# Possíveis Perguntas e Respostas para Arguição - Projeto POO Cinema

Este arquivo contém exemplos de perguntas que seu professor pode fazer durante a arguição sobre o projeto, juntamente com sugestões de respostas baseadas no código desenvolvido e nos conceitos de POO aplicados. Use isso como um guia para seus estudos, mas tente entender o porquê de cada resposta.

**Lembre-se:** A melhor preparação é entender o código e os conceitos por trás dele!

## 1. Conceitos Gerais de POO

**P: O que é Encapsulamento e como você o aplicou neste projeto?**

- **R:** Encapsulamento é o princípio de esconder os detalhes internos de um objeto e controlar o acesso aos seus dados. No projeto, apliquei isso declarando os atributos das classes ( `Ator` , `Cliente` , `Filme` , etc.) como `private` (ou `protected` na classe `Pessoa` , conforme pedido). O acesso a esses atributos é feito somente através de métodos públicos `getters` (para ler) e `setters` (para modificar). Além disso, os `setters` e construtores incluem validações para garantir que apenas dados válidos sejam atribuídos aos atributos, protegendo a integridade do objeto.

**P: Explique o conceito de Herança e mostre onde foi utilizado.**

- **R:** Herança permite que uma classe (subclasse) herde características (atributos e métodos) de outra classe (superclasse). Usei herança nas classes `Ator` e `Cliente` , que `extendem` (herdam) da classe abstrata `Pessoa` . Elas herdam os atributos `cpf` , `nome` , `idade` e seus respectivos `getters` / `setters` , evitando a repetição desse código. Cada subclasse adiciona seus próprios atributos específicos ( `registroProfissional` em `Ator` , `rg` e `estudante` em `Cliente` ).

**P: Por que a classe `Pessoa` foi definida como `abstract` ? O que isso significa?**

- **R:** `Pessoa` foi definida como `abstract` porque ela representa um conceito geral, um modelo base para `Ator` e `Cliente` . Não faz sentido criar um objeto diretamente do tipo "Pessoa" neste contexto; queremos criar objetos específicos como `Ator` ou `Cliente` . Ser `abstract` impede a instanciação direta ( `new Pessoa(...)` daria erro) e permite definir métodos abstratos, como o `mostrar()` , que obrigam as subclasses a fornecerem sua própria implementação.

**P: O que é um método abstrato? Qual método abstrato foi definido em `Pessoa`?**

- **R:** Um método abstrato é um método declarado sem corpo (sem implementação, apenas a assinatura seguida de ponto e vírgula) dentro de uma classe abstrata. Ele define um contrato que as subclasses concretas devem cumprir, ou seja, elas são obrigadas a implementar esse método. Em `Pessoa`, o método `public abstract void mostrar();` foi definido como abstrato, forçando `Ator` e `Cliente` a implementarem sua própria maneira de exibir seus dados.

**P: O que é Polimorfismo? Você consegue identificar algum exemplo no código?**

- **R:** Polimorfismo significa "muitas formas". Em POO, permite que objetos de diferentes classes respondam à mesma mensagem (chamada de método) de maneiras diferentes. Um exemplo claro no projeto é a sobrescrita do método `mostrar()`. Tanto `Ator` quanto `Cliente` têm um método `mostrar()`, mas cada um exibe informações diferentes, específicas da sua classe. Se tivéssemos uma lista de `Pessoa` contendo `Ator` e `Cliente`, poderíamos chamar `pessoa.mostrar()` em cada item, e o método correto seria executado dinamicamente. A sobrescrita do `toString()` em todas as classes também é um exemplo.

**P: O que são Associações entre classes? Dê exemplos do projeto.**

- **R:** Associação é um relacionamento entre classes onde um objeto de uma classe "usa" ou "tem um" objeto de outra classe. No projeto, temos várias associações:
  - `Filme` tem uma associação com `Genero` (um filme tem um gênero).
  - `Sessao` tem uma associação com `Filme`.
  - `Ingresso` tem associações com `Sessao` e `Cliente`.
  - `Elenco` tem associações com `Ator` e `Filme`. Isso é implementado tendo um atributo do tipo da classe associada (e.g., `private Genero genero;` dentro da classe `Filme`).

# 2. Persistência em Arquivos TXT

**P: Como os dados são salvos e lidos dos arquivos `.txt`?**

- **R:** Utilizamos uma classe auxiliar `Persistencia` que centraliza as operações de leitura e escrita. Cada objeto, ao ser salvo, é convertido para uma string formatada usando o método `toString()` (com atributos separados por `;`). Essa string é então escrita como uma linha no arquivo `.txt` correspondente (e.g., `atores.txt`). Para ler, a classe `Persistencia` lê as linhas do arquivo. Cada linha é passada para um método estático `fromString()` dentro da classe do modelo correspondente (e.g.,

`Ator.fromString(linha)`), que faz o parse da string, recria o objeto e busca objetos associados se necessário.

**P: Qual o formato usado dentro dos arquivos `.txt`? Por que escolheu esse formato?**

- **R:** Cada linha representa um objeto, e os atributos são separados por ponto e vírgula (`;`). Escolhi o ponto e vírgula porque é um caractere que geralmente não aparece nos dados em si (nomes, títulos, etc.), tornando a separação (split) da linha mais confiável. É um formato simples, semelhante ao CSV, fácil de ler e escrever programaticamente.

**P: Como você lidou com as associações (e.g., o `Genero` dentro de `Filme`) ao salvar e carregar dos arquivos?**

- **R:** Ao salvar um objeto que tem associações (como `Filme`), não salvamos o objeto `Genero` inteiro na linha do `filmes.txt`. Em vez disso, salvamos apenas o identificador único do `Genero` associado (o `idGenero`). Ao carregar um `Filme` do arquivo (no método `Filme.fromString()`), lemos esse `idGenero` da linha e usamos o método `Genero.consultar(idGenero)` para buscar a instância completa do objeto `Genero` correspondente e associá-la ao `Filme` que está sendo criado.

**P: Qual a vantagem de ter uma classe `Persistencia` separada?**

- **R:** Separar a lógica de persistência (leitura/escrita em arquivo) da lógica de negócio (regras das classes `Ator`, `Filme`, etc.) traz vantagens:
  - **Organização:** O código fica mais organizado e fácil de entender.
  - **Reutilização:** Os métodos genéricos de leitura/escrita da classe `Persistencia` são reutilizados por todas as classes de modelo.
  - **Manutenção:** Se no futuro quiséssemos mudar a forma de armazenamento (e.g., usar um banco de dados em vez de TXT), precisaríamos modificar principalmente a classe `Persistencia`, com impacto mínimo nas classes de modelo.
  - **Desacoplamento:** As classes de modelo não precisam saber como os dados são salvos, apenas que eles são salvos.

## 3. Tratamento de Exceções

**P: Quais exceções personalizadas você criou e por que não usou apenas `Exception` ou `IOException`?**

- **R:** Criei 5 exceções personalizadas: `ValidacaoException`, `PersistenciaException`, `NaoEncontradoException`, `MenuOpcaoInvalidaException` e

`ErroInternoException`. Usar exceções específicas em vez de genéricas como `Exception` ou `IOException` permite tratar diferentes tipos de erro de forma mais adequada e clara. Por exemplo, um erro de validação (`ValidacaoException`) pode exigir uma mensagem diferente para o usuário do que um erro ao ler o arquivo (`PersistenciaException`). Isso torna o código mais robusto e a depuração mais fácil.

**P: Onde e como essas exceções são tratadas no código?**

- **R:** As exceções são lançadas (`throw`) nos métodos onde os erros podem ocorrer (e.g., construtores e setters lançam `ValidacaoException`, métodos CRUD lançam `PersistenciaException` ou `NaoEncontradoException`). Elas são tratadas (`catch`) principalmente na classe `CinemaUI`, dentro dos loops dos menus. Usamos blocos `try-catch` para capturar as exceções que podem ser lançadas pelas operações do menu (inserir, editar, etc.). No bloco `catch`, exibimos uma mensagem de erro amigável para o usuário, informando o que deu errado, e permitimos que o programa continue executando.

**P: Por que o tratamento de exceções é importante em um sistema?**

- **R:** É fundamental porque:
  - **Evita que o programa quebre:** Sem tratamento, um erro inesperado poderia encerrar o programa abruptamente.
  - **Informa o usuário:** Permite exibir mensagens claras sobre o que deu errado (e.g., "CPF já cadastrado", "ID não encontrado").
  - **Permite recuperação:** Em alguns casos, o programa pode tentar se recuperar do erro ou oferecer alternativas.
  - **Melhora a robustez:** Torna o sistema mais confiável e capaz de lidar com situações imprevistas.

# 4. Código Específico e Menu

**P: Explique o fluxo de execução quando o usuário escolhe "Inserir Ator" no menu.**

- **R:** 1. O método `gerenciarAtores()` na `CinemaUI` é chamado.
  1. Dentro dele, o submenu é exibido e a opção 1 (Inserir) é lida.
  2. O programa solicita ao usuário CPF, nome, idade e registro profissional.
  3. Um novo objeto `Ator` é criado usando o construtor da classe `Ator` (que valida os dados e pode lançar `ValidacaoException`).
  4. O método `ator.inserir()` é chamado.
  5. Dentro de `ator.inserir()`, ele primeiro tenta `consultar(cpf)` para ver se já existe (lança `PersistenciaException` se existir).

6. Se não existe, ele chama `ator.toString()` para obter a string formatada.
7. Chama `Persistencia.inserirObjeto(ARQUIVO_ATORES, stringDoAtor)` para adicionar a linha ao arquivo `atores.txt`.
8. Se a escrita for bem-sucedida, retorna `true`.
9. A `CinemaUI` exibe a mensagem de sucesso. (Se ocorrer qualquer exceção no processo, ela é capturada pelo `try-catch` em `gerenciarAtores()` e uma mensagem de erro é exibida).

**P: Como funciona o método `listar()` da classe `Filme` para carregar o `Genero` associado?**

- **R:** 1. O método estático `Filme.listar()` é chamado.
    1. Ele chama `Persistencia.listarLinhas(ARQUIVO_FILMES)` para obter todas as linhas do arquivo `filmes.txt`.
    2. Cria uma `ArrayList<Filme>` vazia.
    3. Itera sobre cada `linha` lida do arquivo.
    4. Para cada `linha`, chama o método estático `Filme.fromString(linha)`.
    5. Dentro de `Filme.fromString(linha)`:
        - A linha é dividida (split) pelo `;` para obter os dados (`idFilme`, `titulo`, `duracao`, `idGenero`).
        - O `idGenero` (que é uma string) é convertido para `int`.
        - **Importante:** Ele chama `Genero.consultar(idGeneroConvertido)` para buscar o objeto `Genero` correspondente no arquivo `generos.txt`.
        - Se o `Genero` for encontrado, um novo objeto `Filme` é criado usando o construtor, passando o objeto `Genero` encontrado.
        - O objeto `Filme` criado é retornado.
    6. O `Filme` retornado por `fromString` é adicionado à `ArrayList<Filme>`.
    7. Após processar todas as linhas, a lista completa de `Filmes` (com seus `Generos` carregados) é retornada.

**P: Explique a estrutura geral da classe `CinemaUI`. Como o menu funciona?**

- **R:** A `CinemaUI` tem o método `main` que inicia tudo. Ele entra em um loop `do-while` que continua até o usuário escolher a opção 0 (Sair). Dentro do loop:
    1. O menu principal é exibido (`exibirMenuPrincipal()`).
    2. A opção do usuário é lida (`lerOpcaoInt()`).
    3. Um `switch` (`processarOpcaoPrincipal()`) direciona a execução para o método de gerenciamento correspondente (e.g., `gerenciarAtores()`, `gerenciarClientes()`).
    4. Cada método de gerenciamento (como `gerenciarAtores()`) tem seu próprio loop `do-while` e exibe um submenu específico (`exibirSubMenu()`).

5. Dentro do submenu, outro `switch` processa as operações CRUD (Inserir, Editar, Listar, Consultar, Mostrar).
6. As chamadas aos métodos das classes de modelo (e.g., `Ator.inserir()`, `Cliente.listar()`) são feitas dentro desses `switch`s.
7. Blocos `try-catch` são usados extensivamente para capturar e tratar erros que podem ocorrer durante as operações, exibindo mensagens ao usuário.
8. Uma função `pressioneEnterParaContinuar()` pausa a execução para que o usuário possa ler as mensagens antes de o menu ser exibido novamente.

**Dica Final:** Ao responder, tente sempre conectar sua resposta aos conceitos de POO e aos requisitos da atividade. Mostre que você não apenas fez o código funcionar, mas que entendeu os princípios por trás dele.