3. Herança e Polimorfismo

Vamos explorar dois pilares da programação orientada a objetos: **herança** e **polimorfismo**. Esses conceitos nos permitem reaproveitar código, reduzir redundâncias e tornar o sistema mais flexível e fácil de manter.

3.1. Herança em Java

3.1.1. Conceito de Herança

Herança é o mecanismo pelo qual uma classe pode herdar atributos e métodos de outra classe. Isso promove a **reutilização de código** e estabelece **relações hierárquicas** entre classes.

- A **classe base** ou **superclasse** contém atributos e comportamentos comuns.
- As **subclasses** herdam esses atributos e comportamentos e podem adicionar suas próprias características ou até sobrescrevê-los.

Exemplo no sistema bancário:

- **Superclasse:** `Cliente`
    - Campos comuns: `codigo`, `cidade`, `estado`
- **Subclasses:**
    - `PessoaFisica` (tem `nome` e `cpf`)
    - `PessoaJuridica` (tem `razaoSocial` e `cnpj`)

Com isso:

```
public class Cliente {
    protected int codigo;
    protected String cidade;
    protected String estado;
}

public class PessoaFisica extends Cliente {
    private String nome;
    private String cpf;
```

```
}

public class PessoaJuridica extends Cliente {
    private String razaoSocial;
    private String cnpj;
}
```

3.1.2. Superclasses e Subclasses

Terminologia:

- **Superclasse**: Classe mais genérica (ex: `Cliente`).
- **Subclasse**: Classe especializada que herda da superclasse (ex: `PessoaFisica`, `PessoaJuridica`).

Visibilidade:

- Para que atributos da superclasse possam ser acessados pelas subclasses, eles devem ser marcados como `protected` ou `public`.

Palavra-chave `super`:

- Utilizada para:
    - Acessar métodos ou atributos da superclasse.
    - Invocar o construtor da superclasse a partir da subclasse.

```
public PessoaFisica(String nome, String cpf, String cidade, String estado) {
    super(); // pode incluir parâmetros se o construtor da superclasse tiver
    this.nome = nome;
    this.cpf = cpf;
    this.cidade = cidade;
    this.estado = estado;
}
```

3.1.3. Herança com Conta Corrente e Conta Poupança

Outro ótimo exemplo de herança está na modelagem das contas bancárias:

- **Superclasse:** `Conta`
    - Campos: `numero`, `agencia`, `saldo`, `cliente`
- **Subclasses:**
    - `ContaCorrente`: inclui `limite`, `juros` e o método `aplicarJuros()`
    - `ContaPoupanca`: inclui `dataAniversario`, `rendimento` e o método `aplicarRendimentos()`

```
public class Conta {
    protected int numero;
    protected String agencia;
    protected double saldo;
    protected Cliente cliente;
}

public class ContaCorrente extends Conta {
    private double limite;
    private double juros;

    public void aplicarJuros() {
        saldo += saldo * juros;
    }
}

import java.util.Date;

public class ContaPoupanca extends Conta {
    private Date dataAniversario;
    private double rendimento;

    public void aplicarRendimentos() {
        saldo += saldo * rendimento;
    }
}
```

Excelente conteúdo, Patrick! Esse trecho explica com bastante clareza os dois pilares importantes da Programação Orientada a Objetos em Java: **herança** e **polimorfismo**. Vou complementar e deixar o material mais didático e completo pra você. Aqui vai uma versão revisada com alguns reforços, exemplos práticos e explicações mais diretas:

3.1. Herança

3.1.1. Conceito de Herança

A **herança** permite que uma classe reutilize atributos e comportamentos (métodos) de outra. É uma das formas mais eficazes de **reutilização de código**, facilitando a **manutenção** e a **organização** dos sistemas.

- A classe que fornece os atributos e métodos é chamada de **superclasse**.
- A classe que herda é chamada de **subclasse**.

No sistema bancário, temos um ótimo exemplo disso:

- Classe **Cliente** é a superclasse.
- Classes **PessoaFisica** e **PessoaJuridica** são subclasses.

Exemplo:

```
public class Cliente {
    protected int codigo;
    protected String cidade;
    protected String estado;

    public String listarDados() {
        return "CÓDIGO: " + codigo + "\nCIDADE: " + cidade + "\nESTADO: " + estado;
    }
}

public class PessoaFisica extends Cliente {
    private String nome;
    private String cpf;

    @Override
    public String listarDados() {
        return "NOME: " + nome + "\nCPF: " + cpf + "\n" + super.listarDados();
    }
}
```

➡ A palavra-chave `extends` indica herança.

➡ `super` é usada para acessar membros da superclasse.

3.1.2. Superclasses e Subclasses

Como vimos, a **superclasse** representa algo mais genérico. Já a **subclasse** representa uma versão mais específica, que pode:

- Herdar métodos e atributos da superclasse.
- Adicionar seus próprios métodos e atributos.
- **Sobrescrever** métodos da superclasse para alterar o comportamento.

3.2. Polimorfismo

3.2.1. Conceito de Polimorfismo

A palavra **polimorfismo** vem do grego *"poli"* (muitas) e *"morphos"* (formas). No contexto da programação orientada a objetos, significa que um **mesmo método pode se comportar de diferentes formas**, dependendo da classe que o implementa.

Exemplo 1: Sobrescrita de métodos (Override)

```
public class Cliente {
    public String listarDados() {
        return "Código: " + codigo;
    }
}

public class PessoaFisica extends Cliente {
    private String nome;
    private String cpf;

    @Override
    public String listarDados() {
        return "Nome: " + nome + "\nCPF: " + cpf + "\n" + super.listarDados();
    }
}
```

➡ O método `listarDados()` foi sobrescrito em `PessoaFisica` para adicionar nome e CPF, mas ainda reutiliza parte da lógica da superclasse com `super.listarDados()`.

Exemplo 2: Polimorfismo em tempo de execução

```
Cliente cliente = new PessoaFisica(); // referência do tipo Cliente, mas objeto é PessoaFisica
System.out.println(cliente.listarDados()); // chama o método da classe PessoaFisica
```

➡ Mesmo usando a referência do tipo `Cliente`, o Java chama o método correspondente ao **tipo real do objeto** (PessoaFisica). Isso é **polimorfismo em tempo de execução**.

3.2.2. Sobrecarga de métodos (Overload)

A **sobrecarga** permite criar **várias versões de um mesmo método**, diferenciadas pela **assinatura** (quantidade e/ou tipo dos parâmetros).

Exemplo:

```
public class Conta {
    public void depositar(double valor) {
        // depósito em dinheiro
    }

    public void depositar(double valor, int numeroCheque) {
        // depósito em cheque
    }
}
```

➡ A decisão de qual método será executado é feita **em tempo de compilação**, com base na assinatura.

Resumo: Diferença entre Override e Overload

Conceito	Override (Sobrescrita)	Overload (Sobrecarga)
Onde ocorre	Em subclasses	Na mesma classe
Quando acontece	Em tempo de execução (runtime)	Em tempo de compilação (compile time)
O que muda	Implementação do método	Parâmetros (tipo, número ou ordem)
Palavra-chave	`@Override` (opcional, mas recomendada)	Nenhuma

Se quiser, posso complementar com exemplos práticos de `ContaCorrente` e `ContaPoupanca` usando herança e polimorfismo. Deseja incluir isso no seu material?