Explicação Detalhada do Projeto POO Cinema

Olá! Este documento detalha a solução desenvolvida para a sua atividade de Programação Orientada a Objetos em Java, focando nos requisitos solicitados e nos conceitos aplicados. O objetivo é te ajudar a compreender o código e se preparar para a arguição com o professor.

1. Estrutura do Projeto

O projeto foi organizado em pacotes para melhor separação de responsabilidades:

- **`src/`**: Diretório raiz do código fonte.
  - **`modelo/`**: Contém as classes que representam as entidades do domínio do problema (Ator, Cliente, Filme, etc.). É o coração do sistema.
  - **`persistencia/`**: Contém a classe `Persistencia`, responsável por centralizar as operações de leitura e escrita nos arquivos de texto (`.txt`). Isso desacopla a lógica de negócio da forma como os dados são armazenados.
  - **`excecoes/`**: Contém as classes de exceção personalizadas (`ValidacaoException`, `PersistenciaException`, `NaoEncontradoException`, `MenuOpcaoInvalidaException`, `ErroInternoException`). Usar exceções específicas torna o tratamento de erros mais claro e robusto.
  - **`ui/`**: Contém a classe `CinemaUI`, responsável pela interface com o usuário (o menu de console) e por orquestrar as operações.
- **`data/`**: Diretório onde os arquivos `.txt` de dados serão armazenados (e.g., `atores.txt`, `clientes.txt`, etc.). Este diretório é criado automaticamente se não existir.

2. Conceitos de POO Aplicados

Diversos conceitos fundamentais de POO foram utilizados:

a) Encapsulamento

- **O que é?** É o princípio de esconder os detalhes internos de implementação de um objeto e expor apenas o necessário através de uma interface pública (métodos).

- **Como foi aplicado?**
  - Todos os atributos das classes de modelo (`Ator` , `Cliente` , `Genero` , etc.) foram declarados como `private` (exceto os da classe `Pessoa` , que são `protected` para permitir acesso direto pelas subclasses, conforme solicitado).
  - O acesso e a modificação desses atributos são feitos exclusivamente através de métodos públicos `getters` (e.g., `getNome()`) e `setters` (e.g., `setNome(String nome)`).
  - Os `setters` e construtores incluem validações (lançando `ValidacaoException`) para garantir a integridade dos dados antes de modificar os atributos.
- **Por que é importante?** Protege os dados de modificações inválidas, facilita a manutenção (mudanças internas não afetam quem usa a classe) e melhora a organização do código.

b) Herança

- **O que é?** Permite que uma classe (subclasse) herde atributos e métodos de outra classe (superclasse), promovendo reutilização de código e criando uma hierarquia "é um(a)".
- **Como foi aplicado?**
  - As classes `Ator` e `Cliente` herdam da classe abstrata `Pessoa` (usando `extends Pessoa`).
  - Elas reutilizam os atributos `cpf` , `nome` , `idade` e os métodos `getters` / `setters` correspondentes definidos em `Pessoa` .
  - Cada subclasse adiciona seus próprios atributos específicos (`registroProfissional` para `Ator` , `rg` e `estudante` para `Cliente`) e implementa o método abstrato `mostrar()` de forma específica.
- **Por que é importante?** Evita duplicação de código, facilita a criação de tipos especializados e organiza as classes de forma lógica.

c) Abstração

- **O que é?** Consiste em focar nos aspectos essenciais de um objeto, ignorando detalhes irrelevantes. Classes abstratas e interfaces são mecanismos para alcançar a abstração.
- **Como foi aplicado?**
  - A classe `Pessoa` foi declarada como `abstract` . Isso significa que não se pode criar um objeto diretamente do tipo `Pessoa` (`new Pessoa(...)` daria erro). Ela serve como um modelo base.

    - O método `mostrar()` foi declarado como `abstract` em `Pessoa`. Isso obriga as subclasses (`Ator`, `Cliente`) a fornecerem sua própria implementação concreta desse método.
- **Por que é importante?** Define um contrato comum para as subclasses, garante que certos comportamentos sejam implementados e modela conceitos gerais que não existem de forma concreta.

d) Polimorfismo (Implícito)

- **O que é?** Significa "muitas formas". Permite que objetos de diferentes classes respondam à mesma mensagem (chamada de método) de maneiras diferentes.
- **Como foi aplicado?**
    - Embora não haja um exemplo explícito de variável do tipo `Pessoa` recebendo `Ator` ou `Cliente` no menu atual, a sobrescrita do método `mostrar()` nas classes `Ator` e `Cliente` é um exemplo. Se tivéssemos uma lista de `Pessoa`, poderíamos chamar `pessoa.mostrar()` e o método correto (de `Ator` ou `Cliente`) seria executado.
    - A sobrescrita do método `toString()` em todas as classes também é polimorfismo. Diferentes objetos têm sua própria forma de serem representados como String.
- **Por que é importante?** Torna o código mais flexível e extensível. Permite tratar objetos de diferentes tipos de forma uniforme.

e) Associação

- **O que é?** Representa um relacionamento entre classes, onde um objeto de uma classe "conhece" ou "usa" um objeto de outra classe.
- **Como foi aplicado?**
    - `Filme` tem um atributo do tipo `Genero` (`private Genero genero;`).
    - `Sessao` tem um atributo do tipo `Filme` (`private Filme filme;`).
    - `Ingresso` tem atributos dos tipos `Sessao` e `Cliente` (`private Sessao sessao; private Cliente cliente;`).
    - `Elenco` tem atributos dos tipos `Ator` e `Filme` (`private Ator ator; private Filme filme;`).
- **Como funciona na persistência?** Ao salvar um objeto que tem associações (e.g., `Filme`), salvamos apenas o identificador do objeto associado (e.g., `idGenero`) no arquivo TXT. Ao carregar (`listar`, `consultar`), lemos o identificador e usamos o método `consultar` da classe associada para buscar e recriar o objeto completo.
- **Por que é importante?** Modela como os objetos do mundo real se relacionam e interagem.

3. Persistência em Arquivos TXT

- **Requisito:** Todos os métodos (`inserir` , `editar` , `listar` , `consultar`) devem ler e escrever em arquivos `.txt` .
- **Abordagem:**
  1. **Classe `Persistencia` :** Centraliza as operações de I/O (Input/Output) com arquivos. Ela possui métodos estáticos como `lerArquivo` , `sobrescreverArquivo` , `adicionarLinha` , `consultarLinhaPorId` , `editarObjetoPorId` , `listarLinhas` .
  2. **Formato dos Arquivos:** Cada linha em um arquivo `.txt` representa um objeto. Os atributos do objeto são separados por um caractere delimitador (escolhemos ponto e vírgula `;`). Ex: `atores.txt` -> `cpf;nome;idade;registroProfissional` .
  3. **Método `toString()` :** Cada classe de modelo sobrescreve `toString()` para retornar a representação do objeto no formato exato que será salvo no arquivo TXT.
  4. **Método `fromString(String linha)` (Privado e Estático):** Cada classe de modelo possui um método auxiliar (geralmente `private static`) que recebe uma linha lida do arquivo e a transforma de volta em um objeto da classe, fazendo o parse dos dados. Ele também busca objetos associados se necessário (e.g., `Filme.fromString` busca o `Genero`).
  5. **Métodos CRUD (`inserir` , `editar` , `listar` , `consultar`):**
     - `inserir()` : Chama `Persistencia.inserirObjeto()` , passando o `toString()` do objeto atual.
     - `editar()` : Chama `Persistencia.editarObjetoPorId()` , passando o ID (CPF ou `id*`), o `toString()` do objeto atualizado e o separador.
     - `listar()` : Chama `Persistencia.listarLinhas()` , itera sobre as linhas e usa `fromString()` para criar a lista de objetos.
     - `consultar(id)` : Chama `Persistencia.consultarLinhaPorId()` , e então usa `fromString()` na linha retornada para criar o objeto.
- **Tratamento de Associações:** Ao salvar (`toString`), apenas o ID/CPF do objeto associado é gravado. Ao carregar (`fromString`), o método `consultar` da classe associada é chamado para obter a instância completa do objeto.

4. Tratamento de Exceções

- **Requisito:** Pelo menos 5 classes com tratamento de exceção diferente de `IOException` .

- **Abordagem:**
    1. **Exceções Personalizadas:** Foram criadas 5 classes de exceção no pacote `excecoes`:
        - `ValidacaoException`: Lançada por construtores e `setters` quando um dado é inválido (e.g., ID negativo, nome vazio).
        - `PersistenciaException`: Lançada pela classe `Persistencia` e pelos métodos CRUD das classes de modelo quando ocorre um erro de leitura/escrita ou um problema relacionado ao armazenamento (e.g., ID duplicado ao inserir).
        - `NaoEncontradoException`: Lançada pelos métodos `consultar` e `editar` quando o objeto com o ID/CPF especificado não existe no arquivo.
        - `MenuOpcaoInvalidaException`: Lançada pela `CinemaUI` quando o usuário digita uma opção de menu inexistente.
        - `ErroInternoException` (RuntimeException): Pode ser usada para erros inesperados que não deveriam ocorrer em condições normais.
    2. **Uso:**
        - Os métodos que podem falhar (construtores, setters, CRUD, leitura de opção) declaram que lançam (`throws`) as exceções apropriadas.
        - Na classe `CinemaUI`, as chamadas a esses métodos estão dentro de blocos `try-catch` para capturar as exceções lançadas, exibir uma mensagem de erro amigável para o usuário e permitir que o programa continue (ou termine graciosamente).
        - Capturamos `InputMismatchException` especificamente para entradas não numéricas no menu.
        - Um `catch (Exception e)` genérico captura qualquer outro erro inesperado.
- **Por que usar exceções personalizadas?** Permite tratar diferentes tipos de erro de formas específicas, tornando o código mais claro e fácil de depurar.

5. Menu Funcional (`CinemaUI`)

- **Requisito:** Um menu funcional para gerenciar todas as operações.
- **Implementação:**
    - A classe `CinemaUI` contém o método `main` que inicia o programa.
    - Um loop `do-while` exibe o menu principal e lê a opção do usuário.
    - Um `switch` direciona para métodos de gerenciamento específicos (e.g., `gerenciarAtores()`).
    - Cada método de gerenciamento exibe um submenu (Inserir, Editar, Listar, Consultar, Mostrar, Voltar) e processa a opção escolhida.
    - Utiliza `Scanner` para ler a entrada do usuário.

- Inclui tratamento de exceções (`try-catch`) para lidar com erros de entrada, validação, persistência e opções inválidas.
- **Observação:** No código fornecido, apenas `gerenciarAtores()` está completamente implementado como exemplo. Os demais (`gerenciarClientes` , etc.) precisam ser preenchidos seguindo o mesmo padrão, adaptando os campos solicitados e as chamadas aos métodos estáticos das classes correspondentes.

6. Como Compilar e Executar

1. **Pré-requisito:** Ter o JDK (Java Development Kit) instalado e configurado no seu sistema (variáveis de ambiente `JAVA_HOME` e `PATH`).
2. **Estrutura:** Certifique-se de que os arquivos `.java` estão dentro da estrutura de pacotes correta (`src/modelo` , `src/persistencia` , etc.) e que o diretório `data/` existe (ou será criado na primeira execução).
3. **Compilação (via terminal/cmd):**
   - Navegue até o diretório que contém a pasta `src` .
   - Execute o comando de compilação, especificando o diretório de saída (vamos usar `bin` como exemplo) e o classpath para encontrar as classes: `bash javac -d bin -cp src src/ui/CinemaUI.java src/modelo/*.java src/persistencia/*.java src/excecoes/*.java`
     - Se estiver no Windows, pode ser necessário usar `\` em vez de `/` e `;` em vez de `:` no classpath.
     - Este comando compila todos os arquivos `.java` necessários e coloca os arquivos `.class` resultantes no diretório `bin` , mantendo a estrutura de pacotes.
4. **Execução (via terminal/cmd):**
   - Ainda no mesmo diretório (o que contém `src` e agora `bin`), execute o comando: `bash java -cp bin ui.CinemaUI`
     - `-cp bin` informa ao Java para procurar as classes compiladas no diretório `bin` .
     - `ui.CinemaUI` é o nome completo da classe que contém o método `main` .
5. **Interação:** O menu será exibido no console. Siga as instruções para interagir com o sistema.

7. Preparação para Arguição (Possíveis Perguntas)

O professor pode perguntar sobre:

- **Conceitos:**
  - "Explique o encapsulamento e mostre onde você o aplicou." (R: Atributos privados, getters/setters, validações).
  - "Onde você usou herança? Quais os benefícios?" (R: `Ator` e `Cliente` herdam de `Pessoa`. Reutilização de código, hierarquia).
  - "Por que `Pessoa` é abstrata? O que é um método abstrato?" (R: Modelo base, não faz sentido instanciar só `Pessoa`. Método sem corpo, obriga subclasse a implementar).
  - "O que é associação? Dê exemplos no seu código." (R: Relacionamento entre classes, e.g., `Filme` tem um `Genero`).
  - "Como você implementou o `toString()` em suas classes? Qual a finalidade aqui?" (R: Retorna string formatada com `;` para facilitar a escrita no arquivo TXT).
- **Persistência:**
  - "Como funciona a leitura e escrita nos arquivos TXT?" (R: Classe `Persistencia`, `FileReader`/`FileWriter` ou `Files`, `toString`/`fromString`, separador `;`).
  - "Como você lida com as associações ao salvar e carregar de TXT?" (R: Salva ID/CPF, ao carregar usa `consultar` da classe associada).
  - "Qual a vantagem de ter uma classe `Persistencia` separada?" (R: Centraliza I/O, desacopla modelo da persistência, facilita trocar para banco de dados no futuro).
- **Tratamento de Exceções:**
  - "Quais exceções personalizadas você criou e por quê?" (R: Listar as 5, explicar o propósito de cada uma - `Validacao`, `Persistencia`, `NaoEncontrado`, etc.).
  - "Mostre onde você trata essas exceções." (R: Blocos `try-catch` na `CinemaUI`).
  - "Por que tratar exceções é importante?" (R: Evita que o programa quebre, informa o usuário sobre erros, permite recuperação).
- **Código Específico:**
  - "Explique o fluxo do método `inserir` da classe `Filme`." (R: Cria objeto, chama `toString`, chama `Persistencia.inserirObjeto`).
  - "Como o método `listar` da classe `Ingresso` funciona para carregar `Sessao` e `Cliente`?" (R: Lê linha, pega `idSessao` e `cpfCliente`, chama `Sessao.consultar()` e `Cliente.consultar()`).
  - "Explique o funcionamento do menu principal e de um dos submenus." (R: Loop, `switch`, chamadas aos métodos de gerenciamento, leitura de entrada).

Dicas para a Arguição:

- **Entenda o fluxo:** Saiba como uma operação (e.g., inserir filme) percorre as camadas (UI -> Modelo -> Persistencia).
- **Conheça os conceitos:** Tenha clareza sobre Encapsulamento, Herança, Abstração, Associação.
- **Justifique suas escolhas:** Por que usar `;` como separador? Por que criar exceções personalizadas?
- **Navegue pelo código:** Esteja preparado para mostrar partes específicas do código que ilustrem suas respostas.
- **Seja claro e conciso:** Responda diretamente às perguntas.

Espero que esta explicação detalhada seja útil! Revise o código junto com este documento para solidificar seu entendimento. Boa sorte na arguição!