Computação Orientada a Objetos: Lista de Exercícios

> Respostas aos exercícios propostos

1.

a) Pacotes permitem agrupar classes, interfaces e mesmo outros pacotes que compartilham de alguma relação arbitrária entre si. O conteúdo deste pacote pode então ser incorporado ao código do usuário por meio do comando `import`. Agrupar trechos de códigos desta forma permite organizá-los e reutilizá-los conforme for conveniente e, desta forma, evitar a fadiga com sucesso.

b) Mecanismos para tratamento de exceções permitem compartimentalizar o código a ser executado em casos excepcionais em um bloco `catch` distinto daquele da lógica principal, o `try`. Não apenas isso, esta permite adequadamente informar o usuário da ocorrência de erros e, onde possível, de repará-los para dar prosseguimento a execução.

c) Métodos, atributos ou classes que implementam um tipo genérico permitem que o tipo de um parâmetro seja também um parâmetro, de tal forma que um mesmo método, atributo ou classe seja capaz de tratar tipos variados de dados sem que por isso seja necessário reescrever várias versões do mesmo código onde o tipo do parâmetro é o único fator de mudança.

2.

A alteração direta dos valores de atributos pode ter consequências imprevisíveis sobre os métodos aqueles que destes atributos dependem. Métodos para inserção, os `setters`, permitem que a inserção de dados se dê de maneira controlada, dentro de parâmetros e em campos especificados como sendo aceitáveis ao funcionamento da classe. Por exemplo, estipulando um limite de tamanho para strings à serem armazenadas, e capturando com uma exceção o caso do usuário repassar uma string de tamanho inválido. Enquanto isso, métodos para extração, os `getters`, permitem acessar os dados considerados pertinentes, isto apesar do acesso direto a estes ser protegido do usuário.

3.

A composição consiste na adição de uma classe enquanto atributo doutra. Essa segunda classe então, pode, com os métodos adequados, fazer uso das funcionalidades da primeira

sem que a segunda consista em uma extensão desta. Isso é útil quando não se deseja implementar na segunda classe todos os métodos da primeira, apenas alguns que sejam convenientes ao seu funcionamento.

4.

Uma forma de implementar esta funcionalidade seria acrescentando à classe um atributo contador com o modificador `static`, a ser incrementado quando um novo objeto é instanciado. Isto pois atributos `static` são compartilhados por todos os objetos de uma mesma classe, desta forma um mesmo contador é incrementado a cada novo objeto criado.

5.

Vê-se que o comando `import` foi passado da seguinte forma:

```
import projeto.mat.*;
import projeto.graficos.*;
```

Um pacote nada mais é do que uma pasta contendo arquivos `.class`. Ao especificar o caminho de uma classe substituindo `/` por `.` é possível utilizá-la. Ao realizar o `import` a raiz do caminho, entretanto, é a pasta em que o arquivo em questão se encontra. Assim sendo, os *imports* acima descritos estão buscando encontrar classes em uma pasta projeto dentro da pasta projeto, a qual não existe. Por isso a falha.

6.

Uma maneira bastante comum é implementar os pacotes aqueles que estão disponíveis com a linguagem, aqueles "*built in*". Estes estão listados aqui, uma porção significativa destes encontra-se listado sob o diretório `java`, como em "`import java.math`". Outra maneira é adicionando outros diretórios ao `classpath` no momento da compilação do código utilizando a flag `--classpath`. Por exemplo,

```
javac --classpath ".:/usr/lib/jvm/<java folder>/lib:/home/name/Desktop"
MyClass.class
```

Permitiria adicionar ao projeto arquivos de classes encontradas no Desktop.

7.

```
> java Ex8.java
[master +5] 18:29:02
Ex8: inicio.
Digite dois números: 20 16
Resultado = 5
Finally.
Ex8: fim.

> java Ex8.java
[master +5] 18:29:28
Ex8: inicio.
Digite dois números: 8 8
Divisão por zero!
Finally.
Ex8: fim.

> java Ex8.java
[master +5] 18:31:03
Ex8: inicio.
Digite dois números: 23 9.5
Finally.
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:943)
        at java.base/java.util.Scanner.next(Scanner.java:1598)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2263)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2217)
        at Ex8.metodo(Ex8.java:10)
        at Ex8.main(Ex8.java:24)
> java Ex8.java
[master +5] 18:31:12
Ex8: inicio.
Digite dois números: 10 20
Finally.
Ex8: fim.

>
```

8.

O bloco `finally` é tal que sempre é executado, ocorrendo ou não excessões.

9.

throw: comando utilizado para explicitamente lançar uma exceção de determinado tipo. Por exemplo:

```
throw new ArithmeticException("Divisão por zero!");
```

A exceção acima pode ser utilizada para indicar um erro de cálculo aritmético, e desta forma ser tratado pelo bloco `catch (ArithmeticException e)` seguinte.

throws: palavra chave utilizada na assinatura de um método para indicar que este pode lançar uma exceção de um dos tipos listados, os quais não são tratados por ela própria. Desta forma, a chamada deste método pode implementar (ou não) blocos catch para tratar a exceção adequadamente.

10.

Esta necessita estender a classe `Throwable` ou `Exception` (pois esta última também estende a `Throwable`).

11.

Exceções verificadas: Aquelas cuja resolução é verificada em tempo de compilação. O método que a emite **necessita** ou resolvê-la em um bloco `catch` ou especificar que este a lança com a palavra chave `throws` para que aquele realizando a chamada do método o faça.

Exceções não verificadas: aquelas que não são verificadas em tempo de compilação, ficando a critério do programador se estas deverão ser tratadas ou se estas interromperão a execução do programa simplesmente (quando elas o fazem geralmente é resultado de uma falha de programação). No Java, todas as exceções derivadas a partir das classes `Error` ou `RuntimeException` não são verificadas.

12.

a. Exceções podem ocorrer múltiplas vezes ao longo da execução de um programa qualquer. Emitir uma exceção enquanto um *Exception*, simplesmente, não permite maior diferenciação desta com relação a demais exceções. Isso, de tal forma que, se o usuário deseja tratá-la de maneira específica, este necessitará criar um par de blocos `try` e `catch` somente para a chamada da classe/método, o que reduziria a clareza deste código.

b. Uma extensão do tipo *Exception*, por outro lado permite a diferenciação da exceção e a sinaliza para seu tratamento.

c. Uma extensão do tipo *RuntimeException* permite a diferenciação da exceção, mas **não** a sinaliza para ser tratada. Estas são usualmente relegadas à exceções as quais espera-se que o programa não seja capaz de se recuperar ou tratar adequadamente (embora isso seja possível), como realizar uma divisão por zero, acessar o endereço de memória nulo, acessar um índice além do tamanho de um dado vetor.

Assim o sendo, para atender a proposta descrita pelo enunciado, a opção **b** é a mais adequada das três.

13 e 14.

```java
public class testQueue {

    public static void testInt() {
        int i, x;
        Queue<Integer> q = new Queue<Integer>(10);

        q.insert(10);
        q.insert(20);
        q.insert(30);
        q.insert(40);
        q.insert(50);
        System.out.println(q);

        for (i = 0; i < 3; i++) {
            x = q.remove();
            System.out.println("Removed element: " + x);
        }
        System.out.println(q);
    }

    public static void testString() {
        int i;
        String s;
        Queue<String> q = new Queue<String>(10);

        q.insert("Este");
        q.insert("é");
        q.insert("um");
        q.insert("teste");
        q.insert(", ok?");
        System.out.println(q);

        for (i = 0; i < 3; i++) {
            s = q.remove();
            System.out.println("Removed element: " + s);
        }
        System.out.println(q);
    }

    public static void main(String[] args) {
        testInt();
        System.out.println("----------------------------------------------------
----------------------");
        testString();
    }
}
```

```
public class Queue<T> {
    private int end;
    private T[] queue;

    @SuppressWarnings("unchecked")
    public Queue(int size) {
        queue = (T[]) new Object[size];
        end = 0;
    }

    public void insert(T item) {
        queue[end] = item;
        end++;
    }

    public T remove() throws IllegalStateException {
        int i;
        T item;

        if (end == 0) {
            throw new IllegalStateException("Queue is empty");
        }

        item = queue[0];

        for (i = 0; i + 1 < end; i++) {
            queue[i] = queue[i + 1];
        }
        end--;
        return item;
    }

    public String toString() {
        int i;
        String s = "Queue:";

        for (i = 0; i < end; i++) {
            s += (" " + queue[i]);
        }
        return s;
    }
}
```

15.