override`) e referências da superclasse
- **Por que:** Flexibilidade, extensibilidade, código genérico
- **Analogia:** Controle remoto - o botão "play" funciona em TV, DVD, som, cada um faz sua ação específica

```
// Exemplo de polimorfismo
Animal animal1 = new Cachorro();
Animal animal2 = new Gato();

animal1.fazerSom(); // "Au au au!"
animal2.fazerSom(); // "Miau!"
// Mesmo método, comportamentos diferentes!
```

🎨 4. ABSTRAÇÃO

"Focar no essencial, ignorar detalhes desnecessários"

- **O que é:** Simplificação da realidade, modelando apenas o que importa
- **Como:** Classes abstratas, interfaces, modelagem conceitual
- **Por que:** Simplicidade, foco no essencial, design limpo
- **Analogia:** Mapa do metrô - mostra apenas o necessário (estações, conexões), ignora detalhes geográficos

```java
// Exemplo de abstração
public abstract class Veiculo { // ABSTRATA - conceito geral
    protected String marca;

    // Método abstrato - cada veículo implementa diferente
    public abstract void acelerar();

    // Método concreto - comum a todos
    public void ligar() { System.out.println("Veículo ligado"); }
}

public class Carro extends Veiculo {
    @Override
    public void acelerar() {
        System.out.println("Pisando no acelerador");
    }
}
```

🤝 COMO OS PILARES TRABALHAM JUNTOS

Exemplo integrado - Sistema de Funcionários:

```java
// ABSTRAÇÃO - Modelo conceitual
public abstract class Funcionario {
    // ENCAPSULAMENTO - Dados protegidos
    private String nome;
    protected double salarioBase;

    // ABSTRAÇÃO - Método que cada tipo deve implementar
    public abstract double calcularSalario();

    // ENCAPSULAMENTO - Acesso controlado
    public String getNome() { return nome; }
    public void setNome(String nome) { this.nome = nome; }
}

// HERANÇA - Vendedor É UM TIPO DE Funcionário
public class Vendedor extends Funcionario {
    private double comissao;

    // POLIMORFISMO - Implementação específica
    @Override
    public double calcularSalario() {
        return salarioBase + comissao;
    }
}

// HERANÇA - Gerente É UM TIPO DE Funcionário
public class Gerente extends Funcionario {
    private double bonus;
```

```
    // POLIMORFISMO - Implementação específica
    @Override
    public double calcularSalario() {
        return salarioBase + bonus;
    }
}

// POLIMORFISMO em ação - código que funciona com qualquer tipo
public class FolhaPagamento {
    public void processarFolha(List<Funcionario> funcionarios) {
        for (Funcionario f : funcionarios) {
            // Cada tipo executa sua versão de calcularSalario()
            System.out.println(f.getNome() + ": R$ " +
f.calcularSalario());
        }
    }
}
```

🎯 BENEFÍCIOS DOS 4 PILARES JUNTOS:

1. 🔒 **Encapsulamento** → **Segurança** e **controle**
2. 👨‍👩‍👧‍👦 **Herança** → **Reutilização** e **organização**
3. 🎭 **Polimorfismo** → **Flexibilidade** e **extensibilidade**
4. 🎨 **Abstração** → **Simplicidade** e **foco no essencial**

Resultado: Código **mais organizado**, **fácil de manter**, **reutilizável** e **flexível**!

1️⃣ NÍVEL INICIANTE: CONCEITOS FUNDAMENTAIS

🏗️ O que é Programação Orientada a Objetos?

A **Programação Orientada a Objetos (POO)** é uma forma de programar que imita o mundo real. Ao invés de escrever código linear, criamos "objetos" que representam coisas reais.

Analogia simples:

- O mundo real tem **pessoas, carros, casas, animais**
- Na POO, criamos **objetos** que representam essas entidades
- Cada objeto tem **características** (cor, tamanho) e **comportamentos** (andar, acelerar)

Por que POO é importante?

- **Organização:** Código mais estruturado e fácil de entender
- **Reutilização:** Podemos usar o mesmo código em diferentes partes
- **Manutenção:** Mudanças são mais fáceis de fazer
- **Colaboração:** Diferentes programadores podem trabalhar juntos

---

🎯 Conceito #1: Classes vs Objetos

🔍 **A diferença mais importante da POO:**

CLASSE = MOLDE/PLANTA

Uma classe é como uma **planta arquitetônica** que define:

- Que características um objeto terá (atributos)
- Que ações um objeto poderá fazer (métodos)

OBJETO = INSTÂNCIA/COISA REAL

Um objeto é uma **casa específica** construída seguindo a planta:

- Cada casa é única, mas segue o mesmo projeto
- Pode ter valores diferentes para cada característica

Exemplo prático:

```
// CLASSE - O molde/planta
class Pessoa {
    // Características (atributos)
    String nome;
    int idade;

    // Comportamentos (métodos)
    void falar() {
        System.out.println(nome + " está falando");
    }
}

// OBJETOS - As instâncias reais
public class TesteClasse {
    public static void main(String[] args) {
        // Criando objetos (instâncias) da classe Pessoa
        Pessoa joao = new Pessoa();    // João é um objeto
        Pessoa maria = new Pessoa();   // Maria é outro objeto

        // Definindo características específicas
        joao.nome = "João";
        joao.idade = 25;

        maria.nome = "Maria";
        maria.idade = 30;

        // Chamando comportamentos
        joao.falar();   // Saída: João está falando
        maria.falar();  // Saída: Maria está falando
    }
}
```

🧠 **Exercício mental:**

- Quantos objetos `Pessoa` posso criar? **R: Infinitos**
- Se mudar a classe, o que acontece com os objetos? **R: Todos são afetados**
- `joao` e `maria` são iguais? **R: Não, são objetos diferentes com valores diferentes**

🔧 Conceito #2: Atributos vs Métodos

ATRIBUTOS = CARACTERÍSTICAS/DADOS

São as **propriedades** que descrevem o objeto:

- Armazenam **informações** sobre o objeto
- Representam o **estado** atual do objeto
- **Exemplos:** nome, idade, cor, tamanho, preço

MÉTODOS = COMPORTAMENTOS/AÇÕES

São as **funções** que o objeto pode executar:

- Definem **o que o objeto pode fazer**
- Podem usar e modificar os atributos
- **Exemplos:** andar(), falar(), acelerar(), calcular()

Exemplo detalhado:

```
class ContaBancaria {
    // ATRIBUTOS - características/dados
    String titular;
    double saldo;
    int numero;

    // MÉTODOS - comportamentos/ações
    void depositar(double valor) {
        saldo = saldo + valor;  // Modifica o atributo
        System.out.println("Depósito realizado. Saldo atual: " + saldo);
    }

    void sacar(double valor) {
        saldo = saldo - valor;  // Modifica o atributo
        System.out.println("Saque realizado. Saldo atual: " + saldo);
    }

    void exibirInfo() {
        System.out.println("Conta: " + numero +
                           " | Titular: " + titular +
                           " | Saldo: " + saldo);
    }
}

// Usando a classe
public class TesteConta {
```

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

        // Definindo atributos
        conta.titular = "João Silva";
        conta.numero = 12345;
        conta.saldo = 1000.0;

        // Usando métodos
        conta.exibirInfo();      // Mostra informações
        conta.depositar(500.0);  // Adiciona dinheiro
        conta.sacar(200.0);      // Remove dinheiro
        conta.exibirInfo();      // Mostra estado atual
    }
}
```

🧠 **Teste seu entendimento:**

- Na classe `ContaBancaria`, quais são os atributos?
- Quais são os métodos?
- O que acontece quando chamo `depositar(100)`?

🆕 Conceito #3: A palavra-chave `new`

Para criar um objeto (instância) de uma classe, usamos a palavra `new`:

```
// Sintaxe: TipoDaClasse nomeDoObjeto = new TipoDaClasse();
Pessoa joao = new Pessoa();
ContaBancaria conta = new ContaBancaria();
```

O que `new` faz:

1. **Reserva espaço na memória** para o objeto
2. **Chama o construtor** da classe (veremos mais tarde)
3. **Retorna a referência** para o objeto criado

Analogia: `new` é como contratar uma construtora para construir uma casa seguindo sua planta arquitetônica.

2️⃣ NÍVEL BÁSICO: ENCAPSULAMENTO E PROTEÇÃO

🔐 Conceito #4: Encapsulamento - Protegendo os Dados

Problema: E se alguém fizer isso?

```
conta.saldo = -1000;  // Saldo negativo inválido!
conta.titular = "";   // Nome vazio!
```

Solução: ENCAPSULAMENTO - esconder os detalhes internos e controlar o acesso.

Modificadores de Acesso:

- `private` - Apenas a própria classe pode acessar
- `public` - Qualquer classe pode acessar
- `protected` - Apenas classes do mesmo pacote e subclasses

Exemplo com encapsulamento:

```
class ContaBancariaSegura {
    // ATRIBUTOS PRIVADOS - protegidos
    private String titular;
    private double saldo;
    private int numero;

    // MÉTODOS PÚBLICOS - interface controlada
    public void depositar(double valor) {
        if (valor > 0) {  // VALIDAÇÃO
            saldo += valor;
            System.out.println("Depósito de R$ " + valor + " realizado");
        } else {
            System.out.println("Valor deve ser positivo!");
        }
    }

    public void sacar(double valor) {
        if (valor > 0 && valor <= saldo) {  // VALIDAÇÃO
            saldo -= valor;
            System.out.println("Saque de R$ " + valor + " realizado");
        } else {
            System.out.println("Saque inválido!");
        }
    }

    // Método para definir titular com validação
    public void setTitular(String nome) {
        if (nome != null && !nome.trim().isEmpty()) {
            this.titular = nome;
        } else {
            System.out.println("Nome não pode ser vazio!");
        }
    }
}
```

🎯 **Benefícios do Encapsulamento:**

- **Proteção:** Dados não podem ser alterados incorretamente
- **Validação:** Podemos verificar se os valores são válidos

- **Controle:** Decidimos como e quando os dados podem ser acessados
- **Manutenção:** Mudanças internas não afetam o código externo

---

🔄 Conceito #5: Getters e Setters

Getters = Métodos para **LER** (get = obter) **Setters** = Métodos para **ESCREVER** (set = definir)

```
class Produto {
    private String nome;
    private double preco;

    // GETTER - para ler o nome
    public String getNome() {
        return nome;
    }

    // SETTER - para definir o nome
    public void setNome(String nome) {
        if (nome != null && !nome.trim().isEmpty()) {
            this.nome = nome;
        } else {
            System.out.println("Nome inválido!");
        }
    }

    // GETTER - para ler o preço
    public double getPreco() {
        return preco;
    }

    // SETTER - para definir o preço
    public void setPreco(double preco) {
        if (preco >= 0) {
            this.preco = preco;
        } else {
            System.out.println("Preço não pode ser negativo!");
        }
    }
}

// Usando getters e setters
public class TesteProduto {
    public static void main(String[] args) {
        Produto p = new Produto();

        // Usando setters para definir valores
        p.setNome("Notebook");
        p.setPreco(2500.0);

        // Usando getters para ler valores
        System.out.println("Produto: " + p.getNome());
```

```
        System.out.println("Preço: R$ " + p.getPreco());

        // Tentativa inválida - será rejeitada
        p.setPreco(-100);  // Saída: Preço não pode ser negativo!
    }
}
```

🔍 **Padrão de Nomenclatura:**

- **Getter:** `get` + nome do atributo (primeira letra maiúscula)
- **Setter:** `set` + nome do atributo (primeira letra maiúscula)
- **Boolean:** `is` + nome do atributo (ex: `isAtivo()`)

🏗️ Conceito #6: Construtores

Construtor = Método especial que **inicializa** o objeto quando ele é criado.

Características dos construtores:

- **Mesmo nome da classe**
- **Não tem tipo de retorno** (nem void)
- **Chamado automaticamente** quando usamos `new`

```
class Veiculo {
    private String marca;
    private String modelo;
    private int ano;

    // CONSTRUTOR PADRÃO (sem parâmetros)
    public Veiculo() {
        this.marca = "Não informada";
        this.modelo = "Não informado";
        this.ano = 2020;
        System.out.println("Veículo criado com valores padrão");
    }

    // CONSTRUTOR PARAMETRIZADO
    public Veiculo(String marca, String modelo, int ano) {
        setMarca(marca);    // Usa setter para validar
        setModelo(modelo);  // Usa setter para validar
        setAno(ano);        // Usa setter para validar
        System.out.println("Veículo criado: " + marca + " " + modelo);
    }

    // Setters com validação
    public void setMarca(String marca) {
        if (marca != null && !marca.trim().isEmpty()) {
            this.marca = marca;
        } else {
            this.marca = "Marca inválida";
```

```
        }
    }

    public void setModelo(String modelo) {
        if (modelo != null && !modelo.trim().isEmpty()) {
            this.modelo = modelo;
        } else {
            this.modelo = "Modelo inválido";
        }
    }

    public void setAno(int ano) {
        if (ano >= 1900 && ano <= 2024) {
            this.ano = ano;
        } else {
            this.ano = 2020;  // Valor padrão
        }
    }

    // Getters
    public String getMarca() { return marca; }
    public String getModelo() { return modelo; }
    public int getAno() { return ano; }
}

// Usando construtores
public class TesteVeiculo {
    public static void main(String[] args) {
        // Usando construtor padrão
        Veiculo v1 = new Veiculo();

        // Usando construtor parametrizado
        Veiculo v2 = new Veiculo("Toyota", "Corolla", 2022);

        System.out.println(v1.getMarca() + " " + v1.getModelo());
        System.out.println(v2.getMarca() + " " + v2.getModelo());
    }
}
```

🎯 **Sobrecarga de Construtores:**

- Podemos ter **múltiplos construtores** na mesma classe
- Devem ter **parâmetros diferentes**
- Java escolhe qual usar baseado nos argumentos fornecidos

🔗 Conceito #7: A palavra `this`

`this` = Referência ao **próprio objeto** que está executando o método.

Quando usar `this`:

1. **Resolver ambiguidade** entre parâmetro e atributo

2. **Chamar outro construtor** da mesma classe
3. **Deixar código mais claro**

```
class Funcionario {
    private String nome;
    private double salario;

    public void setNome(String nome) {
        // SEM this - ERRO! Java não sabe qual 'nome' você quer
        // nome = nome;  // ❌ Não funciona

        // COM this - CORRETO!
        this.nome = nome;  // ✅ this.nome = atributo, nome = parâmetro
    }

    public void setSalario(double salario) {
        if (salario > 0) {
            this.salario = salario;  // this deixa claro que é o atributo
        }
    }

    // Usando this para chamar outro construtor
    public Funcionario() {
        this("Nome não informado", 0.0);  // Chama o construtor
parametrizado
    }

    public Funcionario(String nome, double salario) {
        this.nome = nome;
        this.salario = salario;
    }

    public String getNome() { return nome; }
    public double getSalario() { return salario; }
}
```

💡 **Dica:** Use `this` sempre que houver **dúvida** entre atributo e parâmetro!

3️⃣ NÍVEL INTERMEDIÁRIO: HERANÇA E RELACIONAMENTOS

👨‍👩‍👧‍👦 Conceito #8: Herança - "É um tipo de"

Herança permite que uma classe **filha** herde características e comportamentos de uma classe **pai**.

Relação: `is-a` (é um tipo de)

- Gerente **é um tipo de** Funcionário
- Carro **é um tipo de** Veículo
- Gato **é um tipo de** Animal

```java
// SUPERCLASSE (classe pai)
class Animal {
    protected String nome;    // protected = filhos podem acessar
    protected int idade;

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

    public void dormir() {
        System.out.println(nome + " está dormindo...");
    }

    public void comer() {
        System.out.println(nome + " está comendo...");
    }

    // Getters
    public String getNome() { return nome; }
    public int getIdade() { return idade; }
}

// SUBCLASSE (classe filha)
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 au!");
    }

    public String getRaca() { return raca; }
}

// OUTRA SUBCLASSE
class Gato extends Animal {
    private boolean temPelo;

    public Gato(String nome, int idade, boolean temPelo) {
        super(nome, idade);
        this.temPelo = temPelo;
    }

    // Método específico de Gato
    public void miar() {
        System.out.println(nome + " está miando: Miau!");
    }
```

```
    public boolean isTemPelo() { return temPelo; }
}

// Testando herança
public class TesteHeranca {
    public static void main(String[] args) {
        Cachorro dog = new Cachorro("Rex", 5, "Labrador");
        Gato cat = new Gato("Mimi", 3, true);

        // Métodos herdados de Animal
        dog.dormir();  // Funciona!
        dog.comer();   // Funciona!
        cat.dormir();  // Funciona!
        cat.comer();   // Funciona!

        // Métodos específicos
        dog.latir();   // Só Cachorro tem
        cat.miar();    // Só Gato tem

        System.out.println("Cachorro: " + dog.getNome() + ", " +
dog.getRaca());
        System.out.println("Gato: " + cat.getNome() + ", peludo: " +
cat.isTemPelo());
    }
}
```

🔑 **Palavras-chave da Herança:**

- `extends` - Define que uma classe herda de outra
- `super` - Acessa membros da superclasse
- `protected` - Visível para subclasses

🔄 Conceito #9: Sobrescrita (Override)

Sobrescrita permite que a classe filha **redefina** o comportamento de um método herdado.

```
class Veiculo {
    protected String marca;
    protected int velocidade;

    public Veiculo(String marca) {
        this.marca = marca;
        this.velocidade = 0;
    }

    public void acelerar() {
        velocidade += 10;
        System.out.println(marca + " acelerou para " + velocidade + "
km/h");
```

```
    }

    public void exibirInfo() {
        System.out.println("Veículo: " + marca + " - Velocidade: " +
velocidade);
    }
}

class Carro extends Veiculo {
    private int numeroPortas;

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

    // SOBRESCRITA - comportamento específico de Carro
    @Override
    public void acelerar() {
        velocidade += 20;  // Carro acelera mais que veículo genérico
        System.out.println("Carro " + marca + " acelerou para " +
velocidade + " km/h");
    }

    @Override
    public void exibirInfo() {
        super.exibirInfo();  // Chama método da superclasse primeiro
        System.out.println("Número de portas: " + numeroPortas);
    }
}

class Bicicleta extends Veiculo {
    private boolean temCestinha;

    public Bicicleta(String marca, boolean temCestinha) {
        super(marca);
        this.temCestinha = temCestinha;
    }

    // SOBRESCRITA - comportamento específico de Bicicleta
    @Override
    public void acelerar() {
        velocidade += 5;  // Bicicleta acelera menos
        System.out.println("Bicicleta " + marca + " pedalou para " +
velocidade + " km/h");
    }

    @Override
    public void exibirInfo() {
        super.exibirInfo();
        System.out.println("Tem cestinha: " + (temCestinha ? "Sim" :
"Não"));
    }
}
```

```
// Testando sobrescrita
public class TesteSobrescrita {
    public static void main(String[] args) {
        Veiculo veiculo = new Veiculo("Genérico");
        Carro carro = new Carro("Honda Civic", 4);
        Bicicleta bike = new Bicicleta("Caloi", true);

        // Cada um acelera de forma diferente!
        veiculo.acelerar();  // +10 km/h
        carro.acelerar();    // +20 km/h
        bike.acelerar();     // +5 km/h

        System.out.println("\n--- Informações ---");
        veiculo.exibirInfo();
        carro.exibirInfo();
        bike.exibirInfo();
    }
}
```

🎯 **Anotação @Override:**

- **Não é obrigatória**, mas é **boa prática**
- **Documenta** que é sobrescrita intencional
- **Compilador verifica** se realmente está sobrescrevendo
- **Previne erros** de digitação no nome do método

4️⃣ NÍVEL AVANÇADO: POLIMORFISMO E RELACIONAMENTOS

🎭 Conceito #10: Polimorfismo - "Muitas Formas"

Polimorfismo permite que objetos diferentes respondam à mesma mensagem de formas específicas.

Ideia central: "Fale para todos os animais fazerem som, cada um fará seu som específico"

```
class Funcionario {
    protected String nome;
    protected double salarioBase;

    public Funcionario(String nome, double salarioBase) {
        this.nome = nome;
        this.salarioBase = salarioBase;
    }

    // Método que será sobrescrito
    public double calcularSalario() {
        return salarioBase;
    }

    public void exibirInfo() {
```

```
        System.out.println("Funcionário: " + nome +
                           " | Salário: R$ " + calcularSalario());
    }

    public String getNome() { return nome; }
}

class Vendedor extends Funcionario {
    private double comissao;
    private double vendas;

    public Vendedor(String nome, double salarioBase, double comissao) {
        super(nome, salarioBase);
        this.comissao = comissao;
        this.vendas = 0;
    }

    public void registrarVenda(double valor) {
        this.vendas += valor;
    }

    @Override
    public double calcularSalario() {
        return salarioBase + (vendas * comissao / 100);
    }
}

class Gerente extends Funcionario {
    private double bonus;

    public Gerente(String nome, double salarioBase, double bonus) {
        super(nome, salarioBase);
        this.bonus = bonus;
    }

    @Override
    public double calcularSalario() {
        return salarioBase + bonus;
    }
}

// POLIMORFISMO EM AÇÃO!
import java.util.ArrayList;

public class TestePolimorfismo {
    public static void main(String[] args) {
        // Lista de Funcionarios (superclasse)
        ArrayList<Funcionario> funcionarios = new ArrayList<>();

        // Adicionando diferentes tipos de funcionários
        funcionarios.add(new Funcionario("João", 3000));
        funcionarios.add(new Vendedor("Maria", 2500, 5));  // 5% comissão
        funcionarios.add(new Gerente("Pedro", 4000, 1500));
```

```
        // Registrando vendas para o vendedor
        ((Vendedor) funcionarios.get(1)).registrarVenda(10000);  // R$
10.000 em vendas

        System.out.println("=== FOLHA DE PAGAMENTO ===");

        // POLIMORFISMO: cada objeto executa SEU próprio calcularSalario()
        for (Funcionario f : funcionarios) {
            f.exibirInfo();  // Chama calcularSalario() específico de cada
tipo
        }

        /* Saída esperada:
         * Funcionário: João | Salário: R$ 3000.0
         * Funcionário: Maria | Salário: R$ 3000.0  (2500 + 500 de
comissão)
         * Funcionário: Pedro | Salário: R$ 5500.0  (4000 + 1500 de bônus)
         */
    }
}
```

🧠 **Como funciona o Polimorfismo:**

1. **Tempo de compilação:** Java vê apenas `Funcionario`
2. **Tempo de execução:** Java descobre o tipo real (`Vendedor`, `Gerente`)
3. **Método correto:** Chama a versão específica de `calcularSalario()`

🎯 **Benefícios:**

- **Flexibilidade:** Código funciona com diferentes tipos
- **Extensibilidade:** Posso adicionar novos tipos sem mudar o código existente
- **Manutenção:** Mudanças localizadas em cada classe

🔗 Conceito #11: Relacionamentos entre Classes

1. Herança ("É um tipo de") - is-a

```
class Gerente extends Funcionario {
    // Gerente É UM TIPO DE Funcionário
}
```

2. Composição ("Tem um/Contém") - has-a

Dependência forte: Se o "todo" morre, as "partes" morrem também.

```
class Carro {
    private Motor motor;        // Carro TEM UM Motor
```

```
    private ArrayList<Roda> rodas;  // Carro TEM Rodas

    public Carro() {
        // Motor é criado junto com o Carro
        this.motor = new Motor("V6", 300);
        this.rodas = new ArrayList<>();

        // Rodas são criadas junto com o Carro
        for (int i = 0; i < 4; i++) {
            rodas.add(new Roda(17));
        }
    }

    // Se o Carro for destruído, Motor e Rodas também são
}

class Motor {
    private String tipo;
    private int potencia;

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

class Roda {
    private int tamanho;

    public Roda(int tamanho) {
        this.tamanho = tamanho;
    }
}
```

3. Agregação ("Usa/Tem") - has-a

Dependência fraca: As partes podem existir independentemente do todo.

```
class Universidade {
    private String nome;
    private ArrayList<Professor> professores;

    public Universidade(String nome) {
        this.nome = nome;
        this.professores = new ArrayList<>();
    }

    public void contratarProfessor(Professor prof) {
        professores.add(prof);  // Professor já existia antes
    }
```

```
    public void demitirProfessor(Professor prof) {
        professores.remove(prof);  // Professor continua existindo
    }
}

class Professor {
    private String nome;
    private String especialidade;

    public Professor(String nome, String especialidade) {
        this.nome = nome;
        this.especialidade = especialidade;
    }

    // Professor pode existir sem Universidade
}

// Teste
public class TesteAgregacao {
    public static void main(String[] args) {
        // Professor existe independentemente
        Professor prof1 = new Professor("Dr. Silva", "Java");
        Professor prof2 = new Professor("Dra. Costa", "Python");

        Universidade uninassau = new Universidade("UNINASSAU");

        // Universidade USA os professores
        uninassau.contratarProfessor(prof1);
        uninassau.contratarProfessor(prof2);

        // Se a universidade fechar, os professores continuam existindo
        // e podem trabalhar em outras universidades
    }
}
```

🔍 **Resumo dos Relacionamentos:**

- **Herança:** Gerente **é um tipo de** Funcionário
- **Composição:** Carro **contém** Motor (dependência forte)
- **Agregação:** Universidade **usa** Professores (dependência fraca)

🛡️ Conceito #12: Modificadores de Acesso Avançados

```
package empresa.modelo;

public class Funcionario {
    // PRIVATE - apenas esta classe
    private String cpf;

    // PROTECTED - esta classe, subclasses e mesmo pacote
```

```
    protected double salario;

    // PUBLIC - qualquer lugar
    public String nome;

    // DEFAULT (sem modificador) - apenas mesmo pacote
    String departamento;
}

package empresa.modelo;  // Mesmo pacote

class Gerente extends Funcionario {
    public void exemplo() {
        // this.cpf = "123";        // ❌ ERRO! private
        this.salario = 5000;        // ✅ OK! protected
        this.nome = "João";         // ✅ OK! public
        this.departamento = "TI";   // ✅ OK! mesmo pacote
    }
}

package empresa.teste;  // Pacote diferente

import empresa.modelo.Funcionario;

class Estagiario extends Funcionario {
    public void exemplo() {
        // this.cpf = "123";        // ❌ ERRO! private
        this.salario = 1000;        // ✅ OK! protected (herança)
        this.nome = "Maria";        // ✅ OK! public
        // this.departamento = "RH"; // ❌ ERRO! não é mesmo pacote
    }
}
```

📊 **Tabela de Visibilidade:**

Modificador	Mesma Classe	Mesmo Pacote	Subclasse	Qualquer Lugar
private	✅	❌	❌	❌
default	✅	✅	❌	❌
protected	✅	✅	✅	❌
public	✅	✅	✅	✅

🎯 SÍNTESE: INTEGRANDO TODOS OS CONCEITOS

🏢 Exemplo Completo: Sistema de Empresa

```
// CLASSE BASE
abstract class Pessoa {
```

```
    protected String nome;
    protected String cpf;

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

    public abstract void exibirInfo();  // Cada tipo implementa diferente

    public String getNome() { return nome; }
    public String getCpf() { return cpf; }
}

// HERANÇA + ENCAPSULAMENTO + POLIMORFISMO
class Funcionario extends Pessoa {
    private double salario;
    private String departamento;

    public Funcionario(String nome, String cpf, double salario, String
departamento) {
        super(nome, cpf);
        setSalario(salario);
        this.departamento = departamento;
    }

    public void setSalario(double salario) {
        if (salario > 0) {
            this.salario = salario;
        } else {
            throw new IllegalArgumentException("Salário deve ser
positivo");
        }
    }

    public double getSalario() { return salario; }
    public String getDepartamento() { return departamento; }

    @Override
    public void exibirInfo() {
        System.out.printf("Funcionário: %s | CPF: %s | Salário: R$ %.2f |
Depto: %s%n",
                          nome, cpf, salario, departamento);
    }
}

class Gerente extends Funcionario {
    private double bonus;
    private ArrayList<Funcionario> subordinados;

    public Gerente(String nome, String cpf, double salario, String
departamento, double bonus) {
        super(nome, cpf, salario, departamento);
        this.bonus = bonus;
```

```
        this.subordinados = new ArrayList<>();
    }

    public void adicionarSubordinado(Funcionario func) {
        subordinados.add(func);
    }

    @Override
    public double getSalario() {
        return super.getSalario() + bonus;  // Salário + bônus
    }

    @Override
    public void exibirInfo() {
        System.out.printf("GERENTE: %s | CPF: %s | Salário Total: R$ %.2f |
Depto: %s | Subordinados: %d%n",
                          nome, cpf, getSalario(), getDepartamento(),
subordinados.size());
    }
}

// COMPOSIÇÃO
class Empresa {
    private String nome;
    private ArrayList<Funcionario> funcionarios;
    private ArrayList<Gerente> gerentes;

    public Empresa(String nome) {
        this.nome = nome;
        this.funcionarios = new ArrayList<>();
        this.gerentes = new ArrayList<>();
    }

    public void contratar(Funcionario func) {
        funcionarios.add(func);

        if (func instanceof Gerente) {
            gerentes.add((Gerente) func);
        }

        System.out.println("Funcionário contratado: " + func.getNome());
    }

    public void exibirFolhaPagamento() {
        System.out.println("=== FOLHA DE PAGAMENTO - " + nome + " ===");

        double total = 0;

        // POLIMORFISMO: cada tipo exibe de forma específica
        for (Funcionario f : funcionarios) {
            f.exibirInfo();
            total += f.getSalario();
        }
```

```
        System.out.printf("TOTAL DA FOLHA: R$ %.2f%n", total);
    }
}

// TESTE COMPLETO
public class SistemaEmpresa {
    public static void main(String[] args) {
        Empresa empresa = new Empresa("TechCorp");

        // Criando funcionários
        Funcionario dev1 = new Funcionario("Ana Silva", "111.111.111-11",
5000, "TI");
        Funcionario dev2 = new Funcionario("Carlos Santos", "222.222.222-
22", 4500, "TI");
        Gerente gerTI = new Gerente("Roberto Lima", "333.333.333-33", 8000,
"TI", 2000);

        // Estabelecendo hierarquia
        gerTI.adicionarSubordinado(dev1);
        gerTI.adicionarSubordinado(dev2);

        // Contratando na empresa
        empresa.contratar(dev1);
        empresa.contratar(dev2);
        empresa.contratar(gerTI);

        // Exibindo resultado
        empresa.exibirFolhaPagamento();

        /* Saída esperada:
         * Funcionário contratado: Ana Silva
         * Funcionário contratado: Carlos Santos
         * Funcionário contratado: Roberto Lima
         * === FOLHA DE PAGAMENTO - TechCorp ===
         * Funcionário: Ana Silva | CPF: 111.111.111-11 | Salário: R$
5000,00 | Depto: TI
         * Funcionário: Carlos Santos | CPF: 222.222.222-22 | Salário: R$
4500,00 | Depto: TI
         * GERENTE: Roberto Lima | CPF: 333.333.333-33 | Salário Total: R$
10000,00 | Depto: TI | Subordinados: 2
         * TOTAL DA FOLHA: R$ 19500,00
         */
    }
}
```

📝 CHECKLIST DE ESTUDO - VERIFIQUE SEU APRENDIZADO

✅ Nível 1 - Fundamentos e os 4 Pilares

- ☐ Sei explicar os 4 pilares da POO (Encapsulamento, Herança, Polimorfismo, Abstração)
- ☐ Sei explicar a diferença entre classe e objeto

- [ ] Sei identificar atributos e métodos em uma classe
- [ ] Sei usar a palavra `new` para criar objetos
- [ ] Entendo o conceito básico de POO

✅ Nível 2 - Pilar do Encapsulamento

- [ ] Sei usar `private` e `public` corretamente
- [ ] Sei criar getters e setters com validação
- [ ] Sei criar construtores (padrão e parametrizado)
- [ ] Entendo quando usar `this`
- [ ] Compreendo como o encapsulamento protege os dados

✅ Nível 3 - Pilar da Herança

- [ ] Sei usar `extends` para criar subclasses
- [ ] Sei usar `super` para acessar a superclasse
- [ ] Sei usar `@Override` para sobrescrever métodos
- [ ] Entendo quando usar `protected`
- [ ] Compreendo relacionamentos "é um tipo de"

✅ Nível 4 - Pilares do Polimorfismo e Abstração

- [ ] Sei implementar polimorfismo em arrays/listas
- [ ] Entendo como abstração simplifica a modelagem
- [ ] Sei usar classes abstratas e métodos abstratos
- [ ] Entendo a diferença entre herança, composição e agregação
- [ ] Sei quando usar cada tipo de relacionamento
- [ ] Consigo criar sistemas integrados usando todos os 4 pilares

---

🎓 DICAS FINAIS PARA O SUCESSO

🔥 Para fixar o conteúdo:

1. **Pratique MUITO** - Digite os códigos, não apenas leia
2. **Crie seus próprios exemplos** - Use contextos familiares (escola, casa, hobby)
3. **Desenhe diagramas** - Visualize as relações entre classes
4. **Ensine alguém** - Explique os conceitos para um colega

🚀 Sequência de estudo recomendada:

1. **Domine os fundamentos** antes de avançar
2. **Um conceito por vez** - não tente aprender tudo junto
3. **Conecte os conceitos** - veja como eles se relacionam
4. **Aplique em projetos** - crie sistemas pequenos mas completos

💡 Principais armadilhas para evitar:

- **Não confundir** classe com objeto
- **Sempre validar** dados nos setters

- **Usar herança** apenas para relacionamentos "é um tipo de"
- **Não esquecer** do @Override na sobrescrita
- **Lembrar** que polimorfismo acontece em tempo de execução

📚 PRÓXIMOS PASSOS

Depois de dominar estes fundamentos, você estará pronto para:

- **Interfaces** e **Classes Abstratas**
- **Coleções** (ArrayList, HashMap, etc.)
- **Tratamento de Exceções**
- **Padrões de Projeto** (Strategy, Observer, etc.)
- **Frameworks** como Spring Boot

Lembre-se: POO é a **base** de quase tudo na programação moderna. Domine estes conceitos e você terá uma base sólida para toda sua carreira!

Material elaborado para facilitar o aprendizado progressivo dos conceitos fundamentais de Programação Orientada a Objetos em Java.

📍 **Versão:** 1.0 | 📅 **Data:** Setembro 2025 | 🎯 **Público:** Estudantes de ADS - 2º Período