Programação Orientada por Objetos

Exame Época Normal, 3 de julho de 2019 – 09:30

A duração do exame é de 2 horas, sem tolerâncias.
O aluno deve permanecer na sala pelo menos 30m.
Responda aos grupos 1-2 e 3-4 em folhas separadas. Identifique todas as folhas.

Grupo 1: (4 Valores)

1.1 Um atributo **protected** de uma classe **C1** pertencente ao pacote **p** é um atributo visível a partir de: (Nota: pode existir mais do que uma resposta correta)

a) uma classe **C2** que não herda de **C1** e que pertence a outro pacote
b) **uma classe C3 que herda de C1 e que pertence a outro pacote**
c) **uma classe C4 que herda de C1 e que pertence ao mesmo pacote**
d) **uma classe C5 que não herda de C1 e que pertence ao mesmo pacote**

1.2 Se uma classe **Cavalo** que herda da classe **Animal**, não tiver construtor então:

a) Durante a construção de um objeto de tipo **Cavalo**, um erro de execução acontece se a classe **Animal** não tiver um construtor definido.
b) Não é possível criar um objeto da classe **Cavalo**.
c) **Durante a criação de um objeto de tipo Cavalo os atributos privados de Animal** são alocados **em memória e inicializados pelo(s) construtor(es) da classe Animal.**

1.3 Considerando que a classe **Estudante** deriva da classe **Pessoa**, qual das seguintes sequências de instruções está correta?

a) `Pessoa paulo = new Pessoa(); Estudante s = (Estudante) paulo; Pessoa s = p;`
b) `Estudante paulo = new Estudante(); Pessoa p = paulo; Estudante s = p;`
c) `Estudante paulo = new Estudante(); Pessoa p = paulo; Estudante s = (Estudante) p;`
d) `Pessoa paulo = new Pessoa(); Estudante s = (Estudante) paulo; Pessoa s = (Pessoa) p;`

1.4 O que é o conceito de Tipo Genérico?

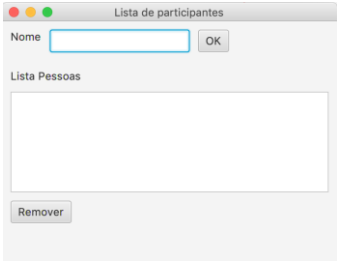
a) Um conceito que permite ter uma classe, um método ou uma coleção fixa para cada uso.
b) Um conceito que permite ter um código chamado em cada classe de forma idêntica.
c) **Um conceito que permite não especificar um tipo particular para uma classe, coleção ou método para ter um código reutilizável.**
d) Um conceito que permite que uma classe não tenha uma subclasse.

1.5 Em JavaFX, um nó contentor pode conter apenas objetos gráficos da UI:

a) **Verdeiro**
b) **Falso**

1.6 Na figura mostrada, qual o tipo de contentor e a ordem em que os elementos foram adicionados?

a) **BorderLayout, Label, TextField, Button, Label, ListBox, Button**
b) **VBox, HBox, VBox**
c) **GridPane, Label, TextField, Button, VBox**
d) **VBox, GridPane**



1.7 Considere o caso em que o programa tenta gravar dados num disco. O programa deve antecipar que o disco pode estar cheio e prever uma exceção que contempla a verificação do sucesso da operação. Esta exceção é de que tipo?

a) checked exception.
b) **unchecked exception**.
c) da classe **`Error`**

1.8 Para que uma subclasse possa ser concreta, deve implementar:

a) Apenas os métodos concretos da superclasse
b) Os métodos concretos da superclasse, se está classe implementar uma interface
c) Todos os métodos da superclasse
d) Todos os métodos abstratos herdados.

Grupo 2: (8,0 valores)

Pretende-se desenvolver uma aplicação para gerir um laboratório inteligente onde a temperatura e a luminosidade podem ser regulados automaticamente. O laboratório é constituído por vários postos de trabalho (**`WorkStation`**) equipados com sensores: um sensor de presença para saber se alguém está a trabalhar na estação e um sensor de luminosidade para saber se é preciso ligar ou desligar o candeeiro do posto de trabalho. Para o protótipo inicial foram criadas as classes **`Laboratory`**, **`WorkStation,`** **`Sensor`**, **`AirConditioner`**, **`TemperatureSensor`** e **`SmartLab`**. assim como a interface **`Adjustable`** da **Figura 1**. Na **Figura 2** encontra-se algum código de teste da aplicação na classe **`SmartLab`** e o *output* produzido pela sua execução. Tendo em conta os tipos mencionados e a execução da **`Figura 2`** complete o código da aplicação de acordo com o que é pedido nas alíneas seguintes:

2.1 (2.5) A classe **`Sensor`**, mostrada na Figura 1, é abstrata e tem os atributos e métodos comuns às suas classes derivadas. A classe **`TemperatureSensor`** é uma das suas subclasses e representa um sensor de temperatura instalado no laboratório. Defina agora as classes **`LightSensor`** e **`PresenceSensor`** por forma a representar dois sensores do tipo **`StationSensor`** instalados nas estações de trabalho. Um para indicar a presença de um trabalhador nessa secretária e outro para representar um sensor de Luz. Defina o construtor e os métodos seletores necessários para devolver um valor aleatório, de acordo com as características da informação específica de cada sensor:

Sensor de presença: presente/ausente.

Sensor de luz: um valor entre 3 e 400 lumens.

Sensor de estação: guarda a referência da estação onde está instalado.

```
public abstract class StationSensor extends
Sensor {
    private WorkStation workstation;
    public StationSensor(int id, WorkStation
station){
        super(id);
        this.workstation = station;
    }
    public WorkStation getWorkStation(){
        return workstation;
    }
}
```

```
public class LightSensor extends
StationSensor{
    private static final double
RANGE_MIN = 3.0;
    private static final double
RANGE_MAX = 400.0;

    public LightSensor(int id,
WorkStation station){
        super(id,station);
    }

    public double getLight(){
        Random r = new Random();
        double randomValue = RANGE_MIN +
(RANGE_MAX - RANGE_MIN) *
r.nextDouble();
        return randomValue;
    }
}
```

```
    public class PresenceSensor extends
StationSensor{
      public PresenceSensor(int id, WorkStation
station){
        super(id,station);
      }
    public boolean getPresence(){
        Random randomno = new Random();
        boolean value = randomno.nextBoolean();
        return value;
    }
}
```

2.2 (1.0) O laboratório está equipado com aparelhos elétricos, como um ar condicionado ou um candeeiro de um posto de trabalho, que podem ser ligados ou desligados. O equipamento inicial do laboratório é o ar condicionado e o sensor de temperatura. A classe **`Sensor`**, mostrada na **Figura 1**, é abstrata e tem os atributos e métodos comuns às suas classes derivadas. Sendo assim, complete a classe fornecendo o seguinte código:

- Defina a interface **`switchable`**, que contem um método **`switchOnOff`** para ligar e desligar dispositivos elétricos como um ar condicionado ou um candeeiro de um posto de trabalho.
- Sabendo que o candeeiro de um posto de trabalho é representado pelo atributo **`isLightOn`**, altere a classe para implementar que implemente esta interface.
- Faça a mesma alteração à classe **`AirConditioner`** para que implemente também a mesma interface.

```
public interface switchable {

    public void switchOnOff();
}

public class WorkStation implements switchable{
…
    public void switchOnOff (){
        isLightOn = !isLightOn;
    }
}
```

```
public class AirConditioner implements switchable{
…
    public void switchOnOff (){
        isOn = !isOn;
    }
}
```

2.3 (3.0) A classe **`Laboratory`** representa o laboratório com as estações de trabalho, os sensores e o ar condicionado. Inicialmente, o laboratório não contém estações de trabalho. Sendo assim, complete a classe fornecendo o seguinte código:

- Adicione um atributo que permite guardar o conjunto dos aparelhos elétricos que podem ser ligados/desligados;
- Defina o construtor que recebe o nome do laboratório. O equipamento inicial do laboratório é composto pelo ar condicionado e o sensor de temperatura. Este equipamento deve ser adicionado às respetivas coleções.
- Defina o método **`addWorkStation`**, que permite adicionar uma estação de trabalho ao laboratório, com os seus respetivos sensores de luz e de presença.
- Defina o método **`showStatus`** para mostrar o estado das estações de trabalho do laboratório.
- Defina o método **`switchEverything`** para ligar e desligar todos os dispositivos elétricos do laboratório ao mesmo tempo.

```
 //2.3.1 atributo
private Set<switchable> equipment;

//2.3.2 construtor
public Laboratory(String name){
        this.name=name;
        workStations = new HashMap();
        equipment = new HashSet();
        sensors = new HashSet();
        this.temperatureSensor = new
 TemperatureSensor(sensors.size()+1);
        sensors.add(temperatureSensor);
        airconditioner = new AirConditioner();
        equipment.add(airconditioner);
    }

 //2.3.5
    public void switchEverything(){
        for(switchable e : equipment){
            e.switchOnOff();
        }
    }
```

```
    // 2.3.2
    public void addWorkStation(){
        int numberOfSensors = sensors.size();
        int idStation = workStations.size()+1;
        WorkStation workStation = new WorkStation(idStation);
        workStations.put(idStation,workStation);
        LightSensor lightSensor = new LightSensor(numberOfSensors+1,workStation);
        PresenceSensor presenceSensor = new
           PresenceSensor(numberOfSensors+2,workStation);
        workStation.setLightSensor(lightSensor);
        workStation.setPresenceSensor(presenceSensor);
        sensors.add(lightSensor);
        sensors.add(presenceSensor);
        equipment.add(workStation);
    }

//2.3.4
    public void showStatus(){
        System.out.println("Luzes:");
        for(WorkStation s: workStations.values()){
            System.out.println(s);
        }
        System.out.println("Ar Condicionado: " + this.airconditioner + "\n");

    }
```

2.4 (1.5) Na classe **Laboratory**,

- defina um método **showSensors** que mostra a lista de sensores do laboratório.
- defina um método **removeSensor** que recebe o id do sensor e que remove o mesmo do laboratório.

```
//2.4.1
    public void showSensors(){
        System.out.println("Sensores:");
        for(Sensor s: sensors){
            System.out.println(s);
        }
    }
```

```
//2.4.2
    public void removeSensor (int id){
        Iterator<Sensor> iterator = sensors.iterator();
        while (iterator.hasNext()) {
        Sensor sensor = iterator.next();
        if (sensor.getId() == id) {
             iterator.remove();
        }
    }
```

Grupo 3: (4,0 valores)

Pretende-se agora fazer algumas melhorias e afinações do código anterior.

3.1 (2.0) A primeira melhoria é começar a utilizar exceções que previnam a presença de erros no código. Sendo assim, foi criado o seguinte código em que se testa a criação de um sensor de temperatura e se mostra o resultado produzido no caso dos erros identificados:

```
try {
    TemperatureSensor ts = new TemperatureSensor(0);
} catch (SensorException exc) {
    System.out.println("ERRO: " + exc.getMessage());
    System.out.println("ID: " + exc.getSensorId());
}

try {
    TemperatureSensor ts = new TemperatureSensor(-2);
} catch (SensorException exc) {
    System.out.println("ERRO: " + exc.getMessage());
    System.out.println("ID: " + exc.getSensorId());
}
```

```
// Output produzido


Erro: identificador do sensor nulo
ID: 0



Erro: identificador do sensor negativo
ID: -2
```

- Escreva o código da classe **SensorException** sabendo que se trata de uma exceção verificada.

```
public class SensorException extends Exception {
    int sensorId;

    public SensorException(String string, int sensorId) {
        super(string);
        this.sensorId = sensorId;
    }

    public int getSensorId() {
        return sensorId;
    }
}
```

- Reescreva o construtor da classe **TemperatureSensor** para que seja gerada a exceção anterior nos casos em que o valor do parâmetro **id** é inválido. Veja o *output* produzido pelo código mostrado.

```
Uma vez que o parâmetro id é gerido na classe Sensor a exceção tem de ser gerada por esta classe;
    public Sensor(int id) throws SensorException {
        if (id == 0) {
            throw new SensorException("identificador do sensor nulo", id);
        }
        if (id < 0) {
            throw new SensorException("identificador do sensor negativo", id);
        }
        this.id = id;
    }

Na classe TemperatureSensor:
    public TemperatureSensor(int id) throws SensorException{
        super(id);
    }
```

3.2 (2.0) Por vezes é necessário representar um objeto que possui um número de identificação unívoco que é um valor inteiro sequencial. Neste caso, usa-se normalmente um valor estático que é inicializado a 1 e é incrementado sempre que o seu valor é atribuído a um objeto. A classe **`Sensor`**, por exemplo, poderia usar esta estratégia em vez de receber um valor **`id`** no construtor. Sendo assim, defina o seguinte código:

- Uma classe genérica **`IdentifiedElement`** que permite acrescentar a um objeto de qualquer classe um identificador inteiro unívoco e sequencial como foi explicado. Inclua o construtor e os métodos seletores que achar necessários.

```
public class IdentifiedElement<E> {
    private E element;
    private int id;
    private static int nextId = 1;

    public IdentifiedElement(E element) {
        this.element = element;
        this.id = nextId++;
    }
    public E getElement() {
        return element;
    }
    public int getId() {
        return id;
    }
}
```

- A classe **`WorkStation`** também tem um **`id`** que é recebido no construtor. Considerando que esta classe deixa de receber o **`id`** no construtor, ficando então com um construtor sem argumentos e que deixa de ter o atributo **`id`**, como faria para criar um objeto identificado, usando a classe genérica que definiu. Responda, criando o objeto pedido e escrevendo as instruções que adicionam este objeto a uma coleção **`workstations`**, tal como está definida na classe **`Laboratory`**. Deve criar a coleção pedida.

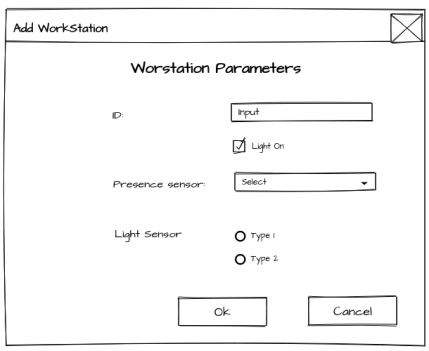
```
workStations = new HashMap();

Workstation workstation = new Workstation();
IdentifiedElement<Workstation> ws = new IdentifiedElement<>(workstation);
workStations.put(ws.getId(), workstation);

// Também se podia ter um HashMap de <Integer,IdentifiedElement<Workstation>>
```

Grupo 4: (4,0 valores)

Pretende-se agora implementar a interface gráfica em JavaFX. Neste sentido foi projetada a janela de diálogo que se mostra na figura abaixo e cuja função é permitir ao utilizador criar uma nova estação de trabalho fornecendo os seus parâmetros.



4.1 (2.0) Tendo em conta o esboço da janela de diálogo defina uma classe **`WorkstationDialog`** derivada de **`Stage`** para representar esta janela. Inclua os atributos e o construtor que recebe uma lista de sensores de presença. O construtor deverá criar todos os controlos mostrados no esboço e posicioná-los de acordo com o que é mostrado. Também deve ser chamado o método que mostra a janela de diálogo tendo em conta as parametrizações habituais para este tipo de janela. Não é necessário incluir as ações dos botões. No caso da **`ComboBox`** utilizada para selecionar um sensor de presença, devem ser visualizados neste controlo os sensores de presença recebidos no construtor.

```
private WorkStation workstation = null;
    private CheckBox lightOn;
    private TextField inputId;
    private RadioButton sensorType1;
    private RadioButton sensorType2;
    private ComboBox<PresenceSensor> cbxSensor;

    public WorkstationDialog(List<PresenceSensor> sensors) {
        setTitle("Add WorkStation");

        // Criar controlos
        Text title = new Text("Worstation Parameters");
        title.setFont(new Font(20));
        Label lblId = new Label("ID:");
        inputId = new TextField("input");
        inputId.setPrefWidth(100);
        lightOn = new CheckBox("Light On");
        lightOn.setSelected(true);
        Label lblSensor = new Label("Presence Sensor");
        ObservableList<PresenceSensor> obsSensors = FXCollections.observableArrayList(sensors);
        cbxSensor = new ComboBox(obsSensors);
        cbxSensor.setPrefWidth(100);
        Label lblLightSensor = new Label("Light Sensor");
        sensorType1 = new RadioButton("Type 1");
        sensorType2 = new RadioButton("Type 2");
        ToggleGroup ligthSensors = new ToggleGroup();
        sensorType1.setToggleGroup(ligthSensors);
        sensorType2.setToggleGroup(ligthSensors);
        Button okButton = new Button("Ok");
        Button cancelButton = new Button("Cancel");

        GridPane dataPane = new GridPane();
        dataPane.setPadding(new Insets(0, 0, 0, 60));
        dataPane.setHgap(20);
        dataPane.setVgap(20);
        dataPane.add(lblId, 0, 0);
        dataPane.add(inputId, 1, 0);
        dataPane.add(lightOn, 1, 1);
        dataPane.add(lblSensor, 0, 2);
        dataPane.add(cbxSensor, 1, 2);
        dataPane.add(lblLightSensor, 0, 3);
        dataPane.add(sensorType1, 1, 3);
        dataPane.add(sensorType2, 1, 4);

        HBox hbxButtons = new HBox(20);
        hbxButtons.setPadding(new Insets(10, 60, 10, 10));
        hbxButtons.setAlignment(Pos.CENTER_RIGHT);
        hbxButtons.getChildren().addAll(okButton, cancelButton);

        VBox root = new VBox(20);
        root.getChildren().addAll(title, dataPane, hbxButtons);
        root.setPadding(new Insets(40, 10, 20, 40));
        root.setAlignment(Pos.CENTER);
        setScene(new Scene(root, 400, 400));

        // Configuração da janela de diálogo
        setResizable(false);
        initStyle(StageStyle.UTILITY);
        initModality(Modality.APPLICATION_MODAL);
        setIconified(false);
        centerOnScreen();
    }
```

4.2 (2.0) Assumindo que foi acrescentado um atributo **`workstation`** do tipo **`Workstation`** à janela criada anteriormente defina o código das ações dos botões de **Ok** e **Cancel**. O botão **Ok** deve criar o objeto **workstation** e preenchê-lo com a informação que está nos controlos. No caso do sensor de luz o tipo determina o id desse sensor. Por exemplo, se estiver selecionado o **`RadioButton`** "**Type 1**" deve ser criado um sensor de luz com o **id** igual a1.

```
// No constructor:
      cancelButton.setOnAction(e -> this.close());
      okButton.setOnAction(e -> createWorkstation());

// Na classe
      private void createWorkstation() {

          int id = 0;
          try {
              id = Integer.parseInt(inputId.getText());
          } catch (NumberFormatException numberFormatException) {
          }
          workstation = new WorkStation(id);

          int lightSensorId = 0;
          if (sensorType1.isSelected()) {
              lightSensorId = 1;
          } else if (sensorType2.isSelected()) {
              lightSensorId = 2;
          }
          LightSensor lightSensor = new LightSensor(lightSensorId, workstation);
          boolean isLightOn = lightOn.isSelected();

          PresenceSensor presenceSensor = cbxSensor.getSelectionModel().getSelectedItem();
          workstation.setLightSensor(lightSensor);
          workstation.setPresenceSensor(presenceSensor);
          if (isLightOn && !workstation.isOn()) {
              workstation.switchOnOff();
          } else if (!isLightOn && workstation.isOn()) {
              workstation.switchOnOff();
          }

          this.close();
      }
```

```
public class Laboratory {
   private String name;
   private double minimumLight;
   private double minTemperature;
   private double maxTemperature;
   private AirConditioner airconditioner;
   private TemperatureSensor temperatureSensor;
   private Set<Sensor> sensors;
   private Map<Integer, WorkStation> workStations;

   public Laboratory(String name){
     // Alínea 2.3
   }

   public void monitor(){
      monitorTemperature();
      monitorStations();
   }

   public void monitorTemperature(){
      if ( temperatureSensor.getTemperature() <=
           minTemperature){
         airconditioner.warmUp();
      } else {
         if( temperatureSensor.getTemperature() >=
             maxTemperature){
            airconditioner.coolDown();
         }
      }
   }

   public void monitorStations(){
      for(WorkStation s: workStations.values()){
         if (!s.isOn() && s.getPresence() &&
           s.lightSensor.getLight() <= minimumLight) {
            s.switchOnOff();
         }
      }
   }

   public void addWorkStation(){/* Alínea 2.3 */ }

   public void setTemperature(double min, double max){
      this.minTemperature = (min >0) ? min : 21;
      this.maxTemperature = (max > min) ? max : 24;
   }

   public void setMaxTemperature(double max){
      this.maxTemperature =
                (max > minTemperature) ? max : 24;
   }

   public void setMinTemperature(double min){
      this.minTemperature =
          (min > 0 && min< maxTemperature) ? min : 21;
   }

   public void setMinLight(double min){
      this.minimumLight =
          (min > 3 && min <= 400) ? min : 150;
   }

   public void switchEverything(){/* Alínea 2.3 */ }

   public void showStatus(){/* Alínea 2.3 */ }

   public void showSensors(){/* Alínea 2.4 */ }

   public void removeSensor (int id){/* Alínea 2.4 */}
}
```

```
public abstract class Sensor {
   private int id;
   public Sensor(int id){
      this.id=id;
   }
   public int getId(){
      return id;
   }
   @Override
   public String toString(){
      return "ID: " + id;
   }
}
```

```
public class TemperatureSensor extends Sensor{
   private static final double RANGE_MIN = 5.0;
   private static final double RANGE_MAX = 34.0;
   public TemperatureSensor(int id){
      super(id);
   }
   public double getTemperature(){
      Random r = new Random();
      double randomValue = RANGE_MIN +
             (RANGE_MAX - RANGE_MIN) * r.nextDouble();
      return randomValue;
   }
}
```

```
public class AirConditioner {
   private boolean isOn;
   private String brand;

   public AirConditioner(String brand){
      this.brand = brand;
      isOn = false;
   }

   public void warmUp(){
      isOn = true;
   }

   public boolean isOn(){
      return this.isOn;
   }

   public String getBrand(){
      return this.brand;
   }

   public void coolDown(){
      isOn = true;
   }

   public void switchOnOff (){
      // Alínea 2.2
   }

   @Override
   public String toString(){
      return "" + ((isOn) ? "Ligado": "Desligado");
   }

  public int hashCode() {
   /*Código omitido*/
  }

  public boolean equals(Object obj) {
   /*Código omitido*/
  }
}
```

```
public class WorkStation {
   private int id;
   LightSensor lightSensor;
   PresenceSensor presenceSensor;
   private boolean isLightOn;

   public WorkStation(int id){
      this.id = id;
      isLightOn = false;
   }

  public void setLightSensor(LightSensor lightSensor){
      this.lightSensor = lightSensor;
   }

   public void setPresenceSensor(
                  PresenceSensor presenceSensor){
      this.presenceSensor = presenceSensor;
   }
   public boolean getPresence(){
      return this.presenceSensor.getPresence();
   }
   public boolean isOn(){
      return this.isLightOn;
   }
   @Override
   public String toString() {
      return "" + id + " - Luz: " +
             ((isLightOn) ? "Ligada" : "Desligada");
   }

   public void switchOnOff (){/* Alínea 2.2 */ }
}
```

Figura 1: Classes `Laboratory`, `WorkStation`, `Sensor`, `AirConditioner` e `TemperatureSensor`

```
public class SmartLab {

   public static void main(String[] args) {

     Laboratory mediaLab = new Laboratory("Media Lab");
     mediaLab.addWorkStation();
     mediaLab.addWorkStation();
     mediaLab.showStatus();
     mediaLab.switchEverything();
     mediaLab.showStatus();
     mediaLab.switchEverything();
     mediaLab.showStatus();
     mediaLab.showSensors();
     mediaLab.removeSensor(4);
     mediaLab.showSensors();
   }
}
```

Output do método main:

```
Luzes:
1 - Luz: Desligada
2 - Luz: Desligada
Ar Condicionado: Desligado

Luzes:
1 - Luz: Ligada
2 - Luz: Ligada
Ar Condicionado: Ligado

Luzes:
1 - Luz: Desligada
2 - Luz: Desligada
Ar Condicionado: Desligado

Sensores:
ID: 2
ID: 4
ID: 5
ID: 1
ID: 3
Sensores:
ID: 2
ID: 5
ID: 1
ID: 3
```

Figura 2: método `main` e respetivo *output*