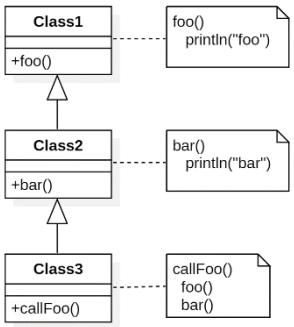
Roteiro 03 – Exercícios de Fixação

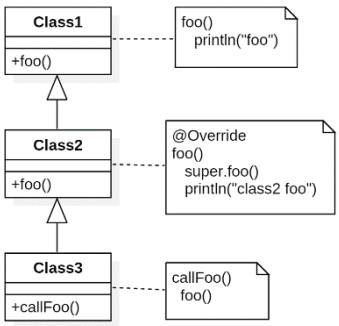
1. Por que a abordagem que utiliza atributos para implementar constantes em Java não quebra o princípio da proteção aos dados?

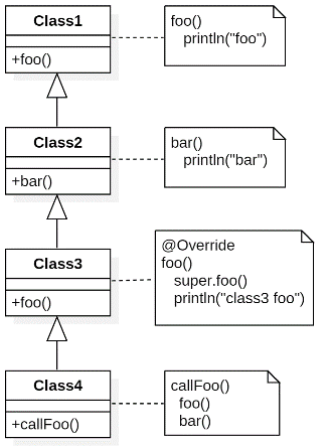
2. Por que, quando utilizamos a abordagem citada na questão anterior, a constante só faz sentido se for pública?

3. O que é uma regra de negócio?

4. Implemente os programas abaixo e explique o funcionamento de `super` em cada um.







Depois...

   a. No primeiro programa, altere a chamada de `foo()` em `callFoo()` para `super.foo()`. O que aconteceu? Explique.
   b. No segundo programa, altere a chamada de `super.foo()` em `foo()` para `foo()`. O que aconteceu? Explique.
   c. No segundo programa, altere a chamada de `foo()` em `callFoo()` para `super.foo()`. O que aconteceu? Explique.
   d. No terceiro programa, altere a chamada de `foo()` em `callFoo()` para `super.foo()`. O que aconteceu? Explique.

5. Você notou que, até o momento, não utilizamos métodos de leitura ou escrita durante a implementação de classes de entidade (**Person**, **Student**, **Employee**, **Professor**)? Explique por que?

6. Explique como o polimorfismo implementado neste roteiro funciona.

7. O que é ligação tardia (também conhecida por ligação dinâmica, ligação em tempo de execução, *late binding*, *dynamic binding* ou *runtime binding*) no contexto da orientação a objetos?

8. Qual a relação da herança com o polimorfismo de inclusão (de subtipo)?

9. Revise a implementação do padrão **Telescoping** em **Employment** e **Student**. Verifique se há necessidade de todos os construtores implementados. Veja quais atributos são realmente obrigatórios (como dica, pense se faria sentido um objeto existir sem aquela informação).

10. Agora que você já trabalhou com redefinição de operações (***override***) e com o modificador **final**, volte à última questão do roteiro 02. Veja se você sabe explicar o cenário e apresentar soluções.

11. Defina o relacionamento de associação? Apresente um exemplo diferente do visto em sala, discutindo os aspectos de cada ponta da associação (nome de papel, navegabilidade, multiplicidade).

Roteiro 03 – Exercícios de Fixação

12. Modele e implemente um programa para embaralhar cartas. Considere dois tipos de enum: Suit (Naipe) e Value (Valor).

    - Suit: spades (espadas), hearts (copas), clubs (paus) e diamonds (ouros).
    - Value: ace (az), 2, 3, 4, 5, 6, 7, 8, 9, 10, jack (valete), queen (rainha) e king (rei).

    Defina uma classe Card (Carta) com dois atributos: Suit e Value. Considere também os respectivos getters e setters.

    Defina uma classe Deck (Baralho), a qual deve conter 52 cartas (objetos Card). As cartas devem ser criadas por meio de um algoritmo (não faça um new para cada carta manualmente no código). Note que são 13 cartas de cada naipe. A classe Deck deve ter as seguintes operações:

    - void shuffle() – embaralha as cartas.
    - Card get(i) – retorna a carta que estiver na posição i.
    - Card[] getAll() – retorna um vetor com todas as cartas, conforme a posição delas no momento da invocação.

    Crie uma classe de teste para realizar as seguintes atividades, considerando um mesmo objeto Deck:

    - Imprimir todas as cartas antes de embaralhá-las.
    - Imprimir todas as cartas depois de embaralhá-las.

    Modele e implemente um jogo de adivinhação (GuessGame) que utilize as classes anteriores. O jogo deve aleatoriamente (ver classe Random em https://docs.oracle.com/javase/8/docs/api/java/util/Random.html) selecionar uma carta do baralho e perguntar ao jogador que terá 3 chances para adivinhar a carta. Se ele acertar na primeira, ele ganha 500 pontos. Se ele errar, antes de pedir uma nova carta (tem que ser diferente da anterior), você apresentará uma dica (naipe). Se ele acertar na segunda tentativa, ele ganha 200 pontos. Se ele errar novamente, mostre uma nova dica (se a carta é maior ou igual ou menor ou igual do que 7). Se ele acertar, ganha 50 pontos. Antes de terminar, pergunte se o jogador que jogar novamente. Modele uma classe UserInterface que fará todas as entradas e saídas do programa (para ler dados do console, você pode utilizar a classe Scanner - http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html) e saídas. A classe Game (Jogo) não deve ter entradas e saídas (ou seja, interface com o usuário). Você pensar também em uma classe Player para guardar informações e a pontuação do jogador. A ideia é que esta mesma classe poderia ser utilizada em uma aplicação móvel, web ou desktop. O que mudaria seria a interface com o usuário.