MAC 321 — Prova 2 — Turma 1

Marcelo Finger

Instruções

Essa prova é composta de 4 questões, cada uma valendo o indicado na própria questão. Exercícios com múltiplos itens tem seu valor dividido igualmente pelo número de itens. A prova tem duração de 100 minutos.

Cada exercício dessa prova vem acompanhado de arquivos de código-fontes auxiliares no seguinte formato: `br.usp.ime.p2.ex<número>`. Importe esses arquivos no Eclipse (no programa, clique em `File->Import-> Existing Projects into Workspace->Select archive file->Browse->Finish`) antes de começar a resolver a prova!

Nos exercícios, é necessário que o aluno escreva o código fazendo os testes escritos em JUnit 4 já inclusos passem sem modificações no código dos testes. O aluno não deve alterar os nomes das classes, variáveis, valores ou qualquer outra parte dos testes.

No Exercícios 4 é necessário que o aluno escreva testes de unidade usando JUnit 4 para os exercícios e, eventualmente, alguma outra parte do código.

Além do arquivo de testes fornecido, existe uma outra versão, não disponível para os alunos, porém com testes semelhantes. O código precisa passar em ambos, para evitar casos em que os alunos façam os testes passarem usando retornos estáticos. Exemplo: o exercício pede a criação de um método `public static olaMundo(String nome)` que deve devolver a String `"Ola <nome>!"`. O teste é `olaMundo("Marcelo")` e espera como saída o String `"Ola Marcelo!"`. O aluno então escreve o seguinte código:

```
1 // Hello.java
2 public class Hello {
3   public static olaMundo(String nome) {
4     return "Ola␣Marcelo!";
5   }
6 }
```

Apesar de o código acima passar nos testes apresentados, ele não cumpre o pedido pelo exercício, e ele não passará nos testes não disponíveis para os alunos (pois usaremos outros parâmetros de entrada para o método), então o aluno zeraria o exercício. Os exercícios devem ser feitos individualmente! Caso seja detectado plágio (dois alunos com códigos parecidos, por exemplo), ambos zerarão a prova, sem possibilidade de SUB. O aluno pode consultar diferentes fontes de informação, incluindo páginas da Internet, a documentação oficial do Java e o Stack Overflow, porém evite copiar e colar códigos da Internet, já que isso pode ser considerado plágio também! Boa sorte!

Exercício 1 (3 pontos)

Criar uma classe `PilhaLimitada` que herde da classe `Stack<Number>`. Essa pilha deverá conter apenas elementos da classe `Number` e não deve permitir que o somatório de seus elementos infrinja os limites inferior (negativo) e superior (positivo) definidos como parâmetros no construtor. Você deve implementar o método `double total()` que retorna o valor atual do somatório dos elementos na pilha. **Um elemento só será inserido na pilha se o valor do somatório após a inserção se mantiver dentro dos limites definidos.**

A classe `PilhaLimitada` deve fazer passar os testes da classe `PilhaLimitadaTest.java` fornecida em anexo. Observe que você não deve mudar os testes.

Atenção: Valores negativos podem ser inseridos na pilha. Qualquer subclasse de `Number` poderá ser fornecida como entrada para esse conjunto.

Exercício 2 (1 ponto)

Refatore o código abaixo. Extraia os métodos, renomeie as variáveis, retire os comentários que achar necessário, etc. Seu programa refatorado deve imprimir as mesmas coisas (ou seja, o que o método `main()` imprime deve ser equivalente) e passar pelos testes fornecidos.

```
public class Esfera {
  public static void main(String[] args) {
    // Raio da esfera
    double x = 5.5;
    // Valor do pi
    double y = 3.1415926535;

    // Calculando a área da esfera
    double a = x * y * x * 4;
    System.out.println("Área da esfera = " + a);

    // Calculando o volume da esfera
    double b = 4/3 * x * y * x * x;
    System.out.println("Volume da esfera = " + b);
  }
}
```

Exercício 3 (3 pontos)

No jogo do jó-kem-pô existem dois jogadores em que cada um pode jogar uma das seguintes três mãos: jó (0 dedos), kem (2 dedos) e pô(5 dedos). Jó ganha de Kem, Kem ganha de Pô e Pô ganha de Jó. Jogadas com mãos iguais são consideradas empates.

Defina as classes necessárias para representar os três tipos de mãos deste jogo, e use o esqueleto da classe Jogo para guardar as jogadas realizadas por cada um dos dois jogadores e crie nela métodos e atributos auxiliares que achar necessários para implementar o jogo. Não é obrigatório que as jogadas sejam introduzidas de forma alternada.

Adicionalmente, esta classe tem as funcionalidades de jogar uma mão (tanto para o jogador 1 como para o jogador 2), que deve lançar uma exceção do tipo `MaoInvalida` caso a mão jogada não faça nenhum sentido para o jogo (por exemplo, um número inválido de dedos), e saber quantos jogos foram ganhos, empatados e perdidos pelo jogador 1 em relação ao jogador 2, onde se o número de jogadas do jogador 1 não for igual ao número de jogadas do 2 então deve-se lançar uma exceção do tipo `PartidaInvalida`. Atenção que a solução tem que ser expansível a novos tipos de mãos sem que seja necessário alterar nenhum dos métodos já realizados, ou seja, se novas jogadas forem introduzidas, alterado-se a ordem das mãos vencedoras, `escreveResultado()` não precisa ser alterado.

Esqueleto da classe Jogo:

```
public class Jogo {
        public Jogo() {}
        /**
         * Adiciona a mão jogada pelo jogador 1.
         *
         * @param mao
         *            A mão jogada pelo jogador 1.
```

```
         *
         * @throws MaoInvalida
         *             Essa exceção é lançada se for jogada uma mão sem
         *             sentido para o jogo.
         **/
        public void adicionaMaoDoJogador1(Mao mao) throws MaoInvalida {...}
        /**
         * Adiciona a mão jogada pelo jogador 2.
         *
         * @param mao
         *            A mão jogada pelo jogador 2.
         *
         * @throws MaoInvalida
         *             Essa exceção é lançada se for jogada uma mão sem
         *             sentido para o jogo.
         **/
        public void adicionaMaoDoJogador2(Mao mao) throws MaoInvalida {...}
        /**
        * Escreve o resultado no stdout [1 (vitória), 0 (empate),
        * -1 (derrota) relativamente ao jogador 1] para cada jogada
        * realizada pelos dois jogadores.
        *
        * @throws PartidaInvalida
        *       Essa exceção é lançada caso os dois jogadores tenham jogado
        *     um numero distinto de jogadas.
        **/
        public void escreveResultado() throws PartidaInvalida {...}
}
```

Exercício 4 (3 pontos)

Dado o código abaixo:

```
package br.usp.ime.p2.ex4;

public abstract class Refrigerante {
    public String nome;
    public String slogan;
    public boolean diet;
}

class CocaCola extends Refrigerante {
    public CocaCola() {
        this.nome = "Cola␣Cola";
        this.slogan = "Abra␣a␣felicidade";
        this.diet = false;
    }
}

class CocaColaZero extends CocaCola {
    public CocaColaZero() {
        super();
        this.nome = "Cola␣Cola␣Zero";
        this.diet = true;
    }
}
```

```
24
25 class Pepsi extends Refrigerante {
26     public Pepsi() {
27         this.nome = "Pepsi";
28         this.slogan = "Pode␣ser";
29         this.diet = false;
30     }
31 }
32
33 class PepsiZero extends Pepsi {
34     public PepsiZero() {
35         super();
36         this.nome = "Pepsi␣Zero";
37         this.diet = true;
38     }
39 }
```

Usando o padrão de projeto *Factory*, crie um `FactoryRefrigerante` com um método `getRefrigerante(String nome, boolean diet)` que retorne um objeto da classe correspondente. Dica: para facilitar, Coca-Cola será sempre passado como "CocaCola" e Pepsi como "Pepsi" para o método `getRefrigerante()`.

Crie testes em JUnit 4 para `getRefrigerante()` que teste a criação de 4 tipos diferentes de refrigerante: coca ou pepsi, diet ou normal. Use o método `getClass()` para verificar se a classe concreta criada pela *Factory* é a correta.