Exame de Paradigmas de Programação

Exame da Época de Recurso - 13/07/2021

Licenciatura em Engenharia Informática do ISEP

Exame sem consulta; Duração: 60 minutos
Responda no enunciado. Sendo necessário, poderá usar folhas de resposta adicionais.
Nas perguntas de escolha múltipla responda no enunciado usando uma cruz ou ■ para assinalar a ou as respostas corretas. Se necessitar anular uma resposta, escreva "anulada" à esquerda do quadrado. As perguntas de escolha múltipla podem ter várias alternativas corretas, devendo todas elas ser assinaladas. Respostas erradas não descontam.
Cotações: 1, 2, 3, 7, 9, 11, 13, 15, 17: 3,33%; 4, 6, 8, 10, 14, 16: 6,66%; 5, 12, 18: 10%

Nome:_____ Número:_____

1. Considerando o seguinte excerto de código:

```
1  public class C {
2      int x[];
3      int y[][];
4      String s;
5      public C(int[][] a, String s) { }
6      public C(int[] a, int[][] b) { }
7      public C(int [][] a, int [] b) { }
8      public C(int[] b) { }
9  }
10 class Main {
11     public static void main(String[] args) {
12         int vec1[][]={{1,2,3,4},{5,6,7,8}};
13         int vec2[]={1,2,3,4};
14         C obj = new C(vec2, vec1);
15     }
16 }
```

   Indique o número da linha do construtor que é invocado na instanciação do objeto *obj*.

   Resposta:

   linha 6

2. Na programação orientada a objetos, para evitar o acesso direto aos dados aplica-se:

   - ☐ Polimorfismo
   - ☑ Encapsulamento
   - ☐ Classes
   - ☐ Construtores
   - ☐ Abstração

3. Das afirmações seguintes, selecione as verdadeiras:

   - ☑ O polimorfismo, associado à herança, permite que métodos abstratos definidos numa classe abstrata sejam implementados nas subclasses, podendo estes métodos, nessas subclasses, apresentar comportamentos distintos.
   - ☐ Uma subclasse pode ter acesso aos membros de uma superclasse, independentemente do modificador de acesso declarado.
   - ☐ A herança consiste na utilização de classes abstratas que contêm atributos e/ou métodos abstratos.
   - ☑ O polimorfismo permite que objetos de classes que foram definidas sem qualquer relação entre si, ou algo em comum (não usando, por exemplo, *implements* e *extends*), sejam tratadas exatamente da mesma forma.

4. Considerando as seguintes classes:

```
public abstract class Account {
    abstract void deposit(double amt);
    public abstract Boolean withdraw(double amt);
}
public class CheckingAccount extends Account {

}
```

Das opções abaixo, indique qual/quais tornariam o código compilável.

- ☐ Alterar a assinatura da classe *Account* para: *public class Account.*
- ☐ Alterar a assinatura da classe *CheckingAccount* para: *public abstract class CheckingAccount.*
- ☒ Implementar métodos públicos para depósito (*deposit*) e levantamento (*withdraw*) na classe *CheckingAccount.*
- ☐ Alterar assinatura da classe *CheckingAccount* para: *CheckingAccount implements Account.*

5. Considerando as seguintes classes:

```
abstract class Example3 {
    public void print() {
        System.out.print("Superclass ");
    }
}
public class Subclass extends Example3 {
    public void print() {
        System.out.print("Subclass ");
    }
}
public class Subclass2 extends Subclass {
    public void print() {
        System.out.print("Subclass2 ");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Subclass q = new Subclass2();
        Example3 q2 = new Subclass();
        ((Example3) q).print();
        q2.print();
    }
}
```

Qual seria o resultado da execução?

- ☐ Superclass Subclass2
- ☒ Subclass2 Subclass
- ☐ Superclass Subclass
- ☐ Seria lançada uma exceção

6. Considere o seguinte programa:

```
enum T {
    S((float) 20.0, (float) 70.0),
    E((float) 16.0, (float) 50.0),
    X((float) 24.0, (float) 70.0),
    Y((float) 17.0, (float) 50.0);

    private float consumption;
    private float capacity;
    private float range;

    private T(float consumption, float capacity) {
        this.consumption = consumption;
        this.capacity = capacity;
        this.range = capacity / consumption * 100;
    }
    public float getRange() {
        return this.range;
    }
}
```

```
public class Trip {
    private static final float Distance =
        (float) 300.0;

    public static void main(String[] args) {
        for (T t: T.values()) {
            if (t.getRange() > Distance) {
                System.out.print(t + " ");
            }
        }
    }
}
```

Indique qual é a saída do programa.

Resposta: S_E_

Nº Aluno: _____ Nome: _____

7. Qual dos seguintes métodos é declarado pela interface *Comparator*?

   - ☐ *compareWith*
   - ☐ *compareTo*
   - ☒ *compare*
   - ☐ *toCompare*

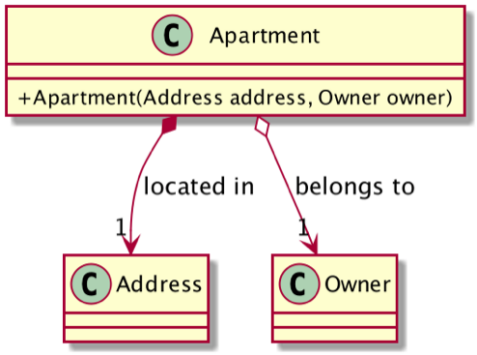
8. Qual/quais das seguintes afirmações sobre soluções que implementam interfaces nativas são verdadeiras?

   - ☒ A interface *Comparator* permite a implementação de uma solução mais flexível, uma vez que possibilita a definição de mais do que uma alternativa de comparação, quando comparada com a interface *Comparable*.
   - ☐ A solução *Comparator* obriga à implementação dessa interface em todas as classes dos objetos a comparar.
   - ☐ A solução *Comparable* permite a definição de vários critérios alternativos de comparação.
   - ☒ A solução *Comparator* não requer a modificação das classes cujos objetos se pretende comparar.

9. Que nome se dá ao relacionamento em que os objetos associados são destruídos quando o objeto que os contém é destruído?

   - ☐ Agregação
   - ☒ Composição
   - ☐ Encapsulamento
   - ☐ Associação

10. Considere o diagrama de classes seguinte.



Codifique os métodos seletores (*getters*) da classe *Apartment*.

```
public Address getAdress(){
this.Adress=Adress;

}

public Owner getOwner(){
this.owner=owner;
}
```

11. Considere o código seguinte.

```
List<String> names = new ArrayList<>();
names.add("Anna");
names.add("Layla");
names.add("Sophie");
names.set(0, "Elena");
names.add(0, "Sarah");
System.out.println(names);
```

Qual será a saída produzida?

- [x] [Sarah, Elena, Layla, Sophie]
- [ ] [Sarah, Elena, Anna, Layla, Sophie]
- [ ] [Sarah, Layla, Sophie]
- [ ] [Elena, Layla, Sophie, Sarah]

12. Considere o seguinte extrato de código.

```
public class Shape implements Printable { /* ... */ }
public class Circle extends Shape { /* ... */ }
public interface Printable { /* ... */ }

public void print(List<? extends Shape> list) {
    for (Printable e : list) {
        System.out.println(e);
    }
}
```

O código compila? Se não compilar, como o poderia corrigir?

Resposta:

O código não compila pois o Printable não é subclasse da classe Shape
Para compilar bastava Pritable e por Circle c ou Shape s na linha 6.

NºAluno: _____ Nome: _____

13. Qual dos seguintes termos não é uma palavra-chave usada no tratamento de exceções em Java:

- ☐ *try*
- ☐ *catch*
- ☒ *fail*
- ☐ *throw*

14. Considere o seguinte excerto de código.

```
class ExampleTryCatch{
   public static void main(String args[]){
      try{
         int arr[]=new int[12];
         arr[24]=24/8;
         System.out.println("Last statement of try block");
      }
      catch(Exception e){
         System.out.println("Some other Exception");
      }
      catch(ArithmeticException e){
         System.out.println("Division by zero");
      }
      catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Out of the array boundaries");
      }
      System.out.println("Out of the try-catch block");
   }
}
```

Qual a saída resultante da execução do código?

- ☐ "Some other Exception"
- ☐ "Out of the array boundaries"
- ☐ "Division by zero"
- ☐ "Out of the try-catch block"
- ☒ Erro de compilação

15. Como impedir que uma variável de instância seja serializada?

- ☒ Declarando a variável como *transient*.
- ☐ Reescrevendo o método *writeObject* na classe a serializar e garantindo que a variável não é escrita.
- ☐ Garantindo que a classe implemente a interface *Serializable*.
- ☐ Evitando que a classe implemente a interface *Serializable*.

16. Preencha o método *saveInfo()* para serializar a informação da classe *ListOfStudents* no ficheiro com o nome *fileName*.

```
public static void saveInfo(String fileName, ListOfStudents info) { /* ... */ }
```

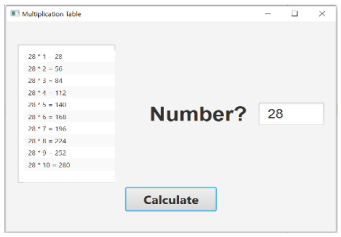
```
public static void saveinfo(String filename, ListOfStudents info){

try{
FileOutputStream out = new FileOutputStream (filename);

ObjectOutputStream oos= new ObjectOutputStream(out);

ListOfStudents list = oos.writeObject(info);

}catch(Exception e){
e.printStackTrace();
}
}
```

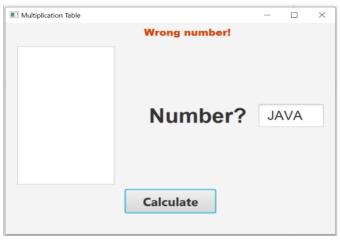
17. Em JavaFX os *Layout Managers* servem para:

- ☐ Permitir que haja sobreposição de janelas e garantir sempre que as janelas são redesenhadas sempre que necessário.
- ☐ Mostrar os componentes visuais da GUI de forma diferente consoante o sistema operativo onde a aplicação está a correr.
- ☒ Colocar e redimensionar os componentes visuais dentro de um contentor.
- ☐ Gerir a visibilidade de componentes dentro de um contentor visual, impedindo a sobreposição.

18. Preencha o método *calculate()* de modo a que apresente o seguinte comportamento:

- Ler um número inteiro de *txtNum* e escrever em *list* a tabuada do inteiro conforme exemplo ilustrado na figura.
- Escrever a mensagem "Wrong number!"em *lblMessage* se a leitura não corresponde a um número válido.






```
public class FXMLController {
    @FXML private Button btn;
    @FXML private TextField txtNum;
    @FXML private ListView list;
    @FXML private Label lblMessage;

    @FXML private void calculate() { /* ... */ }
```

Resposta:

```
public clas s FXMLController {
@FXML private Button btn ;
@FXML private TextField txtNum ;
@FXML private ListView list ;
@FXML private Label lblMessage ;
@FXML private void calculate ( ) {
int number;
try{
number= integer.parseInt(txtNumber.getText());

}catch (Exception e){
lblMessage.setText("wrong number");

for(int i=0;i<11;i++){
int result = number * i;
list.getItems().add(number+ "x"+i+"="+result);
}

}
```