Exercícios Práticos de POO: Classes Abstratas e Interfaces

Introdução

Os exercícios a seguir têm como objetivo consolidar a compreensão sobre o design de software utilizando os princípios de Abstração, Polimorfismo e Encapsulamento, focando especialmente no uso correto de Classes Abstratas e Interfaces.

Questões de Implementação Prática

1. Sistema de Impressão (Interface e `if-else`)

Cenário: Uma aplicação precisa imprimir diferentes tipos de documentos (relatórios, faturas, gráficos). Todos os objetos que podem ser impressos devem seguir um contrato comum.

Requisito:

1. Crie uma **Interface** chamada `IImprimivel` com um único método abstrato: `imprimir(int copias)`.
2. Crie uma classe concreta chamada `Fatura` que implemente `IImprimivel`.
3. No método `imprimir(int copias)` da classe `Fatura`, implemente uma lógica que utilize o comando `if-else`:
    - Se o número de cópias for maior que 5, exiba a mensagem: "Imprimindo $[copias]$ cópias em modo rascunho."
    - Caso contrário, exiba a mensagem: "Imprimindo $[copias]$ cópias em modo alta qualidade."

2. Hierarquia de Funcionários (Classe Abstrata e `switch`)

Cenário: Um sistema de Recursos Humanos necessita calcular o bônus anual de diferentes tipos de funcionários (Desenvolvedor, Gerente, Analista). O cálculo do bônus deve ser obrigatório para todos, mas a lógica varia. Todos os funcionários têm um código de cargo comum.

Requisito:

1. Crie uma **Classe Abstrata** chamada `Funcionario` com os campos concretos `nome` (String) e `codigoCargo` (int). O construtor deve inicializar esses campos.
2. Crie um **método abstrato** na classe `Funcionario` chamado `calcularBonificacao()`.
3. Crie uma subclasse concreta chamada `Desenvolvedor` que herde de `Funcionario` e implemente `calcularBonificacao()`.
4. Dentro do método `calcularBonificacao()` da classe `Desenvolvedor`, implemente a lógica de bônus usando o comando `switch` no campo `codigoCargo`:
    - **Case 1 (Desenvolvedor Júnior):** Bônus de 10% do salário-base.
    - **Case 2 (Desenvolvedor Pleno):** Bônus de 15% do salário-base.
    - **Case 3 (Desenvolvedor Sênior):** Bônus de 20% do salário-base.
    - **Default:** Bônus de 5%.

3. Processamento de Dados (Interface, Polimorfismo e `for`)

Cenário: Vários módulos de um sistema de análise de dados (ex: CSV, JSON, XML) precisam ter a capacidade de serem processados.

Requisito:

1. Crie uma **Interface** chamada `IProcessadorDeDados` com o método `processar(String[] dados)`.
2. Crie uma classe concreta `ProcessadorCSV` que implemente `IProcessadorDeDados`.
3. No método `processar(String[] dados)` da classe `ProcessadorCSV`, utilize um laço **`for`** para iterar sobre o array de `dados` (simulando linhas de um arquivo) e exiba cada elemento no console, prefixado pelo seu índice.
4. No método `main` (ou de teste), crie uma lista genérica de `IProcessadorDeDados` e adicione uma instância de `ProcessadorCSV`. Chame o método `processar` no objeto de interface, demonstrando o **Polimorfismo**.

4. Gerenciador de Tarefas (Interface, Classe Abstrata e `while`)

Cenário: O sistema precisa gerenciar o estado de Tarefas. Uma base comum para todas as tarefas é útil, mas a execução varia muito.

Requisito:

1. Crie uma **Classe Abstrata** chamada `TarefaBase` com um campo concreto `concluida` (boolean, inicializado como `false`). Adicione um método concreto `marcarConcluida()`.
2. Crie uma **Interface** chamada `IExecutavel` com o método `executar()`.
3. Crie uma classe concreta `TarefaLonga` que herde de `TarefaBase` e implemente `IExecutavel`.
4. No método `executar()` da classe `TarefaLonga`, utilize um laço **`while`** para simular o progresso da tarefa:
    - Use uma variável `progresso` (int, 0 a 100).
    - O `while` deve rodar enquanto `progresso < 100`.
    - Dentro do loop, incremente o progresso e exiba o valor.
    - Após o `while` ser concluído, chame `marcarConcluida()`.

5. Validação de Credenciais (Classe Concreta Obrigatória)

Cenário: Em uma aplicação de login, a validação de credenciais deve seguir um padrão para garantir que o usuário está apto a acessar o sistema.

Requisito:

1. Crie uma **Classe Abstrata** chamada `ValidadorBase` com um método concreto `validarFormato(String s)` (que sempre retorna `true`).
2. Crie um **método abstrato** na classe `ValidadorBase` chamado `validarAcesso(String senha)`.
3. Crie uma classe concreta `ValidadorSeguranca` que herde de `ValidadorBase`.
4. No método `validarAcesso(String senha)`, implemente a lógica usando **`if-else`** para verificar se a senha tem pelo menos 8 caracteres e contém a letra 'A'. Exiba "Acesso OK" ou "Acesso Negado" com base na condição.

6. Log de Eventos (Interfaces Múltiplas)

Cenário: Um componente de *logging* (registro de eventos) precisa ser capaz de salvar mensagens tanto em um arquivo local quanto em um banco de dados remoto, sendo cada funcionalidade definida por um

contrato.

Requisito:

1. Crie a **Interface** `ISalvavelEmArquivo` com o método `salvarLocal(String log)`.
2. Crie a **Interface** `ISalvavelEmDB` com o método `salvarRemoto(String log)`.
3. Crie uma classe concreta `Logger` que implemente **ambas as Interfaces**.
4. No método `salvarLocal(String log)`, use um `if` simples para checar se a `log` não é vazia e exiba: "LOG ARQUIVO: [log]".
5. No método `salvarRemoto(String log)`, use um `for` para simular 3 tentativas de conexão, exibindo "Tentativa [i] de conexão com DB." antes de exibir "LOG DB: [log]".

7. Obrigatoriedade do Construtor (Classe Abstrata)

Cenário: Você está definindo a base para todos os veículos do sistema. Todo veículo precisa, obrigatoriamente, ter sua cor definida no momento da criação, mesmo que a classe base seja abstrata.

Requisito:

1. Crie uma **Classe Abstrata** chamada `Veiculo` com o campo `cor` (String).
2. Defina um **construtor** em `Veiculo` que aceite e inicialize o campo `cor`.
3. Crie uma subclasse concreta `Carro` que herde de `Veiculo`.
4. No construtor de `Carro`, você deve **obrigatoriamente** chamar o construtor da superclasse (`Veiculo`) usando a palavra-chave `super()`. Explique em um comentário de código a importância dessa chamada.

8. Iteração de Processos (Polimorfismo e Laço `for`)

Cenário: Um *pipeline* de software executa uma série de comandos. Embora todos sejam comandos, a forma como são executados difere (ex: comandos de rede, comandos de arquivo, comandos de sistema).

Requisito:

1. Crie uma **Interface** chamada `IComando` com o método `executar(String[] parametros)`.
2. Crie classes concretas `ComandoRede` e `ComandoArquivo` que implementem `IComando`.
3. No método `main` (ou de teste), crie um `array` ou `List` de objetos do tipo `IComando`. Adicione uma instância de cada classe concreta.
4. Use um laço `for` (preferencialmente *for-each*) para percorrer a coleção e chamar `executar()` em cada elemento. A implementação do `executar()` de cada classe deve apenas exibir uma mensagem que identifique a classe (ex: "Executando Comando de Rede...").

9. Controle de Estoque (Classe Abstrata e `if-else if`)

Cenário: Um sistema de estoque gerencia diferentes tipos de produtos, mas todos têm um método de verificação de disponibilidade obrigatório.

Requisito:

1. Crie uma **Classe Abstrata** chamada `Produto` com o campo concreto `quantidadeEstoque` (int).
2. Crie um **método abstrato** `verificarDisponibilidade(int quantidadeSolicitada)`.
3. Crie uma subclasse concreta `ProdutoAlimenticio` que herde de `Produto`.
4. No método `verificarDisponibilidade()`, implemente a lógica usando a estrutura `if-else if-else`:

- **if:** Se a quantidade solicitada for maior que `quantidadeEstoque`, exiba "Estoque Insuficiente!".
- **else if:** Se a quantidade solicitada for igual a `quantidadeEstoque`, exiba "Últimas Unidades!".
- **else:** Exiba "Disponibilidade OK.".

10. Simulação de Download (Interface e `while`)

Cenário: Um gerenciador de downloads precisa ter uma funcionalidade de "pausar/continuar" para diferentes tipos de transferência de dados (HTTP, FTP).

Requisito:

1. Crie uma **Interface** chamada `ITransferencia` com o método `iniciarDownload(int tamanhoTotal)`.
2. Crie uma classe concreta `TransferenciaHTTP` que implemente `ITransferencia`.
3. No método `iniciarDownload(int tamanhoTotal)`:
   - Use uma variável local `bytesRecebidos` (int) inicializada em 0.
   - Use uma variável booleana `continuar` inicializada como `true`. (Simule uma condição de pausa/interrupção).
   - Use um laço **`while`** que execute enquanto `bytesRecebidos < tamanhoTotal` E `continuar` for `true`.
   - Dentro do `while`, incremente `bytesRecebidos` (simule o recebimento de dados) e exiba o progresso.
   - Adicione uma condição `if` que, ao atingir 50% do total, defina `continuar` como `false` e exiba "Download Pausado Automaticamente!".
   - Ao sair do `while`, use um `if` para exibir se o download foi "Completo" ou "Pausado/Interrompido".