

LPOO

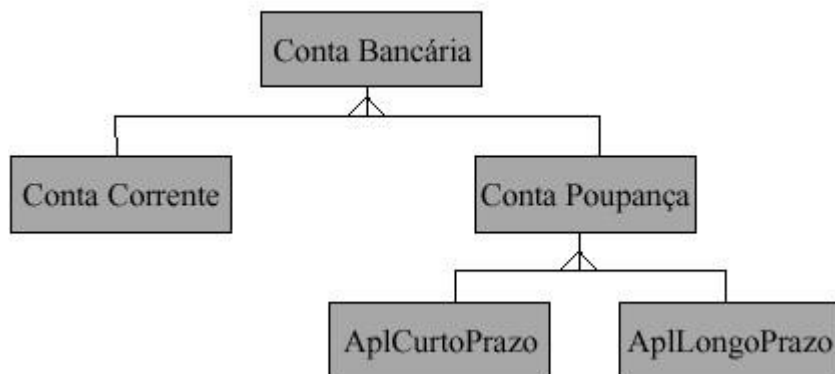
• Herança em Java

É um princípio da Programação Orientada a Objetos que permite que as classes compartilhem atributos e métodos comuns baseados em um relacionamento. A herança possibilita a aplicação de vários conceitos de orientação a objetos que não seriam viáveis sem a organização e estruturação alcançada por sua utilização. Herança é um mecanismo para derivar novas classes a partir de classes existentes através de um processo de refinamento. Uma classe derivada herda a estrutura de dados e métodos de sua classe “base”, mas pode seletivamente:

- *Adicionar novos métodos*
- *Estender a estrutura de dados*
- *Redefinir a programação de métodos já existentes*

Uma classe “base” proporciona a funcionalidade que é comum a todas as suas classes derivadas, enquanto que uma classe derivada proporciona a funcionalidade adicional que especializa seu comportamento.

EXEMPLO DE HERANÇA



Temos uma classe Conta Bancária com a estrutura de funcionamento básico de uma conta, as demais contas herdam estas características e programam apenas as suas diferenças.

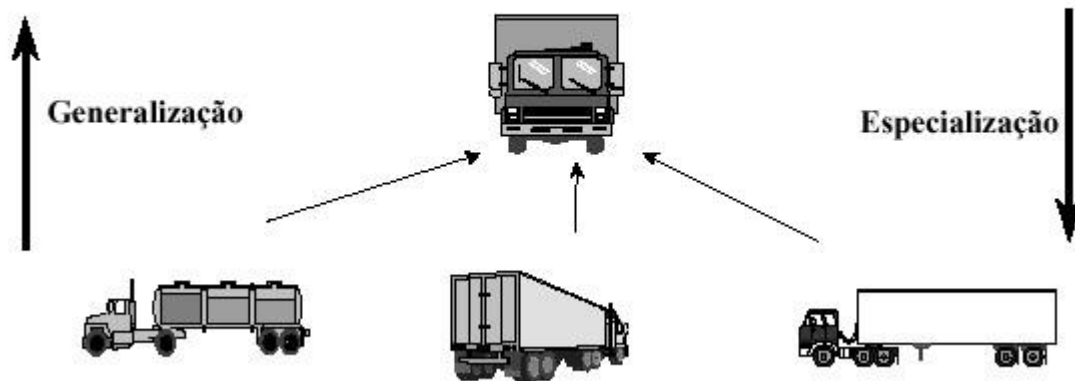
GENERALIZAÇÃO /ESPECIALIZAÇÃO

É a primeira abordagem a ser feita sobre herança e possibilita a vantagem mais direta e evidente que é a reutilização de código.

A generalização é o agrupamento de características (atributos) e regras (métodos) comuns em um modelo de sistema. Já a especialização é o processo inverso, é a definição das particularidades de

cada elemento de um modelo de sistemas, detalhando características e regras que são específicas de um o objeto.

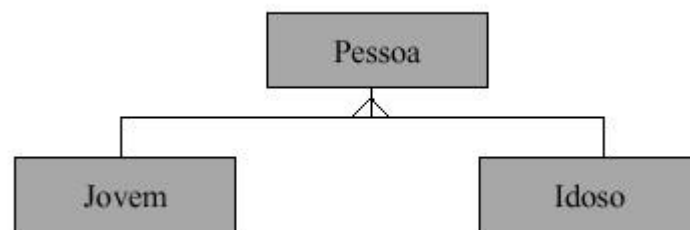
Generalização é a abstração que permite **compartilhar** semelhanças, preservando diferenças



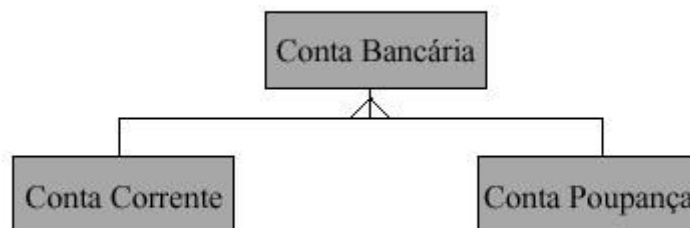
Tipos de Herança

- **Herança de Comportamento**

Uma classe S é um subtipo de T se e somente se S proporciona pelo menos o comportamento de T.

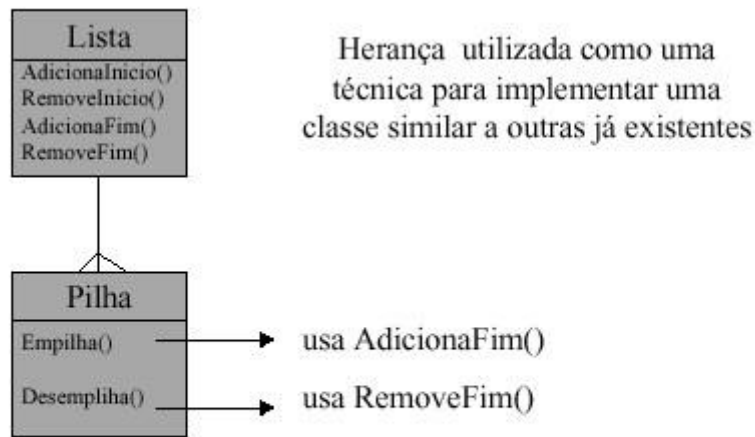


Outro exemplo:



- **Herança de Implementação**

Neste tipo de herança acrescentam-se elementos a uma nova classe, que a distingue da original apenas por executarem ações diferentes, embora na essência possuam os mesmos métodos.

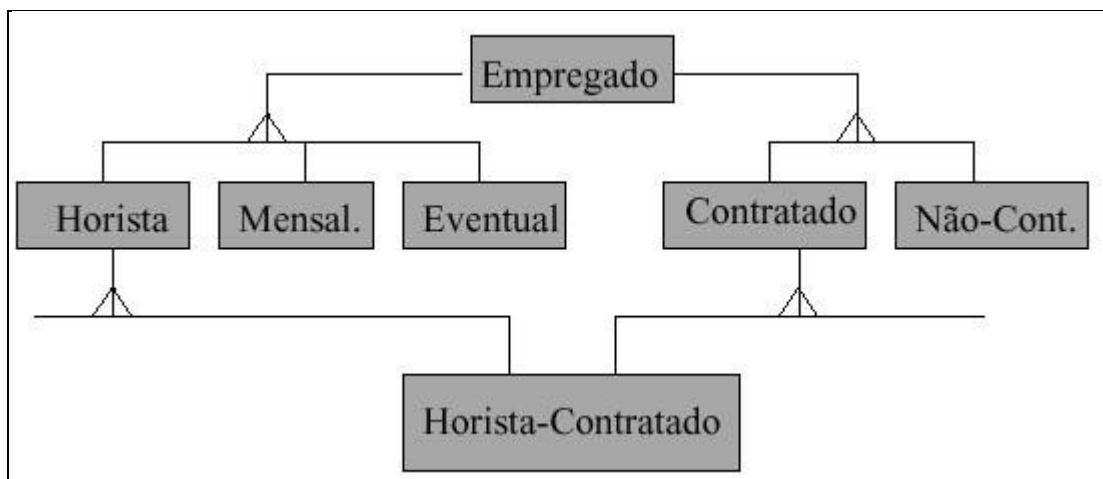


Herança por inclusão (estruturas iguais - comportamentos diferentes)

• Herança Múltipla

Permite que uma classe tenha mais que uma superclasse associada e que herde características de todas elas. Neste caso a classe filha, pode ter mais de um pai, herdando métodos e propriedades que forem necessárias a programação da funcionalidade da classe.

As estruturas herdadas multiplamente podem possuir grandes benefícios na reutilização do código, mas tornam a estrutura bastante complexa, logo seu uso deve ser feito com extremo cuidado, não se recomenda utilizá-las a menos que sejam extremamente necessárias.



Nota: Devida a sua complexidade e a grande exigência de memória necessária para executar uma herança múltipla ela não é permitida na linguagem Java.

• Vantagens do uso de herança

1. Possibilidade de combinação de informações de diversas fontes.
2. Maior capacidade na especificação de classes.
3. Aumento da possibilidade de reuso.

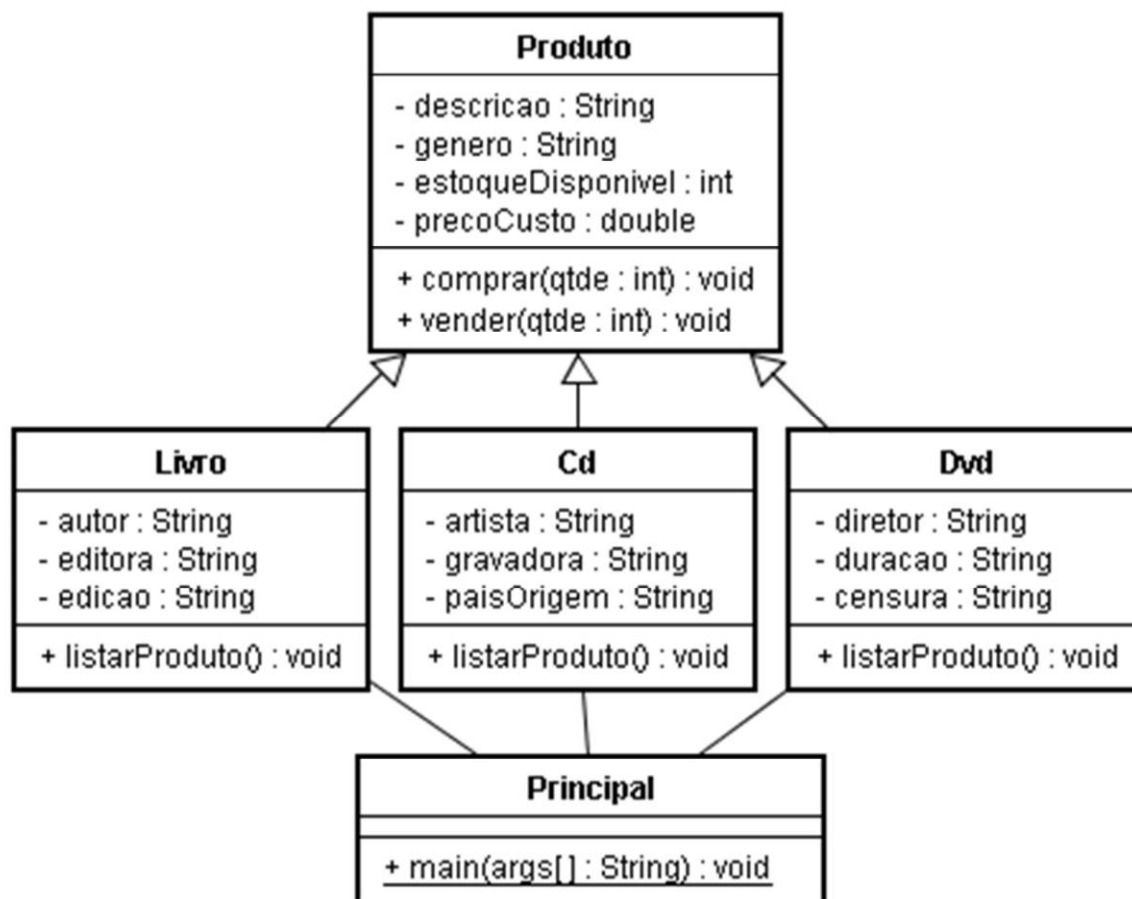
• Herança Simples

Permite que uma classe tenha apenas uma superclasse associada e que herde características dela. Neste caso a classe filha não pode ter mais de um pai, só herdando métodos e propriedades que forem necessárias a programação da funcionalidade da classe de apenas uma única superclasse. A vantagem desta herança é a simplicidade conceitual e de programação, embora tenha menos Reusabilidade de código. Esta herança é utilizada em Java.

Programando herança em Java

Java permite que objetos acessem os métodos e dados de outra classe, os combinando na formação de uma nova classe. Essa capacidade de reutilizar classes existentes acaba economizando tempo de programação. A herança é feita em Java usando-se a palavra reservada **extends**.

Para entender a herança vamos trabalhar a partir do seguinte projeto:



O diagrama permite reconhecer que a superclasse é a classe **Produto**, ela servirá de base para construir as classes **Livro**, **Cd** e **DVD**, que na verdade não deixam de serem produtos, apenas possuem propriedades a mais.

Crie um novo projeto:

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name:

Contents


☒ Create new project in workspace
☐ Create project from existing source

Directory:

Inicialmente vamos criar a nossa superclasse chamada Produto:

New Java Class

Java Class

 The use of the default package is discouraged.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☐ public ☒ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)
☐ Constructors from superclass
☐ Inherited abstract methods

Esta classe deve ser preenchida com as propriedades descritas no diagrama UML. Além dos getters e setters específicos.

O resultado inicial da classe será este:

```

class Produto {
private String descricao;
private String genero;
private int estoqueDisponivel;
private double precoCusto;

public String getDescricao() {
    return descricao;
}

public void setDescricao(String descricao) {
    this.descricao = descricao;
}

public String getGenero() {
    return genero;
}

public void setGenero(String genero) {
    this.genero = genero;
}

public int getEstoqueDisponivel() {
    return estoqueDisponivel;
}

public void setEstoqueDisponivel(int estoqueDisponivel) {
    this.estoqueDisponivel = estoqueDisponivel;
}

public double getPrecoCusto() {
    return precoCusto;
}

public void setPrecoCusto(double precoCusto) {
    this.precoCusto = precoCusto;
}

}

```

Acrescenta agora os métodos construtores:

```

Produto() {
    this("", "", 0, 0);
}

public Produto(String descricao, String genero, int estoqueDisponivel,
    double precoCusto) {
    this.descricao = descricao;
    this.genero = genero;
    this.estoqueDisponivel = estoqueDisponivel;
    this.precoCusto = precoCusto;
}

```

Programar o método comprar:

```

void comprar(int qtd){
    JOptionPane.showMessageDialog(null, "Estoque antigo =" + this.getEstoqueDisponivel());
    this.setEstoqueDisponivel(this.getEstoqueDisponivel() + qtd);
    JOptionPane.showMessageDialog(null, "Estoque novo =" + this.getEstoqueDisponivel());
}

```

Programar o método vender:

```

void vender(int qtd){
    JOptionPane.showMessageDialog(null, "Estoque antigo =" + this.getEstoqueDisponivel());
    this.setEstoqueDisponivel(this.getEstoqueDisponivel() - qtd);
    JOptionPane.showMessageDialog(null, "Estoque novo =" + this.getEstoqueDisponivel());
}

```

Pronto. Nossa classe completa ficará assim:

```

import javax.swing.JOptionPane;

class Produto {
    private String descricao;
    private String genero;
    private int estoqueDisponivel;
    private double precoCusto;

    Produto() {
        this("", "", 0, 0);
    }

    public Produto(String descricao, String genero, int estoqueDisponivel,
        double precoCusto) {
        this.descricao = descricao;
        this.genero = genero;
        this.estoqueDisponivel = estoqueDisponivel;
        this.precoCusto = precoCusto;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public String getGenero() {
        return genero;
    }

    public void setGenero(String genero) {
        this.genero = genero;
    }

    public int getEstoqueDisponivel() {
        return estoqueDisponivel;
    }

    public void setEstoqueDisponivel(int estoqueDisponivel) {
        this.estoqueDisponivel = estoqueDisponivel;
    }

    public double getPrecoCusto() {
        return precoCusto;
    }

    public void setPrecoCusto(double precoCusto) {
        this.precoCusto = precoCusto;
    }
}

```



```

void comprar(int qtd){
    JOptionPane.showMessageDialog(null, "Estoque antigo =" + this.getEstoqueDisponivel());
    this.setEstoqueDisponivel(this.getEstoqueDisponivel() + qtd);
    JOptionPane.showMessageDialog(null, "Estoque novo =" + this.getEstoqueDisponivel());
}

void vender(int qtd){
    JOptionPane.showMessageDialog(null, "Estoque antigo =" + this.getEstoqueDisponivel());
    this.setEstoqueDisponivel(this.getEstoqueDisponivel() - qtd);
    JOptionPane.showMessageDialog(null, "Estoque novo =" + this.getEstoqueDisponivel());
}
}

```

Vamos programar a classe Livro a partir da classe Produto:

New Java Class

The use of the default package is discouraged.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Você obterá o seguinte código básico:

```

public class Livro extends Produto {
}

```

Codificar agora as propriedades e seus respectivos getters e setters:


```

public class Livro extends Produto {
    private String autor;
    private String editora;
    private String edicao;

    public String getAutor() {
        return autor;
    }
    public void setAutor(String autor) {
        this.autor = autor;
    }
    public String getEditora() {
        return editora;
    }
    public void setEditora(String editora) {
        this.editora = editora;
    }
    public String getEdicao() {
        return edicao;
    }
    public void setEdicao(String edicao) {
        this.edicao = edicao;
    }
}

```

Codificar os construtores, aqui você combina a montagem manual a montagem automatizada do Eclipse.

```

- Livro() {
    super("", "", 0, 0);
    this.autor="";
    this.editora="";
    this.edicao="";
    // TODO Auto-generated constructor stub
}

- public Livro(String autor, String editora, String edicao,
    String descricao, String genero, int estoqueDisponivel,
    double precoCusto) {
    super(descricao, genero, estoqueDisponivel, precoCusto);
    this.autor=autor;
    this.editora=editora;
    this.edicao=edicao;
}

```

Programa o método listar Produto:

```
public void listarProduto() {
    JOptionPane.showMessageDialog(null, "Descricao:"+this.getDescricao()+
        "\nGenero:"+this.getGenero()+
        "\nEstoque:"+this.getEstoqueDisponivel()+
        "\nPreço:"+this.getPrecoCusto()+
        "\nAutor:"+this.getAutor()+
        "\nEditora:"+this.getEditora()+
        "\nEdição:"+this.getEdicao());
}
```

Nossa completa agora fica assim:

```
import javax.swing.JOptionPane;

public class Livro extends Produto {
    private String autor;
    private String editora;
    private String edicao;

    Livro() {
        super("", "", 0, 0);
        this.autor = "";
        this.editora = "";
        this.edicao = "";
    }

    public Livro(String autor, String editora, String edicao,
        String descricao, String genero, int estoqueDisponivel,
        double precoCusto) {
        super(descricao, genero, estoqueDisponivel, precoCusto);
        this.autor = autor;
        this.editora = editora;
        this.edicao = edicao;
    }

    public String getAutor() {
        return autor;
    }

    public void setAutor(String autor) {
        this.autor = autor;
    }

    public String getEditora() {
        return editora;
    }

    public void setEditora(String editora) {
        this.editora = editora;
    }

    public String getEdicao() {
        return edicao;
    }

    public void setEdicao(String edicao) {
        this.edicao = edicao;
    }
}
```

```

public void listarProduto() {
    JOptionPane.showMessageDialog(null, "Descricao:"+this.getDescricao()+
        "\nGenero:"+this.getGenero()+
        "\nEstoque:"+this.getEstoqueDisponivel()+
        "\nPreço:"+this.getPrecoCusto()+
        "\nAutor:"+this.getAutor()+
        "\nEditora:"+this.getEditora()+
        "\nEdição:"+this.getEdicao());
}
}

```

Construir a classe CD a partir da classe Produto:

Source folder:

Package: (default)

☐ Enclosing type:

Name:

Modifiers: ☐ public ☒ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Resulta na classe básica:

```

class Cd extends Produto {
}

```

Codificar agora as propriedades e seus respectivos getters e setters:

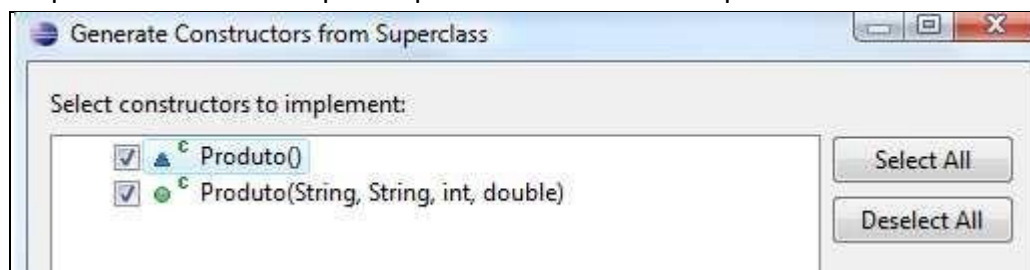
```

class Cd extends Produto {

    private String artista;
    private String gravadora;
    private String paisOrigem;
    public String getArtista() {
        return artista;
    }
    public void setArtista(String artista) {
        this.artista = artista;
    }
    public String getGravadora() {
        return gravadora;
    }
    public void setGravadora(String gravadora) {
        this.gravadora = gravadora;
    }
    public String getPaisOrigem() {
        return paisOrigem;
    }
    public void setPaisOrigem(String paisOrigem) {
        this.paisOrigem = paisOrigem;
    }
}

```

Codificar os construtores, aqui você combina a montagem manual a montagem automatizado do Eclipse. Gerando a base pela superclasse fica mais fácil produzir o resto do código:




```

public Cd() {
    super("", "", 0, 0);
    this.artista="";
    this.gravadora="";
    this.paisOrigem="";

    // TODO Auto-generated constructor stub
}

public Cd(String artista,String gravadora,String paisOrigem,
        String descricao, String genero, int estoqueDisponivel,
        double precoCusto) {
    super(descricao, genero, estoqueDisponivel, precoCusto);

    this.artista=artista;
    this.gravadora=gravadora;
    this.paisOrigem=paisOrigem;
}

```

Programar o método listar Produto:

```

public void listarProduto() {
    JOptionPane.showMessageDialog(null, "Descricao:"+this.getDescricao()+
        "\nGenero:"+this.getGenero()+
        "\nEstoque:"+this.getEstoqueDisponivel()+
        "\nPreço:"+this.getPrecoCusto()+
        "\nArtista:"+this.getArtista()+
        "\nGravadora:"+this.getGravadora()+
        "\nPais:"+this.getPaisOrigem());
}

```

A classe completa ficará assim:

```

import javax.swing.JOptionPane;

class Cd extends Produto {

    private String artista;
    private String gravadora;
    private String paisOrigem;

    public Cd() {
        super("", "", 0, 0);
        this.artista="";
        this.gravadora="";
        this.paisOrigem="";

        // TODO Auto-generated constructor stub
    }

    public Cd(String artista,String gravadora,String paisOrigem,
        String descricao, String genero, int estoqueDisponivel,
        double precoCusto) {
        super(descricao, genero, estoqueDisponivel, precoCusto);

        this.artista=artista;
        this.gravadora=gravadora;
        this.paisOrigem=paisOrigem;
    }
}

```

```

public String getArtista() {
    return artista;
}
public void setArtista(String artista) {
    this.artista = artista;
}
public String getGravadora() {
    return gravadora;
}
public void setGravadora(String gravadora) {
    this.gravadora = gravadora;
}
public String getPaisOrigem() {
    return paisOrigem;
}
public void setPaisOrigem(String paisOrigem) {
    this.paisOrigem = paisOrigem;
}

public void listarProduto() {
    JOptionPane.showMessageDialog(null, "Descricao:"+this.getDescricao()+
        "\nGenero:"+this.getGenero()+
        "\nEstoque:"+this.getEstoqueDisponivel()+
        "\nPreço:"+this.getPrecoCusto()+
        "\nArtista:"+this.getArtista()+
        "\nGravadora:"+this.getGravadora()+
        "\nPais:"+this.getPaisOrigem());
}
}

```

Construir a classe Dvd:

Name:	<input type="text" value="Dvd"/>			
Modifiers:	<input checked="" type="radio"/> public	<input type="radio"/> default	<input type="radio"/> private	<input type="radio"/> protected
	<input type="checkbox"/> abstract	<input type="checkbox"/> final	<input type="checkbox"/> static	
Superclass:	<input type="text" value="Produto"/>			<input type="button" value="Browse..."/>
Interfaces:	<div></div>			<input type="button" value="Add..."/>
				<input type="button" value="Remove"/>

O resultado visual básico será:

```

class Dvd extends Produto {
}

```

Codificar agora as propriedades e seus respectivos getters e setters:

```

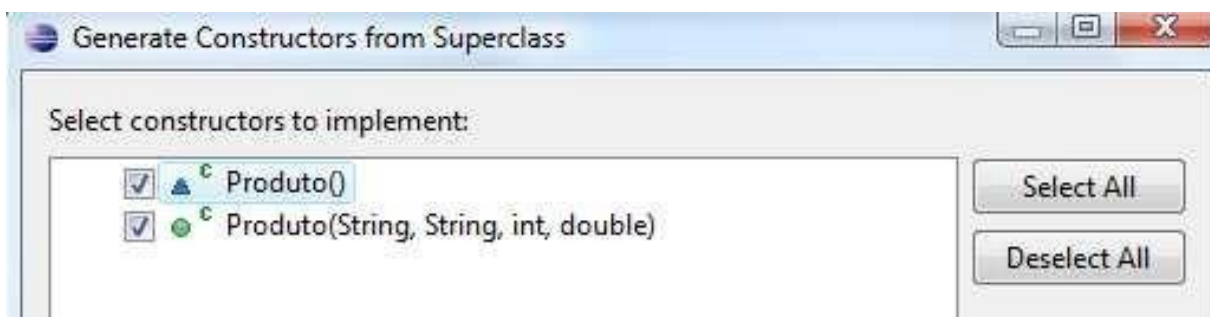
class Dvd extends Produto {

    private String diretor;
    private String duracao;
    private String censura;

    public String getDiretor() {
        return diretor;
    }
    public void setDiretor(String diretor) {
        this.diretor = diretor;
    }
    public String getDuracao() {
        return duracao;
    }
    public void setDuracao(String duracao) {
        this.duracao = duracao;
    }
    public String getCensura() {
        return censura;
    }
    public void setCensura(String censura) {
        this.censura = censura;
    }
}

```

Codificar os construtores, aqui você combina a montagem manual a montagem automatizado do Eclipse. Gerando a base pela superclasse fica mais fácil produzir o resto do código:



O resultado será:


```

    Dvd() {
        super("", "", 0, 0);

        this.diretor="";
        this.duracao="";
        this.censura="";
    }

    public Dvd(String diretor,String duracao,String censura,
        String descricao, String genero, int estoqueDisponivel,
        double precoCusto) {

        super(descricao, genero, estoqueDisponivel, precoCusto);

        this.diretor=diretor;
        this.duracao=duracao;
        this.censura=censura;
    }

```

Programar o método listar Produto:

```

public void listarProduto() {
    JOptionPane.showMessageDialog(null, "Descricao:"+this.getDescricao()+
        "\nGenero:"+this.getGenero()+
        "\nEstoque:"+this.getEstoqueDisponivel()+
        "\nPreço:"+this.getPrecoCusto()+
        "\nDiretor:"+this.getDiretor()+
        "\nDuração:"+this.getDuracao()+
        "\nCensura:"+this.getCensura());
}

```

A classe completa:

```

import javax.swing.JOptionPane;

class Dvd extends Produto {

    private String diretor;
    private String duracao;
    private String censura;

    Dvd() {
        super("", "", 0, 0);

        this.diretor="";
        this.duracao="";
        this.censura="";
    }

    public Dvd(String diretor,String duracao,String censura,
        String descricao, String genero, int estoqueDisponivel,
        double precoCusto) {

        super(descricao, genero, estoqueDisponivel, precoCusto);

        this.diretor=diretor;
        this.duracao=duracao;
        this.censura=censura;
    }
}

```

```

    public String getDiretor() {
        return diretor;
    }

    public void setDiretor(String diretor) {
        this.diretor = diretor;
    }

    public String getDuracao() {
        return duracao;
    }

    public void setDuracao(String duracao) {
        this.duracao = duracao;
    }

    public String getCensura() {
        return censura;
    }

    public void setCensura(String censura) {
        this.censura = censura;
    }

    public void listarProduto() {
        JOptionPane.showMessageDialog(null, "Descricao:"+this.getDescricao()+
            "\nGenero:"+this.getGenero()+
            "\nEstoque:"+this.getEstoqueDisponivel()+
            "\nPreço:"+this.getPrecoCusto()+
            "\nDiretor:"+this.getDiretor()+
            "\nDuração:"+this.getDuracao()+
            "\nCensura:"+this.getCensura());
    }
}

```

Codificando a classe Principal.

```

public class Principal {

    public static void main(String[] args) {

    }

}

```

Neste exemplo simplificado iremos criar tais objetos e checar a funcionalidade das classes consultando um dos objetos.

```

import javax.swing.JOptionPane;

public class Principal {

    public static void main(String[] args) {

        Livro livro = new Livro("José de Alencar", "Saraiva", "Sexta Edição", "A mão e a Luva", "Romance", 10, 48.60);
        Cd cd = new Cd("Chico Buarque", "Record", "Brasil", "Chico e Amigos", "MPB", 30, 23.90);
        Dvd dvd = new Dvd("Noah Share", "45 minutos", "16 banos", "House MD S01", "Seriado", 8, 59.90);

        int op;

        do {

            op = Integer.parseInt(JOptionPane.showInputDialog(null, "1 - Livro\n 2- CD\n 3 - DVD" +
                "\n0- Sair"+
                "\nDigite opção:"));

            if( op ==0) break ;

            switch(op){

                case 1: livro.listarProduto();break;
                case 2: cd.listarProduto();break;
                case 3: dvd.listarProduto();break;
                default: JOptionPane.showMessageDialog(null, "Opção inválida");break;

            }

        } while(true);

    }

}

```

- **Considerações:**

A herança é definida na implementação da subclasse, ou seja, quanto a codificação, a superclasse não apresenta nenhuma diferença. O relacionamento de herança é gerado através do comando `extends` que é programado na assinatura (cabeçalho) da subclasse indicando a sua superclasse.

Podemos resumir que a finalidade dos construtores é inicializar os atributos de um objeto na sua instanciação. Considerando que uma subclasse herda os atributos (além dos métodos) de sua superclasse, devemos, portanto, preparar os construtores da subclasse para inicializarem também os atributos herdados. Para isso utilizamos o método **`super ()`**.

- **Sobrecarga (Overload):**

A sobrecarga de método é a possibilidade de programar dois ou mais métodos com o mesmo nome e diferentes passagens de parâmetros. As diferenças nos parâmetros podem ser na quantidade e/ou no tipo. A sobrecarga pode ser programada tanto em métodos da mesma classe quanto em métodos herdados de uma superclasse. Um exemplo bem comum (e já utilizado) é a programação de dois construtores, um com e outro sem passagem de parâmetros.

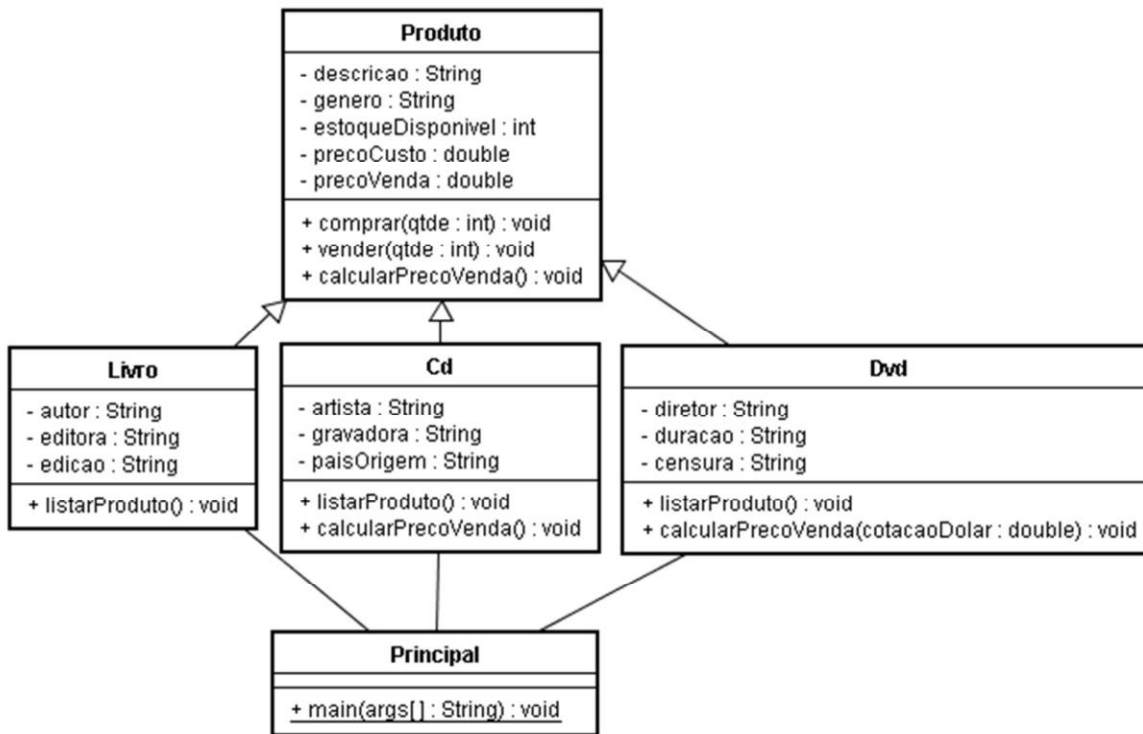
```
public Produto() {  
    this("", "", 0, 0);  
}  
  
public Produto(String descricao, String genero, int estoqueDisponivel, double precoCusto) {  
    this.descricao = descricao;  
    this.genero = genero;  
    this.estoqueDisponivel = estoqueDisponivel;  
    this.precoCusto = precoCusto;  
}
```

- **Sobrescrita (Overwrite):**

A sobrescrita ou reescrita de método está diretamente relacionada com herança e é a possibilidade de manter a mesma assinatura de um método herdado e reescrevê-lo na subclasse. Na chamada de um método sobrescrito o Java considera primeiro a classe a partir da qual o objeto foi instanciado, se a superclasse possuir um método com a mesma assinatura este será descartado.

Tarefa 04

1. Construa as seguintes modificações no projeto: Livraria



Super Classe: Produto

Métodos	Ações a serem realizadas pelo método
Calcular Preço de Venda	Calcula 10% sobre o preço de custo e armazena no atributo precoVenda (precoVenda += precoVenda * 0.1).

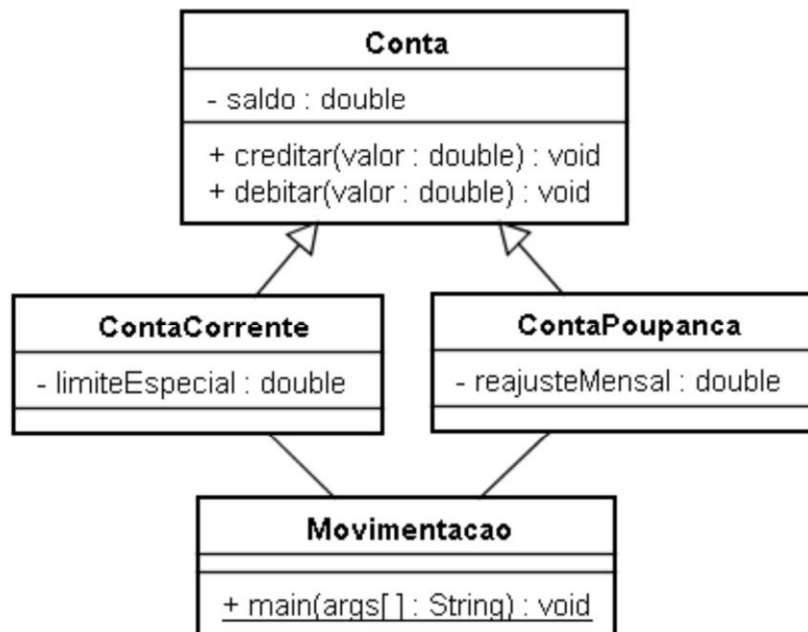
Classe: Cd

Métodos	Ações a serem realizadas pelo método
Calcular Preço de Venda	Calcula 15% sobre o preço de custo e armazena no atributo precoVenda (precoVenda += precoVenda * 0.15).

Classe: Dvd

Métodos	Ações a serem realizadas pelo método
Calcular Preço de Venda	Calcula o preço de venda utilizando a cotação do dólar passada por parâmetro (precoVenda = precoVenda * cotacaoDolar).

2. Construa o seguinte Projeto em Java: Projeto Controle Bancário.



Classe: Conta

Métodos	Ações a serem realizadas pelo método
Creditar	Recebe um valor por parâmetro e soma ao atributo saldo
Debitar	Recebe um valor por parâmetro e subtrai do atributo saldo desde que haja saldo suficiente, caso não tenha, apresenta mensagem de saldo insuficiente.

Classe: Conta Corrente

Atributo	Inicialização
Limite Especial	O atributo será criado, mas não será utilizado por enquanto.

Classe: Conta Poupança

Atributo	Inicialização
Limite Especial	O atributo será criado, mas não será utilizado por enquanto.

Classe: Movimentação

Métodos	Ações a serem realizadas pelo método
Main	Instanciar um objeto do tipo Conta Corrente chamado cc1 com saldo inicial de 500 e limite especial de 1000.

Instanciar um objeto do tipo Conta Poupança chamada cp1 com saldo inicial de 5000 e reajuste Mensal de 1% (0.01). Apresentar uma lista com as opções:

1 – Conta corrente

2 – Poupanca

0 – Sair

Apresentar outra lista com as opções:

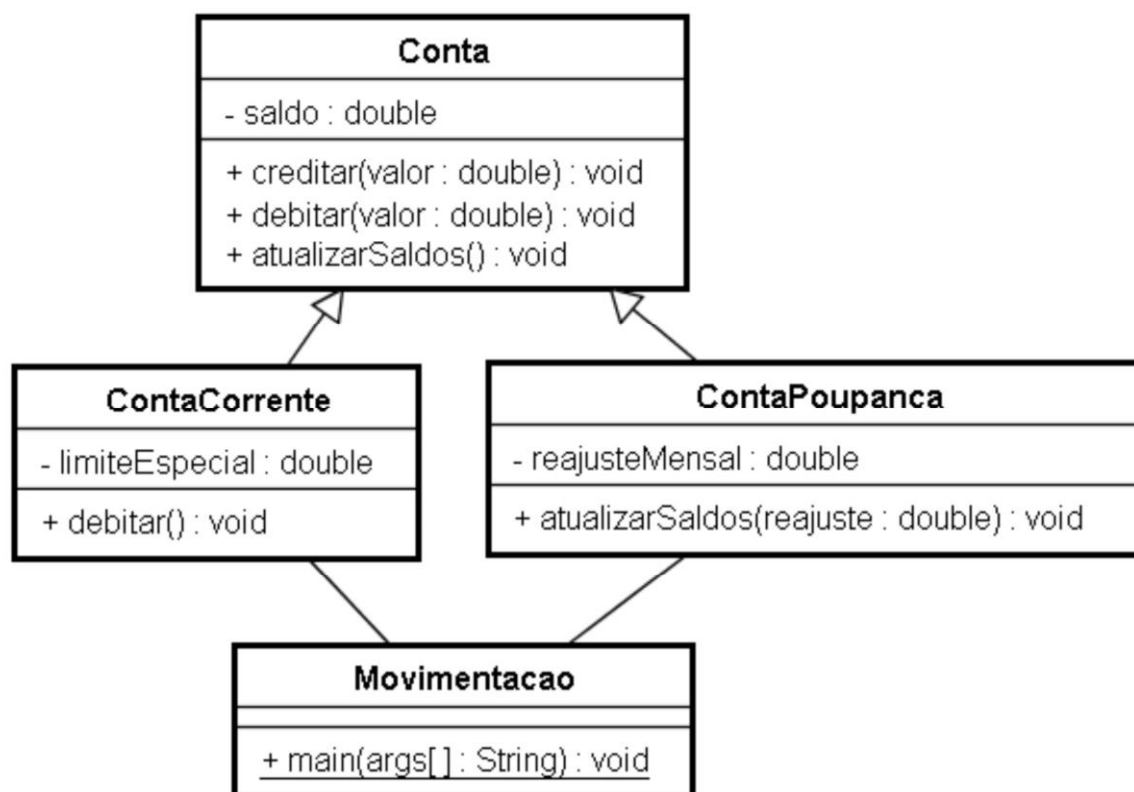
1 – Depositar

2 – Sacar

3 – Consultar saldo 0 – Sair

Realizar as chamadas aos métodos de acordo com as opções do usuário.

1. Construa as modificações no Projeto em Java: Projeto Controle Bancário.



Classe: Conta

Métodos	Ações a serem realizadas pelo método
Creditar	Recebe um valor por parâmetro e soma ao atributo saldo
Debitar	Recebe um valor por parâmetro e subtrai do atributo saldo desde que haja saldo suficiente, caso não tenha, apresenta mensagem de saldo insuficiente.
Atualizar Saldo:	Verifica se o atributo saldo está negativo, caso esteja, calcula 8% (0.08) sobre o valor excedente e subtrai do saldo (Cobra juros pela utilização de limite especial). Apresentar o saldo anterior e o saldo atualizado.

Classe: Conta Corrente

Métodos	Ações a serem realizadas pelo método
Debitar	Sobrescrever o método debitar considerando o atributo limite Especial. O saldo poderá ficar negativo até o valor indicado em limite Especial.

Classe: Conta Poupança

Métodos	Ações a serem realizadas pelo método
Atualizar Saldo:	<p>Sobrecarregar o método atualizar Saldo de modo que ele receba por parâmetro uma porcentagem para reajuste (um valor decimal double).</p> <p>Calcular a porcentagem informada sobre o saldo e somar ao saldo (Rendimento da poupança).</p> <p>Armazenar a porcentagem informada no atributo reajuste Mensal.</p> <p>Apresentar o saldo anterior e o saldo atualizado.</p>

Classe: Movimentação

Métodos	Ações a serem realizadas pelo método
Main	<ul style="list-style-type: none">• Instanciar um objeto do tipo Conta Corrente chamado cc1 com saldo inicial de 500 e limite especial de 1000.

Instanciar um objeto do tipo Conta Poupança chamada cp1 com saldo inicial de 5000 e reajuste Mensal de 1% (0.01) Apresentar uma lista com as opções:

1 – Conta corrente

2 – Poupança

0 – Sair

- Apresentar outra lista com as opções:

1 – Depositar

2 – Sacar

3 – Consultar saldo

4 – Reajustar saldo 0 – Sair

Realizar as chamadas aos métodos de acordo com as opções do usuário.

Obs.: No reajuste da poupança informar a porcentagem a ser aplicada (em formato decimal)

• Classes Concretas e classes abstratas

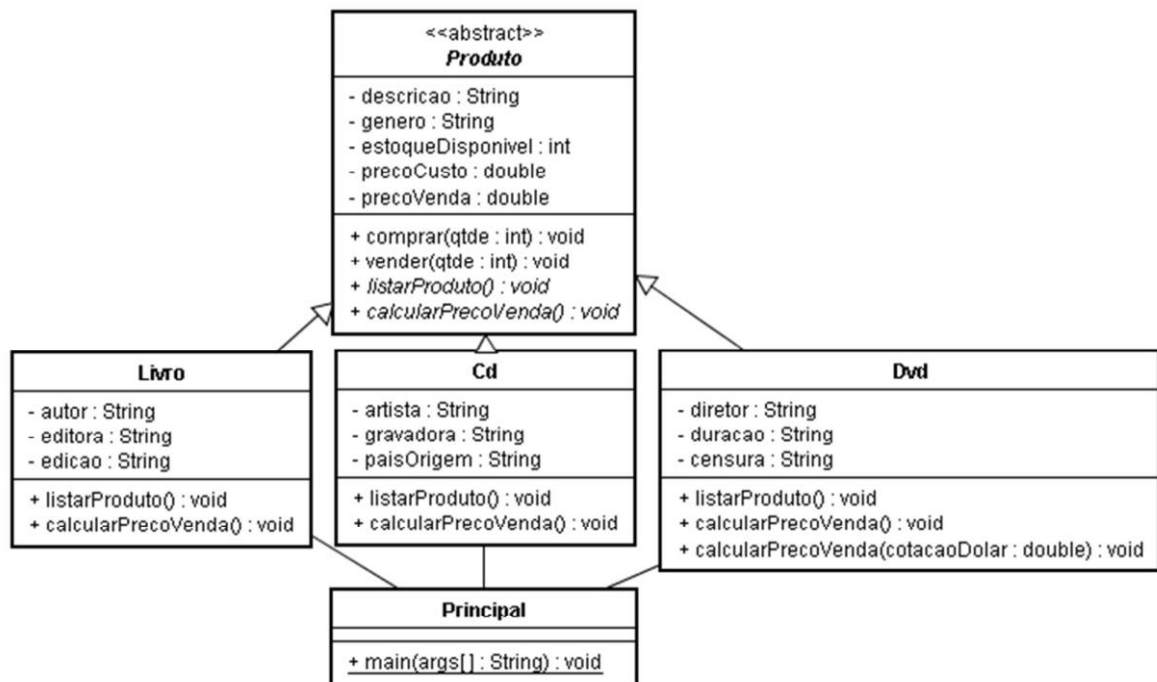
Uma classe abstrata é desenvolvida para representar entidades e conceitos abstratos. A classe abstrata é sempre uma superclasse que não possui instâncias. Ela define um modelo para uma funcionalidade e fornece uma programação incompleta (a parte genérica dessa funcionalidade) que é compartilhada por um grupo de classes derivadas (subclasses). Cada uma das classes derivadas completa a funcionalidade da classe abstrata adicionando um comportamento específico.

• Métodos Abstratos

Uma classe abstrata pode conter métodos concretos, porém, um método abstrato só pode ser definido em uma classe abstrata. Esses métodos são programados nas suas subclasses (concretas) com o objetivo de definir um comportamento (regras) específico. Um método abstrato define apenas a assinatura do método e, portanto, não contém código.

No exemplo da livraria, a classe Produto nunca será utilizada para instanciar um objeto porque os produtos efetivamente comercializados são livros, CDs e DVDs. A finalidade da classe Produto é

somente a generalização dos produtos, portanto, conceitualmente ela deve ser definida como uma classe abstrata. Partindo desse princípio, o método `calcularPrecoVenda()` a ser programado na classe também é um forte candidato a ser um método abstrato, porque, nesse exemplo, cada produto tem sua própria regra de calculo. Observe o diagrama UML:



Vamos modificar nossa classe produto para que ela se torne abstrata.

```

import javax.swing.JOptionPane;

abstract class Produto {
    private String descricao;
    private String genero;
    private int estoqueDisponivel;
    private double precoCusto;

    Produto() {
        this("", "", 0, 0);
    }

    public Produto(String descricao, String genero, int estoqueDisponivel,
        double precoCusto) {
        this.descricao = descricao;
        this.genero = genero;
        this.estoqueDisponivel = estoqueDisponivel;
        this.precoCusto = precoCusto;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public String getGenero() {
        return genero;
    }

    public void setGenero(String genero) {
        this.genero = genero;
    }

    public int getEstoqueDisponivel() {
        return estoqueDisponivel;
    }

    public void setEstoqueDisponivel(int estoqueDisponivel) {
        this.estoqueDisponivel = estoqueDisponivel;
    }

    public double getPrecoCusto() {
        return precoCusto;
    }

    public void setPrecoCusto(double precoCusto) {
        this.precoCusto = precoCusto;
    }
}

```

Acrescentar no final da classe os métodos abstratos:

```

void comprar(int qtd){
    JOptionPane.showMessageDialog(null, "Estoque antigo =" + this.getEstoqueDisponivel());
    this.setEstoqueDisponivel(this.getEstoqueDisponivel() + qtd);
    JOptionPane.showMessageDialog(null, "Estoque novo =" + this.getEstoqueDisponivel());
}

void vender(int qtd){
    JOptionPane.showMessageDialog(null, "Estoque antigo =" + this.getEstoqueDisponivel());
    this.setEstoqueDisponivel(this.getEstoqueDisponivel() - qtd);
    JOptionPane.showMessageDialog(null, "Estoque novo =" + this.getEstoqueDisponivel());
}

abstract void listarProduto();
abstract void calcularPrecoVenda();
}

```

Programar os métodos abstratos nas classes filhas, no nosso caso o calcular preço de venda, já que ao listar Produto é que encontra criado. Vamos começar na classe Livro: (15 % de lucro no preço de custo)

```

public void listarProduto(){
    JOptionPane.showMessageDialog(null, "Descricao:" + this.getDescricao() +
        "\nGenero:" + this.getGenero() +
        "\nEstoque:" + this.getEstoqueDisponivel() +
        "\nPreço:" + this.getPrecoCusto() +
        "\nAutor:" + this.getAutor() +
        "\nEditora:" + this.getEditora() +
        "\nEdição:" + this.getEdicao());
}

public void calcularPrecoVenda(){
    JOptionPane.showMessageDialog(null, "Preço Venda=" + this.getPrecoCusto() + this.getPrecoCusto() * 0.15);
}

```

Na classe Cd vamos ter um lucro variável, dependendo do Cd vendido, neste caso vamos fazer a reescrita do método abstrato.

```

public void listarProduto(){
    JOptionPane.showMessageDialog(null, "Descricao:" + this.getDescricao() +
        "\nGenero:" + this.getGenero() +
        "\nEstoque:" + this.getEstoqueDisponivel() +
        "\nPreço:" + this.getPrecoCusto() +
        "\nArtista:" + this.getArtista() +
        "\nGravadora:" + this.getGravadora() +
        "\nPais:" + this.getPaisOrigem());
}

public void calcularPrecoVenda(){
}

public void calcularPrecoVenda(double pcLucro){
    JOptionPane.showMessageDialog(null, "Preço Venda=" + this.getPrecoCusto() + this.getPrecoCusto() * pcLucro / 100);
}

```

Na classe DVD nós dependemos da cotação do dólar para calcular o preço, logo o método tem que ser reescrito também:

```
public void calcularPrecoVenda() {  
}  
public void calcularPrecoVenda(double cotacaoDolar, double pcLucro) {  
    JOptionPane.showMessageDialog(null, "Preço Venda="+  
        this.getPrecoCusto() + (this.getPrecoCusto() * cotacaoDolar) * pcLucro / 100);  
}
```

Note que obrigatoriamente temos que escrever o método abstrato como ele foi declarado na classe mãe, à sobrescrita de métodos abstratos não é permitida em Java, apenas a sobrecarga. Para que ocorra uma redefinição de método, é necessário que o novo método possua assinatura idêntica à de um método herdado pela classe que será redefinido. O método da subclasse e o método herdado devem coincidir quanto ao tipo de resultado, ao nome e à lista de parâmetros (mesmo número de parâmetros, de mesmos tipos e na mesma ordem).

Não havendo essa coincidência de assinaturas entre um método herdado e um método definido na subclasse, mas apenas uma coincidência entre nomes de métodos, o que ocorre é apenas uma sobrecarga de métodos. É importante ressaltar que, neste último caso, os tipos de resultado dos métodos podem ser diferentes apenas se houverem diferenças também nas listas de parâmetros.

• Classes Polimórficas

O polimorfismo permite que objetos de diferentes subclasses sejam tratados como objetos de uma única superclasse. Pode-se dizer que o Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas. De uma forma mais clara podemos afirmar que o polimorfismo nos dá a possibilidade de manipular um objeto como sendo outro. O polimorfismo não quer dizer que o objeto se transforma em outro. Um objeto sempre será do tipo que foi instanciado o que pode mudar é a maneira como nos referimos a ele.

Pense no nosso projeto Livraria. Os produtos efetivamente comercializados, e que devem ser gerenciados, são livro, cd e DVDs. Contudo, a afirmação: “Livro, cd e DVD são produtos” esta correta.

O conceito de polimorfismo possibilita que tratemos um livro, cd ou DVD como um produto (pois, afinal de contas, eles são produtos), mas sem perdermos as suas características específicas porque apesar de serem produtos eles não deixam de ser um livro, um cd ou um DVD (especializações). Partindo desse raciocínio podemos definir métodos que reconheçam objetos através de suas superclasses, mas que mantenham seus atributos e métodos programados nas subclasses de origem. Para exemplificar isto, vamos modificar nosso método Main na classe Principal para refletir esse conceito.

Nela não vamos criar nenhum momento criar um objeto que seja da classe Cd, Dvd ou Livro, iremos declarar um objeto da classe Produto e usar os construtores da subclasses para construir nosso produto específico. O bom desta técnica polimórfica é que preservamos as semelhanças, desprezando as diferenças de construção das classes, o que é possível graças a abstração feita na classe mãe.

```
import javax.swing.JOptionPane;

public class Principal {

    public static void main(String[] args) {

        Produto produto;

        int op;

        do {

            op = Integer.parseInt(JOptionPane.showInputDialog(null,"1 - Livro\n2- CD\n3 - DVD" +
            "\n0- Sair"+
            "\nDigite opção:"));

            if( op ==0) break ;

            switch(op){

                case 1:{
                    produto = new Livro("José de Alencar","Saraiva","Sexta Edição","A mão e a Luva","Romance",10,48.60);
                    produto.listarProduto();
                }
                break;
                case 2:{
                    produto =new Cd("Chico Buarque","Record","Brasil","Chico e Amigos","MPB",30,23.90);
                    produto.listarProduto();
                }
                break;
                case 3: {
                    produto = new Dvd("Noah Share","45 minutos","16 banos","House MD S01","Seriado",8,59.90);
                    produto.listarProduto();
                }
                break;
                default: JOptionPane.showMessageDialog(null, "Opção inválida");break;

            }

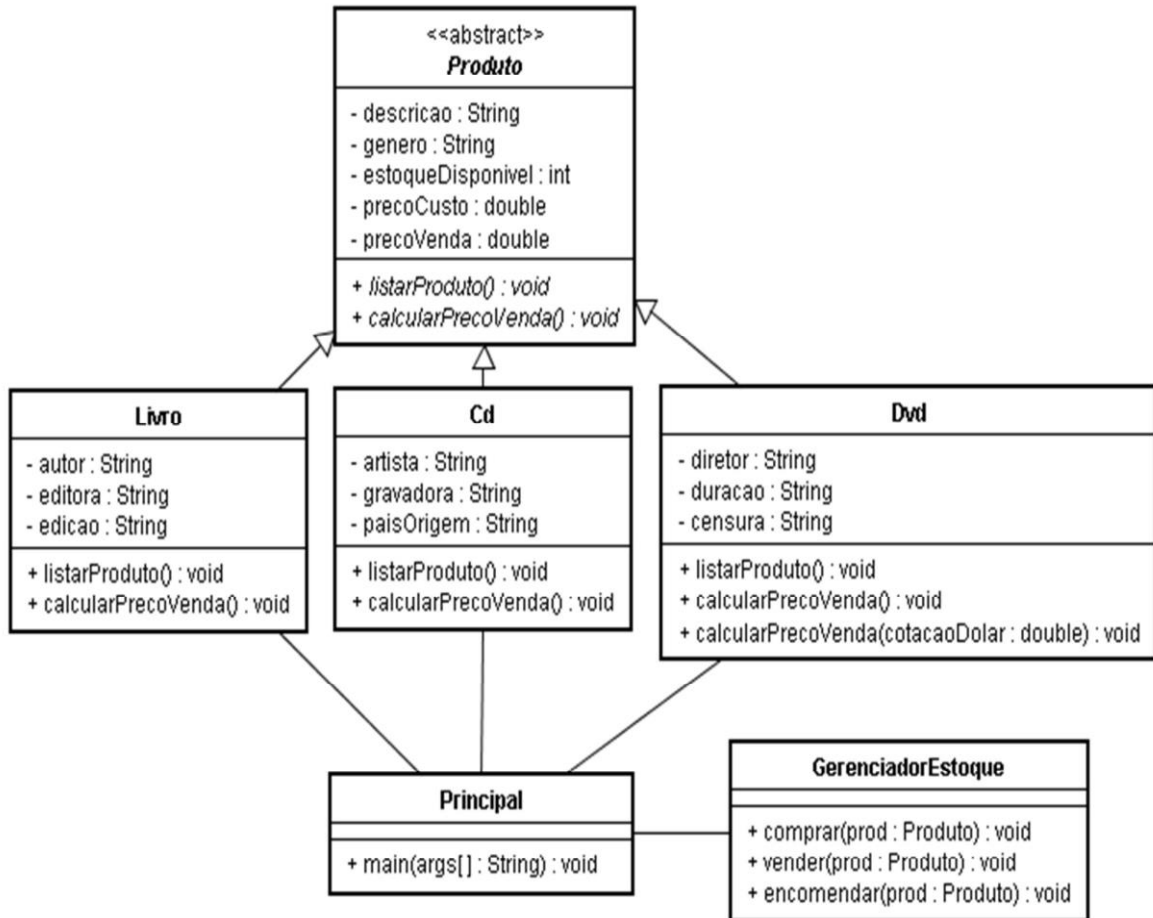
        } while(true);

    }

}
```


EXERCÍCIO

Reescreva o projeto livreria de acordo com o diagrama UML abaixo:



• Matrizes polimórficas

O conceito de polimorfismo pode ser utilizado também na manipulação de arrays de objetos.

Podemos armazenar objetos em arrays de tipos referenciados pela superclasse e armazenar objetos especializados (instanciados a partir de subclasses) tendo acesso sua estrutura de origem.

No nosso caso podemos tratar cada elemento da matriz como sendo de uma classe filha e a matriz como da classe mãe. Codificando ficaria desta forma:

```

import javax.swing.JOptionPane;

public class Principal {

    public static void main(String[] args) {

        Produto produto[] = new Produto[3];

        produto[0] = new Livro("José de Alencar", "Saraiva", "Sexta Edição", "A mão e a Luva", "Romance", 10, 48.60);
        produto[1] = new Cd("Chico Buarque", "Record", "Brasil", "Chico e Amigos", "MPB", 30, 23.90);
        produto[2] = new Dvd("Noah Share", "45 minutos", "16 banos", "House MD S01", "Seriado", 8, 59.90);
        int op;

        do {

            op = Integer.parseInt(JOptionPane.showInputDialog(null, "1 - Livro\n2- CD\n3 - DVD" +
                "\n0- Sair"+
                "\nDigite opção:"));

            if( op ==0) break ;

            switch(op){

                case 1: produto[0].listarProduto();break;
                case 2: produto[1].listarProduto();break;
                case 3: produto[2].listarProduto();break;
                default: JOptionPane.showMessageDialog(null, "Opção inválida");break;

            }

        } while(true);

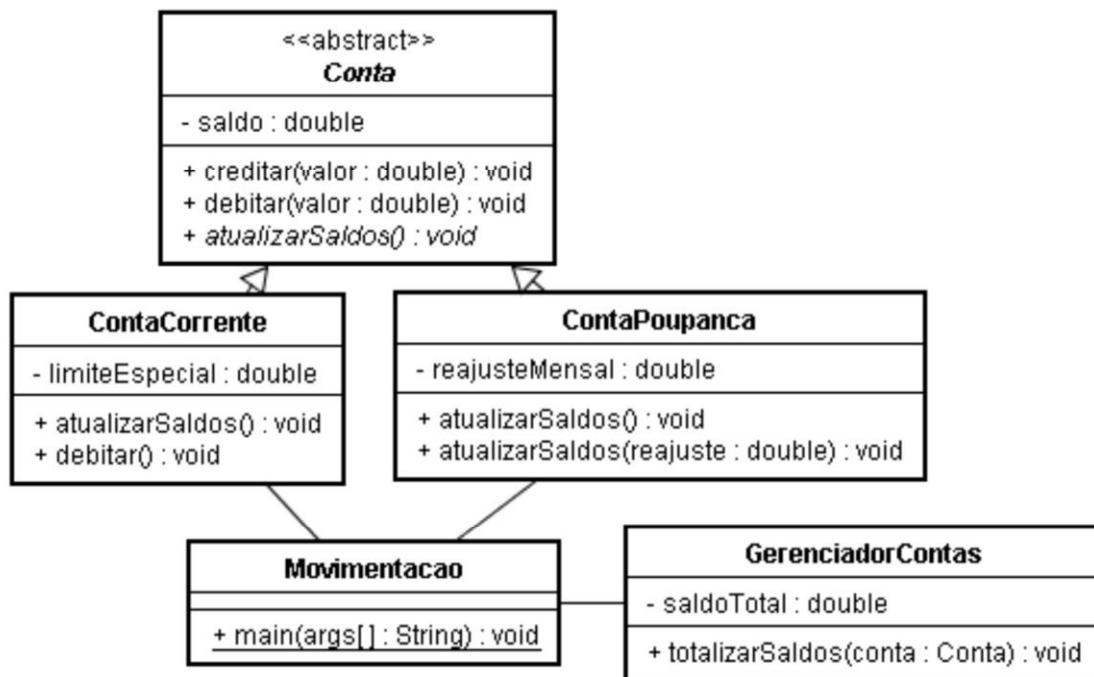
    }

}

```

Tarefa 05

1. Construa o seguinte Projeto em Java: Projeto Controle Bancário.



Classe: Conta

Métodos	Ações a serem realizadas pelo método
Creditar	Recebe um valor por parâmetro e soma ao atributo saldo
Debitar	Recebe um valor por parâmetro e subtrai do atributo saldo desde que haja saldo suficiente, caso não tenha, apresenta mensagem de saldo insuficiente.
Atualizar Saldos (abstrato)	Somente a assinatura do método.

Classe: Conta Corrente

Métodos	Ações a serem realizadas pelo método
Atualizar Saldos	Construção do método (abstrato na superclasse) que verifica se o atributo saldo está negativo, caso esteja, calcula 8% (0.08) sobre o valor excedente e subtrai do saldo (Cobra juros pela utilização de limite especial). Apresentar o saldo anterior e o saldo atualizado.
Debitar	Sobrescrever o método debitar considerando o atributo limite Especial. O saldo poderá ficar negativo até o valor indicado em limite Especial.

Classe: Conta Poupança

Métodos	Ações a serem realizadas pelo método
Atualizar Saldo	Sem parâmetro. Método (abstrato na superclasse) herdado que deve ter, pelo menos, a assinatura inserida na subclasse.
Atualizar Saldo	(Com parâmetro): Sobrecarregar o método atualizar Saldo de modo que ele receba por parâmetro uma porcentagem para reajuste (um valor decimal double). Calcular a porcentagem informada sobre o saldo e somar ao saldo (Rendimento da poupança). Armazenar a porcentagem informada no atributo reajuste Mensal. Apresentar o saldo anterior e o saldo atualizado.

Classe: Gerenciador Contas

Métodos	Ações a serem realizadas pelo método
Totalizar Saldos	Recebe um objeto do tipo conta por parâmetro e soma (acumula) o saldo do objeto recebido ao atributo saldo Total.

Classe: Movimentação

Métodos	Ações a serem realizadas pelo método
Main	<p>Instanciar um objeto do tipo Conta Corrente chamado cc1 com saldo inicial de 500 e limite especial de 1000.</p> <p>Instanciar um objeto do tipo Conta Poupança chamada cp1 com saldo inicial de 5000 e reajuste Mensal de 1% (0.01)</p> <p>Instanciar um objeto do tipo Gerenciador Contas chamado gerenciador com saldo Total inicializado com zero.</p> <p>Apresentar uma lista com as opções:</p> <p>1 – Conta corrente</p>

2 – Poupança

3 – Consultar total dos saldos

0 – Sair

Apresentar outra lista com as opções:

1 – Depositar

2 – Sacar

3 – Consultar saldo

4 – Reajustar saldo

5 – Totalizar saldos 0 – Sair

Realizar as chamadas aos métodos de acordo com as opções do usuário. Obs.:

1 – No reajuste da poupança informar a porcentagem a ser aplicada (em formato decimal)

2 – Na opção “Consultar total dos saldos”, apresentar o conteúdo do atributo saldo Total.

3 – Na opção “Totalizar saldos” chamar o método totalizar Saldos duas vezes, uma passando o objeto cc1 e outra passando o objeto cp1 por parâmetro.