Nome:_____ **Nº:**_____ **Curso:**_____ (**A**)

Exame de Programação Orientada aos Objectos **(A)**

MiEI e LCC - DI/UMinho

14/06/2021
Duração: **2h**

Leia o teste com muita atenção antes de começar
Assuma que gets e sets estão disponíveis, salvo se forem explicitamente solicitados.
Na Parte I não existem erros sintácticos propositados.

Parte I - 7.5 valores

1. Considere que lhe pediram para fazer uma aplicação para a gestão do campeonato de formação de hóquei em patins - CHP. O CHP é constituído por clubes, que podem ter várias equipas inscritas nos diversos escalões e cada equipa tem um capitão de equipa e atletas.

   Considerando as definições na Figura 1, qual seria a implementação correcta, numa estratégia de composição de objetos, para o método,

   ```
   public Equipa getEquipa(String idClube, String idEquipa)
          throws ClubeNaoExisteException, EquipaNaoExisteException
   ```

   que enviado a uma instância de CHP devolve a Equipa correspondente (caso esta exista – ver próxima folha):

Figura 1. Gestão de Campeonatos de Hóquei em patins

```
public class Equipa {
    private String id;
    private String escalao;
    private Pessoa capitao;
    private Set<Pessoa> atletas;
...
}
public class Clube {
    private String nome;
    private Map<String,Equipa> equipas;
...
}

public class CHP {
    private Map<String, Clube> clubes;
...
}
```

◯
```
public Equipa getEquipa(String idClube, String idEquipa)
    throws ClubeNaoExisteException, EquipaNaoExisteException {
    for(String c : this.clubes.keySet()){
        if(c.equals(idClube)){
            for(Equipa e: c.getEquipas().values()){
                if (e.getId().equals(idEquipa)){
                    return e;
                }
            }
        }
    }
    throw new EquipaNaoExisteException();
}
```

◯
```
public Equipa getEquipa(String idClube, String idEquipa)
    throws ClubeNaoExisteException, EquipaNaoExisteException{
    return this.clubes.get(idClube).get(idEquipa).clone();
}
```

◯
```
public Equipa getEquipa(String idClube, String idEquipa)
    throws ClubeNaoExisteException, EquipaNaoExisteException{
    return this.clubes.values().stream()
           .filter(eq -> eq.getNome().equals(idClube))
           .findFirst().get()
           .getEquipas().values().stream()
           .filter(e-> e.getId().equals(idEquipa))
           .findFirst().get().clone();
}
```

◯
```
public Equipa getEquipa(String idClube, String idEquipa)
    throws ClubeNaoExisteException, EquipaNaoExisteException{
    Equipa res = null;
    Clube c = this.clubes.get(idClube);
    if (c!= null){
        Map<String,Equipa > equipas = c.getEquipas();
        if (equipas.containsKey(idEquipa)){
            res = equipas.get(idEquipa).clone() ;
        }
    }else{
        throw new ClubeNaoExisteException()
    }
    if (res == null){
        throw new EquipaNaoExisteException();
    }
    return res;
}
```

2. Considere que lhe pediram para fazer uma aplicação para a gestão do campeonato de formação de hóquei em patins - CHP. O CHP é constituída por equipas, que podem ter várias equipas inscritas nos diversos escalões e cada equipa tem um capitão de equipa e atletas.

   Considerendo as definições na Figura 1, qual seria a implementação correcta, numa estratégia de composição de objetos, para o método

```
List<Equipa> getEquipas(String idClube, String escalao)
    throws ClubeNaoExisteException
```

que enviado a uma instância de CHP devolve a Lista de Equipas de um clube que são de um determinado escalão

◯
```
public List<Equipa> getEquipas(String idClube, String escalao) throws ClubeNaoExisteException{
    if (this.clubes.containsKey(idClube)){
        return this.clubes.get(idClube).getEquipas()
                .values().stream()
                .filter( e->e.getEscalao().equals(escalao))
                .collect(Collectors.toList());
    }else{
        throw new ClubeNaoExisteException();
    }
    return new ArrayList();
}
```

◯
```
public List<Equipa> getEquipas(String idClube, String escalao) throws ClubeNaoExisteException {
    List<Equipa> res = new List<>();
    for(Equipa e: this.equipas.values()){
        if (e.getNome().equals(idEquipa)){
            for (Equipa b : e.getEquipas().values()){
                if(b.getEscalao().equals(escalao)){
                    res.add(b.clone());
                }
            }
        }else{
            throw new ClubeNaoExisteException();
        }
    }
    return res;
}
```

◯
```
public List<Equipa> getEquipas(String idClube, String escalao) throws ClubeNaoExisteException {
    List<Equipa> res = new ArrayList<>();
    if (this.clubes.containsKey(idClube)){
        for(Equipa e: this.clubes.get(idClube).getEquipas().values()){
            if(e.getEscalao().equals(escalao)){
                res.add(e.clone());
            }
        }
    }else{
        throw new ClubeNaoExisteException();
    }
    return res;
}
```

◯
```
public List<Equipa> getEquipas(String idClube, String escalao) throws ClubeNaoExisteException {
    List<Equipa> res = new ArrayList<>();
    for( Map.Entry<String, Clube> c : this.clubes.entrySet())
        if(c.equals(idClube)){
            for(Equipa e: c.getValue().getEquipas().values()){
                if(e.getEscalao().equals(escalao)){
                    res.add(e);
                }
            }
        }
    return res;
}
```

3. Considere as seguintes definições:

```
public interface Empregado {
    public String getEmpregador();
}

public class Aluno {
    ...
    public Aluno() { ... }
    public boolean epocaEspecial() { return false; }
}

public class AlunoTE extends Aluno implements Empregado {
    ...
    public AlunoTE() { ... }
    public boolean epocaEspecial() { return true; }
    public String getEmpregador() { return "Externo"; }
}

public class Funcionario implements Empregado {
    ...
    public Funcionario() { ... }
    public String getEmpregador() { return "UMinho"; }
}
```

Considere ainda que estão disponíveis as seguintes definições:

```
public List<Boolean> getEEstatus1(List<Empregado> l) {
    return l.stream().filter(e -> e instanceof Aluno).map(a -> a.epocaEspecial())
                     .collect(Collectors.toList());
}

public List<Boolean> getEEstatus2(List<Aluno> l) {
    return l.stream().filter(a -> a instanceof Empregado).map(e -> e.epocaEspecial())
                     .collect(Collectors.toList());
}

public List<Boolean> getEEstatus3(List<Empregado> l) {
    return l.stream().map(e -> (Aluno) e).map(a -> a.epocaEspecial())
                     .collect(Collectors.toList());
}
```

Sabendo que irão ser utilizadas as seguintes listas:

```
List<Empregado> lemp = new ArrayList<>();
lemp.add(new Funcionario());
lemp.add(new AlunoTE());
lemp.add(new Funcionario());
lemp.add(new AlunoTE());
lemp.add(new Funcionario());

List<Aluno> lal = new ArrayList<>();
lal.add(new AlunoTE());
lal.add(new Aluno());
lal.add(new AlunoTE());
lal.add(new Aluno());
```

para cada afirmação assinale, **caso exista**, a opção que a torna verdadeira (se nenhuma opção for válida, não assinale nada):

a) A expressão ◯ `getEEstatus1(lemp);` | ◯ `getEEstatus2(lal);` | ◯ `getEEstatus3(lemp);` gera um erro de compilação.

b) A expressão ◯ `getEEstatus1(lemp);` | ◯ `getEEstatus2(lal);` | ◯ `getEEstatus3(lemp);` gera a lista `[true,true]`.

c) A expressão ◯ `getEEstatus1(lemp);` | ◯ `getEEstatus2(lal);` | ◯ `getEEstatus3(lemp);` gera um erro de execução.

d) A expressão ◯ `getEEstatus1(lemp);` | ◯ `getEEstatus2(lal);` | ◯ `getEEstatus3(lemp);` gera a lista `[true,false,true,false]`.

4. Considere o seguinte tipo de dados para representar as turmas de um curso. Cada turma, indexada pelo seu nome, possui um conjunto de alunos, indexados pelo seu número:

```
private Map<String, Map<Integer, Aluno>> turmas;
```

Considere o seguinte método, que irá indicar a turma com maior média de notas, considerando apenas os alunos com nota média maior do que 10. Caso várias turmas tenham a mesma média, deve-se selecionar a que tiver o maior número de alunos (independentemente da nota). Assuma que o método `getMedia` da classe `Aluno` existe e calcula a média de um aluno.

```
public String melhorTurma() {
    Comparator<Map.Entry<String, Map<Integer, Aluno>>> comp = (a, b) -> {
        double va = a.getValue().values().stream().filter(al -> al.getMedia() > 10.0)
                .mapToDouble(Aluno::getMedia).average().orElse(0.0);
        double vb = b.getValue().values().stream().filter(al -> al.getMedia() > 10.0)
                .mapToDouble(Aluno::getMedia).average().orElse(0.0);
        int sizea = a.getValue().size();
        int sizeb = b.getValue().size();
        if (va==vb) return sizeb - sizea;
        else return (int) (vb - va);
    };
    return turmas.entrySet().stream().sorted(comp)
                .map(e -> e.getKey()).findFirst().orElse("N/A");
}
```

Selecione a alínea correta:

◯ O método está corretamente implementado, mas na última linha usa-se um `orElse` desnecessário; bastava terminar a linha com `findFirst`.

◯ O método está corretamente implementado, mas as turmas sem alunos são consideradas com média 0.0.

◯ O método está corretamente implementado, porém numa estratégia de composição seria necessário acrescentar invocações apropriadas ao método `clone`.

◯ O método está incorretamente implementado, pois não se podem construir streams sobre o resultado do método `entrySet`.

◯ O método está incorretamente implementado pois, como se pretende ter uma ordenação com dois critérios, deve-se usar dois `Comparators`.

◯ O método está incorretamente implementado visto que é necessário usar o método `compareTo` para produzir o resultado do `Comparator`.

5. Considere o código da Figura 1. Considere ainda que os métodos `Set<Pessoa> getAtletas()` e `setAtletas(Set<Pessoa> s)`, da classe `Equipa`, foram implementados do seguinte modo:

```
public Set<Pessoa> getAtletas() {
    return atletas.clone().stream().collect(Collectors.toSet());
}

public void setAtletas(Set<Pessoa> s) {
    atletas = s.stream().map(Pessoa::clone).collect(Collectors.toSet());
}
```

Assinale a afirmação verdadeira:

- ◯ Os métodos estão correctamente implementados e respeitam o encapsulamento, se a relação entre `Equipa` e `Pessoa` for de agregação.
- ◯ Os métodos estão correctamente implementados e respeitam o encapsulamento, se a relação entre `Equipa` e `Pessoa` for de composição.
- ◯ Os métodos não estão estão correctamente implementados, não sendo consistentes no tratamento do encapsulamento.
- ◯ Não é possível dizer, apenas a partir da sua implementação, se estes métodos estão, ou não, correctamente implementados, no que respeita à noção de encapsulamento.

Parte II - 12.5 valores

Considere que se pretende ter um sistema que implemente uma serviço de disponibilização de podcasts. Um podcast possui um identificador (um nome) e tem associada uma lista de episódios que foram disponibilizados.

A entidade episódio de um podcast foi definida da seguinte forma:

```
public class Episodio {
    private String nome;
    private double duracao;
    private int classificacao; //dada pelos seus ouvintes (valor de 0..100)
    private List<String> conteudo; //corresponde ao texto que e' dito
                                   //quando se reproduz o episodio
    private int numeroVezesTocada; //qts vezes e' que o podcast foi ouvido
    private LocalDateTime ultimaVez; //regista quando foi reproduzido
                                     //ultima vez

   ...
   ...
}
```

Considere também que o sistema completo a desenvolver `SpotifyPOO` guarda, além dos podcasts existentes e dos episódios destes, informação relativa aos utilizadores do sistema. Para cada utilizador guarda-se o seu identificador (que neste sistema é a String do seu email), o seu nome e a informação dos podcasts que tem subscritos.

Resolva os seguintes exercícios:

Nome:_____ **Nº:**_____ **Curso:**_____ (**A**)

6. Efectue a declaração das classes `Podcast`, `Utilizador` e `SpotifyPOO`, identificando apenas as variáveis existentes e codificando o método `public List<Episodio> getEpisodios(String nomePodcast)`, da classe `SpotifyPOO`, que dado um identificador de podcast devolve, numa lógica de composição, uma lista com os episódios disponíveis para esse podcast.

7. Desenhe o Diagrama de Classes da solução `SpotifyPOO`. Considere que não necessita de colocar os métodos `get` e `set`.

   Resposta:

8. Codifique o método `public void remove(String nomeP) throws...`, da classe `SpotifyPOO`, que remove do sistema o podcast identificado. Esta remoção não poderá ser possível se o podcast não existir registado no sistema ou se o mesmo podcast tiver utilizadores que actualmente o estejam a subscrever. Indique na assinatura do método as excepções de que necessitar (não necessita de as codificar).

   Resposta:

9. Codifique o método `public Episodio getEpisodioMaisLongo(String u)`, da classe `SpotifyPOO`, que para o utilizador passado por parâmetro, devolve o episódio mais longo de entre os podcasts que esse utilizador tem subscritos.

10. Desenvolva o método `public Map<Integer,List<Episodio>> episodiosPorClassf()`, da classe `SpotifyPOO`, que associa a cada valor de classificação a lista dos episódios, de todos os podcasts, com essa mesma classificação.

11. Considere agora que a classe `Episodio` deverá implementar a interface `Playable`, definida como

```
public interface Playable {
  public void play();
}
```

Tendo em consideração que existirá um objecto chamado `System.media`, que tem o mesmo comportamento do `System.out` e que transforma em som o conteúdo em texto do episódio, altere a classe `Episodio` de modo a que implemente `Playable`.

12. Considere agora que se criaram novos tipos de conteúdo que passam pela disponibilização de episódios com som e vídeo. Pretende criar-se o `EpisodioVideo`, que para além do audio também possui uma lista de `Byte` que representa o conteúdo visual. Codifique a classe `EpisodioVideo`, apresentando a sua declaração e variáveis, o construtor parametrizado e a codificação do método `play`. Por simplificação assuma que, para reproduzir estes conteúdos, pode primeiro tratar do vídeo e depois do som, e que o `System.media` também sabe reproduzir vídeo.

13. Considere que é possível efectuar a reprodução de um podcast por parte de um Utilizador, através do método `public void playEpisodio(String idPodCast, String nomeEpisodio) throws AlreadyPlayingException` da classe `Utilizador`. A excepção é lançada quando esse utilizador já está no momento a reproduzir um episódio. Considere que se pretende criar agora a noção de `UtilizadorPremium`, que é um utilizador que, enquanto reproduz um episódio, possui a capacidade de colocar os outros episódos que pretende reproduzir numa lista de espera.

    Codifique a classe `UtilizadorPremium` com as suas variáveis de instância e a implementação do método `playEpisodio`.

14. Codifique o método `public void gravaInfoEpisodiosParaTocarMaisTarde(String fich)`, que grava em ficheiro de texto os episódios dos `UtilizadorPremium` que estão na fila de espera para serem reproduzidos. A informação deve ficar guardada com o formato

```
Nome Utilizador
Id do Episodio - duracao
Id do Episodio - duracao
...
...
Nome Utilizador
Id do Episodio - duracao
...
```

Tenha em atenção as possíveis excepções resultantes do uso de ficheiros.