Atividade em sala: Classes Abstratas e Interfaces

Desafios práticos de codificação, com dificuldade progressiva, focados em Classes Abstratas e Interfaces.

Desafio 1: Formas Geométricas (Fácil/Médio)

Objetivo: Praticar a criação de uma **Classe Abstrata** para compartilhar comportamento e forçar a implementação de métodos específicos.

Descrição:
Você precisa modelar um sistema para calcular áreas e perímetros de diferentes formas geométricas. Todas as formas compartilham o conceito de "ter uma área" e "ter um perímetro", mas a fórmula de cálculo para cada uma é diferente.

Requisitos:

1. Crie uma **Classe Abstrata** chamada `Forma`.
2. Esta classe deve ter um atributo `String nome`.
3. Deve ter um construtor que recebe o `nome`.
4. Deve ter um método concreto `exibirNome()` que imprime o nome da forma.
5. Deve ter dois métodos **abstratos**:
    - `double calcularArea()`
    - `double calcularPerimetro()`
6. Crie duas classes **concretas** que herdam de `Forma`:
    - `Retangulo` (precisa de `largura` e `altura` no construtor).
    - `Circulo` (precisa de `raio` no construtor).
7. Implemente os métodos `calcularArea()` e `calcularPerimetro()` em ambas as classes concretas.
    - *Lembrete Círculo: Área = π x raio², Perímetro = 2 x π x raio.*
    - *Lembrete Retângulo: Área = largura x altura, Perímetro = 2 x (largura + altura).*

Desafio (Classe `Main`):
Crie uma classe `Main` que:

1. Cria uma lista de `Forma` (`List<Forma>`).
2. Adiciona um `Retangulo` e um `Circulo` a esta lista.
3. Percorra a lista usando um loop e, para cada forma, imprima seu nome, sua área e seu perímetro. (Isso demonstrará o **polimorfismo**).

Desafio 2: Sistema de Pagamentos (Médio)

Objetivo: Praticar o uso de **Interfaces** para definir um "contrato" e permitir que diferentes classes o implementem de formas únicas.

Descrição:
Você está criando um sistema de e-commerce que precisa processar pagamentos. Existem vários métodos de pagamento (Cartão de Crédito, PIX, Boleto), e cada um tem uma lógica de processamento e taxa diferente.

Requisitos:

1. Crie uma **Interface** chamada `ProcessadorPagamento`.

2. Esta interface deve ter um único método:
    - `boolean processar(double valor)`
    - (Este método deve retornar `true` se o pagamento foi bem-sucedido e `false` se falhou).
3. Crie três classes **concretas** que implementam `ProcessadorPagamento`:
    - `CartaoCredito`: No construtor, recebe `limite`. O método `processar` só retorna `true` se o `valor` for menor ou igual ao `limite`. Se processar, deve imprimir "Pagamento de R$ [valor] aprovado no Cartão de Crédito.".
    - `Pix`: O método `processar` sempre retorna `true` e imprime "Pagamento de R$ [valor] via PIX recebido com sucesso.".
    - `Boleto`: O método `processar` sempre retorna `true` e imprime "Boleto de R$ [valor] gerado. Aguardando pagamento.".
4. Crie uma classe `Checkout`.
5. A classe `Checkout` deve ter um método `finalizarCompra` que recebe dois parâmetros:
    - `ProcessadorPagamento metodo`
    - `double valorTotal`
6. O método `finalizarCompra` deve chamar `metodo.processar(valorTotal)` e imprimir uma mensagem de sucesso ou falha da transação.

Desafio (Classe `Main`):
Crie uma classe `Main` que:

1. Instancia um `Checkout`.
2. Instancia os três tipos de pagamento (`CartaoCredito` com limite 1000, `Pix`, `Boleto`).
3. Simule três compras usando o `Checkout`:
    - Uma compra de R$ 800 no cartão (deve aprovar).
    - Uma compra de R$ 1200 no cartão (deve falhar).
    - Uma compra de R$ 300 no PIX (deve aprovar).
    - Uma compra de R$ 150 no Boleto (deve aprovar).

Desafio 3: Animais e Habilidades (Difícil)

Objetivo: Combinar **Classes Abstratas** (para estado/comportamento comum) e **Interfaces** (para capacidades/habilidades específicas).

Descrição:
Modele um sistema de animais. Todos os animais têm um `nome` e emitem um `som`, mas apenas alguns podem `voar` e apenas alguns podem `nadar`. Um animal pode ter ambas as capacidades (como um pato).

Requisitos:

1. Crie uma **Classe Abstrata** chamada `Animal`.
2. `Animal` deve ter um atributo `String nome`.
3. `Animal` deve ter um construtor que define o `nome`.
4. `Animal` deve ter um método concreto `comer()` que imprime "[nome] está comendo.".
5. `Animal` deve ter um método **abstrato** `fazerSom()`.
6. Crie duas **Interfaces**:
    - `Voador` (com um método `void voar()`).
    - `Nadador` (com um método `void nadar()`).
7. Crie três classes **concretas**:
    - `Cachorro`: Herda de `Animal`. Implementa `fazerSom()` (imprime "Au au!").

- `Pomba`: Herda de `Animal` e implementa `Voador`. Implementa `fazerSom()` (imprime "Pruu pruu!") e `voar()` (imprime "[nome] está voando alto!").
- `Pato`: Herda de `Animal` e implementa **ambas** as interfaces (`Voador` e `Nadador`). Implementa `fazerSom()` (imprime "Quack quack!"), `voar()` (imprime "[nome] está voando baixo.") e `nadar()` (imprime "[nome] está nadando no lago.").

Desafio (Classe `Main`):
Crie uma classe `Main` que:

1. Cria uma lista de `Animal` (`List<Animal>`).
2. Adiciona um `Cachorro`, uma `Pomba` e um `Pato` à lista.
3. Percorra a lista com um loop `for-each`. Para cada `animal` na lista:
   - Chame `animal.comer()`.
   - Chame `animal.fazerSom()`.
   - Verifique se o animal é uma instância de `Voador` (use `instanceof`). Se for, faça um *cast* e chame o método `voar()`.
   - Verifique se o animal é uma instância de `Nadador` (use `instanceof`). Se for, faça um *cast* e chame o método `nadar()`.
   - Imprima uma linha de separação (`---`).

Desafio 4: Dispositivos de Mídia (Múltiplas Interfaces)

Objetivo: Praticar a implementação de **múltiplas interfaces** em uma única classe, permitindo que um objeto desempenhe vários "papéis" independentes.

Descrição:
Você está modelando dispositivos para uma sala de mídia. Alguns dispositivos podem tocar música, outros podem exibir vídeo. Um dispositivo moderno, como uma "Smart TV", pode fazer as duas coisas.

Requisitos:

1. Crie uma **Interface** chamada `ReprodutorAudio`.
   - Ela deve ter um método: `void tocarMusica(String faixa)`.
2. Crie uma **Interface** chamada `ReprodutorVideo`.
   - Ela deve ter um método: `void tocarVideo(String filme)`.
3. Crie três classes **concretas**:
   - `CaixaDeSom`: Implementa **apenas** `ReprodutorAudio`.
     - `tocarMusica(String faixa)`: Deve imprimir "Tocando música: [faixa]".
   - `Projetor`: Implementa **apenas** `ReprodutorVideo`.
     - `tocarVideo(String filme)`: Deve imprimir "Exibindo vídeo: [filme]".
   - `SmartTV`: Implementa **ambas** as interfaces (`ReprodutorAudio` e `ReprodutorVideo`).
     - `tocarMusica(String faixa)`: Deve imprimir "SmartTV tocando áudio: [faixa]".
     - `tocarVideo(String filme)`: Deve imprimir "SmartTV exibindo filme: [filme]".
4. Crie uma classe `SalaDeMidia`.
   - Ela deve ter dois métodos que recebem as interfaces como parâmetros (demonstrando polimorfismo):
     - `void iniciarSessaoDeMusica(ReprodutorAudio dispositivo, String faixa)`
     - `void iniciarSessaoDeFilme(ReprodutorVideo dispositivo, String filme)`

Desafio (Classe `Main`):
Crie uma classe `Main` que:

1. Instancia uma `CaixaDeSom`, um `Projetor` e uma `SmartTV`.
2. Instancia uma `SalaDeMidia`.
3. Use a `SalaDeMidia` para:
    - Tocar música na `CaixaDeSom`.
    - Tocar vídeo no `Projetor`.
    - Tocar música na `SmartTV`.
    - Tocar vídeo na `SmartTV`.

Desafio 5: Sistema de Banco (Abstrata que Implementa Interface)

Objetivo: Praticar o padrão onde uma **Classe Abstrata** implementa uma **Interface** para fornecer um comportamento padrão, ao mesmo tempo que força suas subclasses a implementar outros métodos abstratos.

Descrição:
Você está modelando contas bancárias. Todas as contas precisam de um "histórico de transações" (um contrato/interface). No entanto, todas as contas também compartilham atributos (saldo, número) e uma lógica de depósito. O que muda é a regra de *saque*.

Requisitos:

1. Crie uma **Interface** chamada `Registravel`.
    - Ela deve ter um método: `void adicionarAoHistorico(String transacao)`.
2. Crie uma **Classe Abstrata** chamada `ContaBancaria` que `implements Registravel`.
3. `ContaBancaria` deve ter:
    - Atributos `protected`: `int numeroConta`, `double saldo`.
    - Um `ArrayList<String> historico` para guardar as transações.
    - Um construtor que inicializa `numeroConta`, `saldo` e o `ArrayList`.
    - **Implementação Concreta (da Interface):**
        - O método `adicionarAoHistorico(String transacao)` deve ser implementado aqui. Ele simplesmente adiciona a string ao `ArrayList historico`.
    - **Método Concreto (da Própria Classe):**
        - `void depositar(double valor)`: Deve aumentar o `saldo` e chamar `adicionarAoHistorico("Depósito: R$" + valor)`.
    - **Método Abstrato (da Própria Classe):**
        - `abstract boolean sacar(double valor)`: Este método será implementado pelas subclasses, pois a regra de saque muda.
4. Crie duas classes **concretas** que herdam de `ContaBancaria`:
    - `ContaCorrente`:
        - No construtor, chama `super()`.
        - Implementa `sacar(double valor)`: Pode sacar se `(saldo >= valor)`. Se sacar, deve chamar `adicionarAoHistorico("Saque: R$" + valor)` e retornar `true`.
    - `ContaPoupanca`:
        - No construtor, chama `super()`.
        - Implementa `sacar(double valor)`: Pode sacar se `(saldo >= valor)`. Cobra uma taxa de R$ 1,00 por saque. Se sacar, deve chamar `adicionarAoHistorico("Saque: R$" +

`valor + " (Taxa: R$1.00)")` e retornar `true`.

Desafio (Classe `Main`):
Crie uma classe `Main` que:

1. Cria uma `ContaCorrente` e uma `ContaPoupanca`.
2. Armazena ambas em uma lista de `ContaBancaria` (`List<ContaBancaria>`).
3. Para cada conta na lista:
    - Deposite R$ 1000,00.
    - Tente sacar R$ 300,00.
    - Tente sacar R$ 900,00 (deve falhar).