Programação Orientada por Objetos

Exame Época Recurso, 24 de julho de 2017 – 18:00

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: (3 Valores)

1.1 Na herança, a classe derivada vai herdar da classe base:

a) Os métodos, o construtor e os atributos.
b) Somente os métodos.
c) Somente os atributos
d) **Os métodos e os atributos.**

1.2 A classe **`Elefante`** é derivada da classe **`Animal`**, que é abstrata. Se a classe **`Elefante`** não tiver chamada ao construtor da classe **`Animal`**, o que sucede?

a) Provoca erro na compilação.
b) Provoca erro na execução.
c) A superclasse não é compilada, só a subclasse é compilada.
d) **O compilador inclui automaticamente a chamada `super()`.**

1.3 Na programação de uma aplicação foi chamado o método **`falar`** para um conjunto de animais de várias classes pertencentes a diferentes hierarquias de classes, mas que implementam todas elas a interface **`Comunicavel`**. Qual o conceito da Programação Orientada por Objetos a que se está a recorrer?

a) Herança.
b) Composição.
c) **Polimorfismo.**
d) Abstração.

1.4 Qual a melhor forma de remover um objeto de uma coleção?

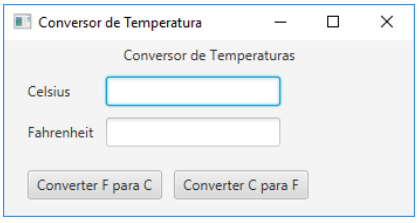
a) Usando um ciclo `for`.
b) Usando um ciclo `for-each`.
c) **Usando um objeto `iterator`.**
d) Usando um objeto `element`.

1.5 Em JavaFX, os objetos gráficos da UI são adicionados a um objeto do tipo:

a) **`Stage`**
b) **`Scene`**
c) **`Application`**
d) **`Pane`**

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

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



Grupo 2: (6 valores)

Pretende-se desenvolver uma aplicação para planos de treinamento de corrida (*run*) e ginásio (*workout*). Para o protótipo inicial foram criadas as classes **Activity**, **Workout** e **Run**, os tipos enumerados **WorkoutType**, **RunType e WeekDay** e a interface **Planner** da **Figura 1**. Na **Figura 2** encontra-se algum código de teste da aplicação 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 (1.0) Comece por definir os métodos **toString()** dos tipos enumerados **RunType** e **WorkoutType**.

```
// RunType
@Override
public String toString(){
    String s = "";
    switch (this){
        case LONGRUN: s = "Corrida longa";
            break;
        case EASYRUN: s = "Corrida leve";
            break;
        case HILLRUN: s = "Corrida em elevação";
            break;
        case TEMPORUN: s = "Corrida cronometrada";
            break;
    }
    return s;
}
```

```
// WorkoutType
@Override
public String toString(){
    String s = "";
switch (this){
        case ENDURANCE: s = "Resiliência";
            break;
        case STRENGTH: s = "Fortalecimento";
            break;
        case BALANCE: s = "Equilíbrio";
            break;
        case FLEXIBILITY: s = "Alongamentos";
            break;
    }
    return s;
}
```

2.2 (1.0) A classe **Activity**, mostrada na Figura 1, é abstrata e tem os atributos e métodos comuns às suas classes derivadas. Sendo assim, complete a classe fornecendo o código dos seguintes métodos:

- **Construtor** – recebe uma String com a descrição da atividade (pode ser "Corrida" ou "Exercício em ginásio", conforme a classe derivada), o dia da semana e a duração da atividade.
- **getters** para o tipo de atividade e o dia da semana.
- **String toString()** – devolve a informação sobre esta atividade. Tenha em atenção o que é mostrado no *output* da aplicação. Exemplo:

```
Domingo: Corrida
        Duração: 30 min.
```

```
public Activity(String description, WeekDay day, int duration) {
    this.description = description;
    this.day = day;
    this.duration = duration;
}

public String getDescription(){
    return description;
}
```

```
public WeekDay getDay(){
    return day;
}

@Override
public String toString(){
    return this.day + ": " + description +
        " - Duração: " + this.duration + " min.";
}
```

2.3 (2.0) Considere agora as classes **Workout** e **Run** mostradas na **Figura 1**. Implemente os seus construtores, os seus métodos **toString()** e qualquer método **setter** ou **getter** necessário. Tenha em conta o que é mostrado no *output* da aplicação.

```
// Classe Workout
public Workout(WeekDay day, int duration,
               WorkoutType workoutType) {
    super("Exercício em ginásio", day,
duration);
    this.workoutType = workoutType;
}

@Override
public String toString() {
    return super.toString() + " - "
                            + workoutType +
"\n";
```

```
// Classe Run
public Run(WeekDay day, int duration, RunType runType, int distance){
    super("Corrida", day, duration);
    this.runType = runType;
    this.distance = distance;
}
public int getDistance(){
    return distance;
}
@Override
public String toString(){
    return super.toString() + " - " + runType + " - Distância: " + distance + " km";
}
```

```
    }
```

2.4 (2.0) Defina a classe **`TrainingPlan`**. Esta classe implementa a interface **`Planner`**. Use um **conjunto ordenado** para guardar as atividades. Inclua o construtor e os métodos definidos pela interface e os que forem necessários tendo em conta o *output* que é produzido na **Figura 2**.

```
public class TrainingPlan implements Planner {
    LinkedHashSet<Activity> activities;

    public TrainingPlan(List activities){
        this.activities = new LinkedHashSet<>(activities);
    }
    public List<Activity> getActivities(){
        return new ArrayList<Activity>(activities);
    }
    public List<WeekDay> GetActivitiesDays(){
        ArrayList<WeekDay> days = new ArrayList<>();
        for (Activity a: activities){
            days.add(a.getDay());
        }
        return days;
    }
    @Override
    public String toString(){
        String s = "Plano semanal:\n";
        for (Activity a: activities){
            s += a.toString() + "\n";
        }
        return s;
    }
}
```

Grupo 3: (6 Valores)

Continuando a aplicação anterior considere agora a classe **`Athlete`** da **Figura 3**. A classe **`Athlete`** representa um atleta ao qual irá ser atribuído um plano semanal de exercícios. Este plano é representado por um objeto da classe **`TrainingPlan`** (Veja a questão 2.4 e assuma que a classe foi feita). O método **`setTrainingPlan()`** da classe **`Athlete`** será então usado para fazer a atribuição desse plano ao atleta. Antes da criação deste método foi decidido definir uma classe de exceção não verificada chamada **`PlanException`** que servirá para identificar os possíveis erros na atribuição do plano de exercícios. Sendo assim:

3.1 (1.0) – Crie a classe **`PlanException`** – é uma classe de exceção não verificada que recebe no construtor uma variável do tipo enumerado **`PlanExceptionType`** (ver Figura 4) com o tipo de exceção que ocorreu. Este valor é guardado num atributo e pode ser obtido posteriormente através de um método seletor.

```
public class PlanException extends RuntimeException {
    private PlanExceptionType planExceptionType;

    public PlanException(PlanExceptionType exceptionType) {
        super(exceptionType.toString());
        this.planExceptionType = exceptionType;
    }

    public PlanExceptionType getPlanExceptionType() {
        return planExceptionType;
    }
}
```

3.2 (2.0) – Crie agora o método **`setTrainingPlan()`** da classe **`Athlete`**. Este método recebe um plano de treino e guarda-o no atributo **`trainingPlan`**. Antes disso verifica se o plano recebido não é **`null`**, caso contrário lança a exceção **`PlanException`**, criada na alínea anterior, com o tipo de erro que ocorreu. Verifica também se a lista de atividades do plano recebido existe e se contém atividades, lançando exceções **`PlanException`** se não se verificarem essas situações. Depois de recebido o plano, é necessário ter um mapa com todas as atividades que devem ser feitas, para isso, dentro do mesmo método, deve passar as atividades do plano de exercícios para o atributo **`plan`**. Cada entrada na coleção **`plan`** vai ter a atividade a desenvolver e se já foi efetuada ou não.

```
public void setTrainingPlan(TrainingPlan trainingPlan) {
    if (trainingPlan == null) {
        throw new PlanException(PlanExceptionType.MISSING_PLAN);
    }
    if (trainingPlan.getActivities() == null) {
        throw new PlanException(PlanExceptionType.INVALID_ACTIVITIES);
    }
    if (trainingPlan.getActivities().isEmpty()) {
        throw new PlanException(PlanExceptionType.NO_ACTIVITIES);
    }
    this.trainingPlan = trainingPlan;

    plan = new HashMap<>();
    for (Activity activity : trainingPlan.getActivities()) {
        plan.put(activity, false);
    }
}
```

3.3 (0.5) – Defina agora o método **`setActivityDone`** da classe **`Athlete`**. Este método recebe uma atividade e, se existir, no atributo **`plan`**, marca-a como feita (valor **`true`**).

```
public void setActivityDone(Activity activity) {
    if (plan.containsKey(activity)) {
        plan.put(activity, true);
    }
}
```

3.4 (1.0) – Reutilizando o código do método **`main`** mostrado da **Figura 4**, exemplifique a captura da exceção **`PlanException`** criada anteriormente. Deve ser mostrado no ecrã o tipo de erro que ocorreu, obtendo a informação a partir do objeto de exceção recebido.

```
try {
    athlete.setTrainingPlan(null);
} catch (PlanException e) {
    System.out.println("Erro: " + e.getPlanExceptionType());
}
```

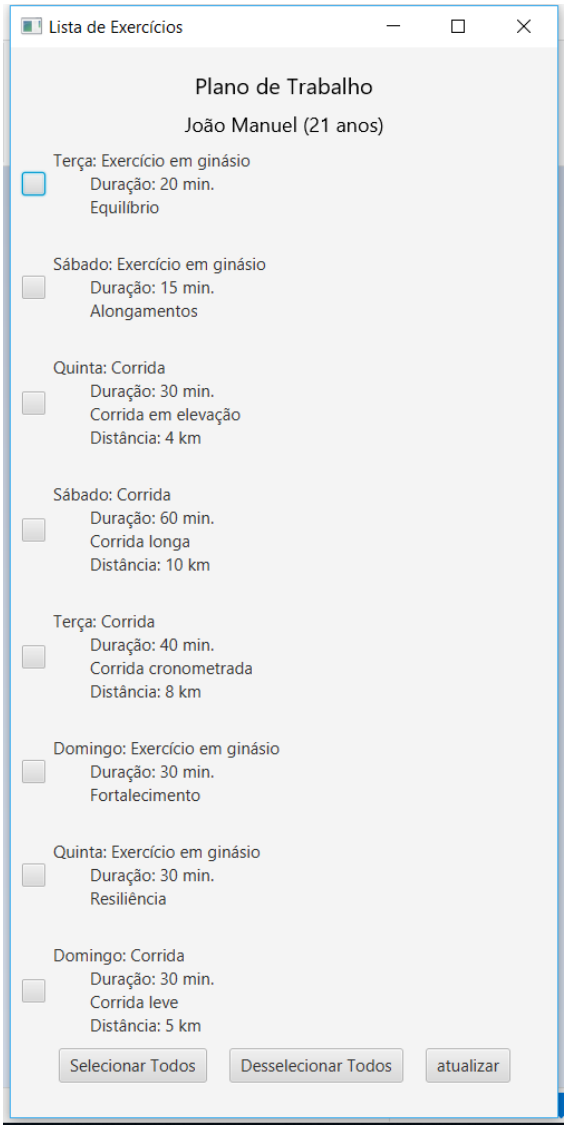
3.5 (1.5) – Defina por fim o método **`runDistance`** da classe **`Athlete`**. Este método devolve o número de quilómetros percorridos pelo atleta de acordo com o seu plano de atividade guardado no atributo **`plan`**. Como é lógico apenas entram neste cálculo as atividades de corrida (*Run*) que foram feitas.

```
public int runDistance() {
    if (plan == null) {
        return 0;
    }
    int distance = 0;
    for (Activity activity : plan.keySet()) {
        if (!plan.get(activity)) {
            continue;
        }
        if (activity instanceof Run) {
            distance += ((Run) activity).getDistance();
        }
    }

    return distance;
}
```

Grupo 4: (5 Valores)

Pretende-se implementar uma interface gráfica em JavaFX, para o programa anterior, com um layout semelhante ao da figura mostrada ao lado. Neste caso, a janela da aplicação é preenchida por um objeto da classe **`AthletePlanPane`** cujo código aparece na Figura 5. Neste grupo apenas se irá definir o código do construtor.



4.1 (1.0) – Comece por criar os elementos gráficos correspondentes ao título, informação do atleta e botões. Este código irá ser colocado no construtor a seguir ao código mostrado na Figura 5.

```
Text title = new Text("Plano de Trabalho");
title.setFont(Font.font("SanSerif", 20));
title.setFill(Color.BLACK);

String info = athlete.getName() + " (" + athlete.getAge() + " anos)";
Text person = new Text(info);
person.setFont(Font.font("SanSerif", 18));
person.setFill(Color.BLACK);

Button buttonAll = new Button("Selecionar Todos");
Button buttonUpdate = new Button("Desselecionar Todos");
Button buttonNone = new Button("atualizar");
```

4.2 (1.0) – Agora pretende-se colocar, em painéis, os elementos criados de forma a recriar a disposição mostrada na figura da direita. Neste caso assuma que a lista de atividades que aparece no centro é constituída por um painel do tipo **`VBox`** com o nome **`activitiesVBox`**, que deve criar, mas sem preencher ainda com as atividades. Escolha também a classe base de **`AthletePlanPane`**.

```
-----  class AthletePlanPane extends VBox -----

getChildren().addAll(title, person);

VBox activitiesVBox = new VBox();
HBox buttonsHBox = new HBox();
buttonsHBox.setSpacing(20);
buttonsHBox.setAlignment(Pos.CENTER);

buttonsHBox.getChildren().addAll(buttonAll, buttonUpdate, buttonNone);
getChildren().add(buttonsHBox);
```

4.3 (1.5) – Tal como foi referido na alínea anterior, a lista de atividades que aparece no centro é constituída por um painel do tipo **`VBox`** com o nome **`activitiesVBox`**. Sabendo que esta **`VBox`** é preenchida por uma lista de **`CheckBox`** que mostra a atividade, indicando se a mesma já foi efetuada ou não, crie o código necessário à criação desta lista a partir do plano de atividades do atleta guardado no atributo **`plan`**, acrescentando-a à **`VBox.`**

```
for (Activity activity : plan.keySet()) {
        CheckBox activityBox = new CheckBox(activity.toString());
        activityBox.setSelected(plan.get(activity));
        activitiesVBox.getChildren().add(activityBox);
    }
getChildren().add(activitiesVBox);
setPadding(insets);
```

4.4 (1.5) – O botão "Selecionar Todos", quando premido, permite marcar como "feita" todas as atividades que são mostradas na figura. Escreva o código da ação deste botão, indicando qualquer alteração que seja necessário fazer à classe **`AthletePlanPane`**.

```
class AthletePlanPane extends VBox {

    // …

    private List<CheckBox>  activitiesBoxes;

    public AthletePlanPane(Athlete athlete) {

        // …

        activitiesBoxes = new ArrayList<>();
        for (Activity activity : plan.keySet()) {

            // ….
            activitiesBoxes.add(activityBox);
        }

        buttonAll.setOnAction(e -> selectAll());
    }

    private void selectAll() {
        for(CheckBox cb : activitiesBoxes)
            cb.setSelected(true);
    }
}
```

```java
public enum RunType {
    LONGRUN, EASYRUN, HILLRUN, TEMPORUN;

    // Alínea 2.1
}
```

```java
public enum WorkoutType {
    ENDURANCE, STRENGTH, BALANCE, FLEXIBILITY;

    // Alínea 2.1
}
```

```java
public enum WeekDay {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY;

    @Override
    public String toString(){
        // Código omitido
    }
}
```

```java
public interface Planner {
    public List<Activity> getActivities();
    public List<WeekDay> getActivitiesDays();
}
```

```java
public abstract class Activity {
    private String description;
    private WeekDay day;
    private int duration; // minutes

    // métodos equals e hashcode já implementados-
    // - código omitido


    // Alínea 2.2
}
```

```java
public class Run extends Activity {
    private RunType runType;
    private int distance;

    // Alínea 2.3
}
```

```java
public class Workout extends Activity {

    private WorkoutType workoutType;

    // Alínea 2.3
}
```

Figura 1: Classes **`Activity`**, **`Run e Workout`**, tipos enumerados **`RunType`**, **`WorkoutType e WeekDay`** e interface **`Planner`**

```java
public class RunnerApp {

    public static void main(String[] args) {
        Activity a;
        ArrayList<Activity> activities = new ArrayList<>();
        a = new Run(WeekDay.SUNDAY, 30, RunType.EASYRUN, 5);
        activities.add(a);
        a = new Workout(WeekDay.SUNDAY, 30,
                        WorkoutType.STRENGTH);
        activities.add(a);
        a = new Run(WeekDay.TUESDAY, 40, RunType.TEMPORUN, 8);
        activities.add(a);
        a = new Workout(WeekDay.TUESDAY, 20,
                        WorkoutType.BALANCE);
        activities.add(a);
        a = new Run(WeekDay.THURSDAY, 30, RunType.HILLRUN, 4);
        activities.add(a);
        a = new Workout(WeekDay.THURSDAY, 30,
                        WorkoutType.ENDURANCE);
        activities.add(a);
        a = new Run(WeekDay.SATURDAY, 60, RunType.LONGRUN, 10);
        activities.add(a);
        a = new Workout(WeekDay.SATURDAY, 15,
                        WorkoutType.FLEXIBILITY);
        activities.add(a);
        TrainingPlan tp = new TrainingPlan(activities);
        System.out.println(tp);

        // Mais Código na Figura 4
    }
}
```

Output do método main:

```
Plano semanal:
Domingo: Corrida
        Duração: 30 min.
        Corrida leve
        Distância: 5 km
Domingo: Exercício em ginásio
        Duração: 30 min.
        Fortalecimento
Terça: Corrida
        Duração: 40 min.
        Corrida cronometrada
        Distância: 8 km
Terça: Exercício em ginásio
        Duração: 20 min.
        Equilíbrio
Quinta: Corrida
        Duração: 30 min.
        Corrida em elevação
        Distância: 4 km
Quinta: Exercício em ginásio
        Duração: 30 min.
        Resiliência
Sábado: Corrida
        Duração: 60 min.
        Corrida longa
        Distância: 10 km
Sábado: Exercício em ginásio
        Duração: 15 min.
        Alongamentos
```

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

```
public class Athlete {

    private String name;
    private int age;
    private TrainingPlan trainingPlan;
    private Map<Activity, Boolean> plan;

    public Athlete(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {  return name; }
    public int getAge() { return age; }
    public Map<Activity, Boolean> getPlan() { return plan; }

    public void setTrainingPlan(TrainingPlan trainingPlan) {
        // Alínea 3.2
    }

    public void setActivityDone(Activity activity) {
        // Alínea 3.3
    }

    public int runDistance() {
        // Alínea 3.5
    }

}
```

```
public enum PlanExceptionType {
    MISSING_PLAN,
    INVALID_ACTIVITIES,
    NO_ACTIVITIES,;

    @Override
    public String toString() {
        // Código omitido
    }
}
```

Figura 3: Classe `Athlete` e tipo enumerado `PlanExceptionType`

```
public class RunnerApp {
    public static void main(String[] args) {

        // Código omitido – está na Figura 2

        TrainingPlan tp = new TrainingPlan(activities);
        System.out.println(tp);

        Athlete athlete = new Athlete("João manuel", 21);
        athlete.setTrainingPlan(tp);
        for (Activity activity : tp.getActivities()) {
            athlete.setActivityDone(activity);
        }
        System.out.println("Minutos de corrida: " + athlete.runDistance());
    }
}
```

Output do método main:

```
Minutos de corrida: 27
```

Figura 4: Continuação do método `main` e respetivo *output*

```
class AthletePlanPane extends ???????? {

    Map<Activity, Boolean> plan;

    public AthletePlanPane(Athlete athlete) {
        if (athlete == null) {
            return;
        }

        plan = athlete.getPlan();
        // Código omitido
     }
}
```

Figura 5: Classe AthletePlanPane