Exame de Paradigmas de Programação

Exame da Época Normal - 28/06/2021

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. Respostas erradas não descontam.

Nome:_____ Número:_____

1. Qual/quais das seguintes afirmações sobre construtores são verdadeiras?

   - ☐ Todos os construtores de uma mesma classe têm que possuir o mesmo modificador de acesso.
   - ☐ Podem possuir um tipo de retorno.
   - ☐ Não podem lançar exceções.
   - ☒ Podem ter um número qualquer de parâmetros.

2. Considerando as seguintes classes

```
class A {
    int i;
    void display() {
        System.out.println(i);
    }
}
class B extends A {
    int j;
    void display() {
        System.out.println(j);
    }
}
public class method_overriding {
    public static void main(String[] args) {
        B obj = new B();

        obj.display();
    }
}
```

   Selecione as instruções a colocar na linha 16 de forma a que a saída seja 7.

   - ☐ obj.i=2; obj.j=5;
   - ☐ obj.i=7; obj.j=5;
   - ☒ obj.i=7; obj.j=7;
   - ☐ obj.i=4; obj.j=3;

3. Das afirmações que se seguem, selecione as verdadeiras:

   - ☐ Denomina-se polimorfismo em Java ao mecanismo de herança múltipla.
   - ☒ O polimorfismo é a propriedade que permite que o tipo real do objecto seja usado para decidir qual a implementação do método a escolher, em vez de o tipo declarado.
   - ☐ O polimorfismo denota o princípio de que o comportamento não depende do tipo real de um objeto.
   - ☐ O polimorfismo consiste na habilidade de uma operação poder ser definida em mais de uma classe e assumir implementações diferentes em cada uma dessas classes.

4. Considerando as seguintes classes:

```
class ClassA {
    public void doing() {
        System.out.println("Doing A");
    }
    public void doing2() {
        System.out.println("Doing2 A");
    }
}
class ClassB extends ClassA {
    public void doing2() {
        System.out.println("Doing2 B");
    }
}
class ClassC extends ClassA {
    public void doing() {
        System.out.println("Doing C");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        List<ClassA> list = new ArrayList<>();
        list.add(new ClassA());
        list.add(new ClassB());
        list.add(new ClassC());
        for (ClassA object : list)
            if (object != null) {
                object.doing();
                object.doing2();
            }
    }
}
```

Indique qual é a saída do programa.

Resposta:

doing a
doing2a
doinga
doing2b
doingc
doing2a

5. Considere que a classe *Undergraduate* é subclasse de *Student*, e que esta por sua vez é subclasse de *Person*. Dada a seguinte declaração de variáveis:

```
Person p = new Person();
Student s = new Student();
Undergraduate ug = new Undergraduate();
```

Qual/quais das seguintes atribuições são possíveis?

- [x] `p = new Undergraduate();`
- [ ] `ug = new Student();`
- [ ] `ug = p;`
- [ ] `s = new Person();`
- [x] `p = ug;`

6. Considere o seguinte código:

```
enum TrafficSignal
{
  RED("STOP"), GREEN("GO"),
  ORANGE("SLOW_DOWN");

  private String action;

  public String getAction() {
    return this.action;
  }

  private TrafficSignal(String action) {
    this.action = action;
  }
```

```
public class codigo3
{
  public static void main(String args[]) {
    TrafficSignal[] signals = TrafficSignal.values();

    for (TrafficSignal signal : signals) {
      System.out.println("Traffic light : " +
        signal.name() +
        " action: " + signal.getAction() );
    }
  }
```

Indique qual é saída do programa.

Resposta:

RED STOP
GREEN GO
SLOW_DOWN ORANGE

NºAluno: _____ Nome: _____

7. Considere o código abaixo:

```
interface Animal { String talk(); }
class Bird implements Animal { public boolean flies() {return true;}   }
class Raven   extends Bird   { public String  talk()  {return "kraa";} }
class Penguin extends Bird   { public boolean flies() {return false;}  }
```

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

Resposta:

na class bird teria de se implementar public String Talk com um string como retorno,

8. Considere o seguinte código:

```
List<X> lst = new ArrayList<>();
lst.add(new Penguin());
```

Quais são os possíveis valores para X para que o código acima compile, usando apenas as entidades introduzidas na pergunta anterior?

Resposta:

animal , bird, penguin

9. Considere as seguintes classes: *Engine* e *Car*.

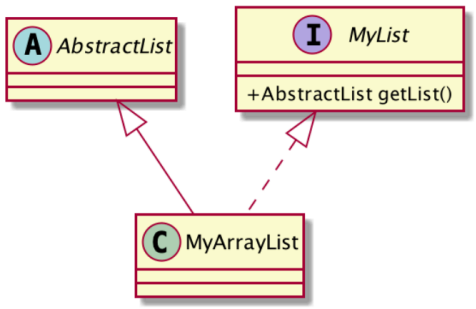
```
public class Engine {
    int power;

    public Engine(int power) {
        this.power = power;
    }

    public String toString() {
        return "␣power␣=␣" + power;
    }
}

public class Car extends Engine {
    String brand;
    String model;

    public Car(String brand, String model, int power) {
        super(power);
        this.brand = brand;
        this.model = model;
    }

    public String toString() {
        return "brand␣=␣" + brand + "␣model␣=␣" + model
               + super.toString();
    }
}
```

Reescreva a classe *Car* de modo a cumprir as melhores práticas da Programação OO.

Resposta:

TROCAR
CLASSE ENGINE EXTENDS CAR

10. Considere o seguinte diagrama de classes:



Quais das opções seguintes representam implementações válidas do método *getList()*, na classe *MyArrayList*?

- [x] `AbstractList getList() {...}`
- [ ] `MyArrayList getList() {...}`
- [ ] `MyList getList() {...}`
- [ ] `Object getList() {...}`

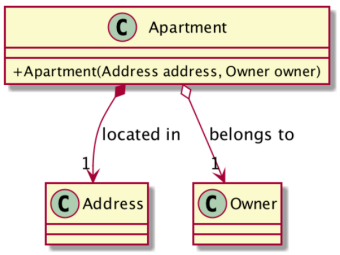
11. Considere o seguinte excerto de uma classe genérica:

```
public class Pair<T, S> {
    private T first;
    private S second;

    public Pair (T first, S second) {
        //...
    }
}
```

Qual/quais das opções seguintes permite a criação de uma instância de *Pair*?

- [ ] `Pair<String, String> p1 = new Pair<>(T, S);`
- [x] `Pair<Integer, String> p2 = new Pair<>(0, "S");`
- [ ] `Pair<T, S> p3 = new Pair<>(T, S);`
- [ ] `Pair<T, S> p4 = new Pair<>("0", "S");`

12. Considere o diagrama de classes seguinte:



Codifique o construtor da classe *Apartment*.

Resposta:

```
public Apartement( Adress address , Owner owner){

this.address= address;
this.owner = owner;
}
```

13. Em Java, as exceções são organizadas hierarquicamente. Quando se pretende capturar múltiplas exceções num bloco de código:

- ☐ Os *catch* devem ser colocados por ordem hierárquica descendente.
- ☒ Os *catch* devem ser colocados por ordem hierárquica ascendente.
- ☐ A ordem em que são colocados os *catch* é irrelevante.
- ☐ Não se pode usar vários *catch*. É necessário usar *throws*.

14. Reescreva o método seguinte para que seja lançada a exceção definida pelo utilizador *MyReadNumberException* quando ocorrer qualquer erro durante a leitura de um número inteiro.

```
private int readNumber() {
    Scanner sc = new Scanner(System.in);
    return sc.nextInt();
}
```

Resposta:

```
private int read Number(){

Scanner sc= new Scanner (System.in);

try{
return sc.nextInt();
}cacth (Exception e){

throw new MyReadNumberException("number format wrong");
}
}
```

15. Considere as classes *Student* e *ListOfStudents*:

```
public class Student {
    int studentId;
    String studentName;
    ...
}
```

```
public class ListOfStudents {
    List<Student> students;
    ...
}
```

Para serializar uma instância de *ListOfStudents*:

- ☐ Apenas a classe *ListOfStudents* deve implementar a interface *Serializable*.
- ☒ Apenas a classe *Student* deve implementar a interface *Serializable*.
- ☐ Nenhuma das classes *ListOfStudents* e *Student* necessita de implementar a interface *Serializable*.
- ☐ Ambas as classes *Student* e *ListOfStudents* devem implementar a interface *Serializable*.

16. Preencha o método *loadInfo()* para desserializar a informação contida no ficheiro *fileName*.

```
public static ListOfStudents loadInfo(String fileName){

}
```

Resposta:

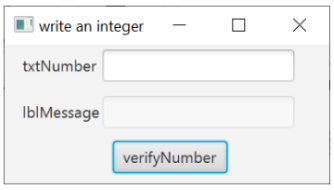
```
public static ListOfStudents loadInfo(String fileName) {

try{

FileInputStream in = new FileInputStream (filename);

ObjectInputStream ois = new ObjectInputStream(in);

ListOfStudents list = ois.readObject();

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

17. Numa aplicação JavaFX, a *tag* *@FXML* aplicada a um atributo de uma classe *controller* é usada para:

    - [x] Permitir que o *loader* (*FXMLLoader*) inicialize o atributo, mesmo que este tenha acesso *private*.
    - [ ] Tornar público (*public*) o acesso ao atributo.
    - [ ] Permitir que o *loader* (*FXMLLoader*) inicialize o atributo, sendo no entanto necessário que o tipo de acesso do atributo seja *protected*.
    - [ ] Permitir que o *loader* (*FXMLLoader*) inicialize o atributo, sendo no entanto necessário que o tipo de acesso do atributo seja *public*.

18. Preencha o método *verifyNumber()* para que leia um número inteiro de *txtNumber* e escreva em *lblMessage* uma das seguintes mensagens:

    - "Negative" se o número lido é < 0.
    - "Positive" se o número lido é $>= 0$.
    - "Wrong number" se a leitura não corresponde a um número válido.




```
public class Controller {
  @FXML
   TextField txtNumber;
  @FXML
   Label lblMessage;
   public void verifyNumber(ActionEvent actionEvent) {

   }
}
```

Resposta:

```
public class Controller {
@FXML
TextFiels txtNumber;
@FXML
Label lblMessage
public void verifyNumber( ActionEvent actionEvent){
int number;
try{
    number = Integer.parseInt(txtNumber.getText());
}
catch (Exception e){
lblMessage.setText("Wrong Number");
}
if(number >= 0){
lblMessage.setText("Positive");
}
else lblMessage.setText("Negative");
}
}
```