

Programação Orientada pelos Objectos

Exame de Recurso (Duração 3h)

LEI 2015/2016

Instruções:

- Antes de começar a resolver, **leia o enunciado do princípio até ao fim.**
 - A interface e classe do grupo III tem mais métodos do que os que deverá implementar na resolução deste teste. Tenha o cuidado de **ver com muita atenção quais os métodos que deve implementar**, para **não desperdiçar o seu tempo a implementar métodos que não lhe são pedidos.**
 - Disponibilizamos a descrição sumária de todos os métodos, incluindo os que não tem de implementar, para que os possa usar na sua resolução.
 - **Pode** usar caneta ou lápis.
 - Não é permitido consultar quaisquer elementos para além deste enunciado.
-

Introdução aos problemas para os grupos I e II:

Nestes 2 grupos terá que implementar algumas das classes necessárias à construção de programas para a gestão dos cartões de transporte de uma empresa de transporte de autocarros. No primeiro grupo, é pedida a implementação completa de algumas classes (Fig. 1). No segundo grupo, é pedida a implementação parcial de uma classe que faz a gestão da colecção dos cartões de transporte (Fig. 2).

Notas:

- Nos diagramas os argumentos dos métodos seguem a ordem da descrição textual.
- Pode considerar que as classes de excepção já estão implementadas.

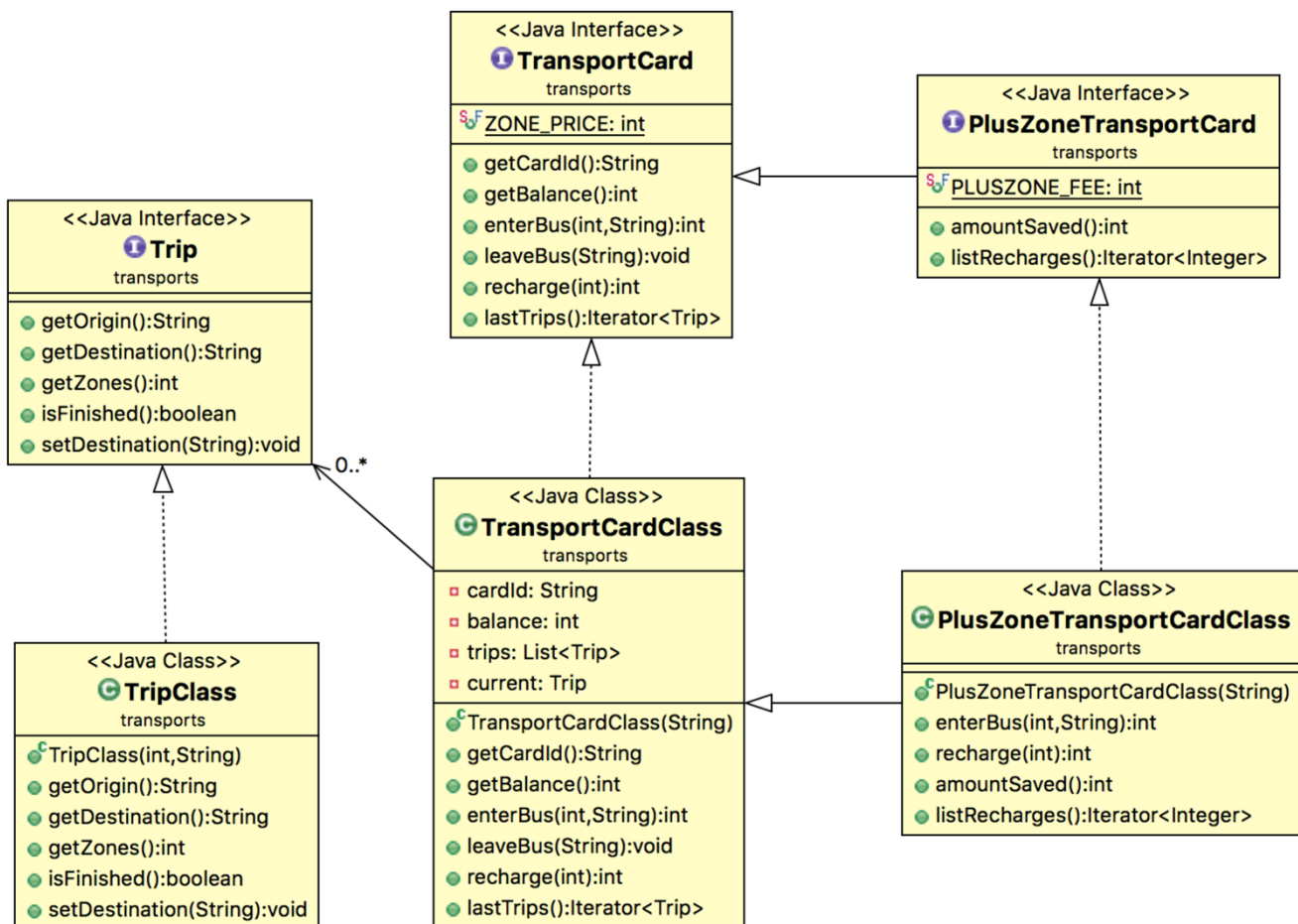


Figura 1 - Cartão de transporte: interfaces e classes.

A interface `TransportCard` representa um cartão de transporte. Um cartão de transporte tem um identificador e um saldo em euros (para simplificar assumimos que todos os valores numéricos são inteiros). O saldo diminui com viagens do utilizador do cartão e aumenta graças a carregamentos efetuados pelo mesmo. O cartão contém ainda as 10 últimas viagens efetuadas com o cartão. Cada viagem tem um local de origem, um local destino e o número de zonas da viagem. O custo de cada viagem depende do número de zonas, sendo que o custo por zona é determinado pela constante `ZONE_PRICE`. Na interface `TransportCard`:

- `getCardId` devolve uma `String` com o identificador do cartão de transporte.
- `getBalance` devolve o saldo do cartão de transporte.
- `enterBus` inicia uma viagem do utilizador do cartão. Este método recebe como argumentos o número de zonas (`zones`) a percorrer e a localização (`location`) de entrada no autocarro, e devolve o saldo após o pagamento da viagem. Esta operação lança as exceções:
`AlreadyTravelingException` no caso de já existir uma viagem iniciada e não terminada;
`NoBalanceException` no caso de não existir saldo suficiente para a viagem (o custo da viagem é determinado pelo número de zonas multiplicado pelo preço por zona).
- `leaveBus` regista o fim da viagem do utilizador do cartão (registra a saída do autocarro). Este método recebe como argumento a localização (`location`) de saída. Esta operação lança a exceção `NotTravelingException` no caso de não existir uma viagem iniciada.
- `recharge` recebe como argumento a quantia a carregar no cartão, incrementa o saldo com esse valor e devolve o novo saldo do cartão. Pode assumir que a quantia é um valor inteiro positivo.

- `lastTrips` devolve um iterador para as 10 últimas viagens efectuadas com o cartão de transporte. Este iterador deve percorrer as viagens por ordem inversa à ordem de inserção (o mais recente primeiro). Caso ainda não existam viagens o método deve devolver `null`.

A classe `TransportCardClass` implementa a interface `TransportCard`. Nesta classe o construtor recebe como argumento o identificador do cartão (`cardId`). Os métodos implementados nesta classe obedecem à especificação já apresentada durante a descrição da interface `TransportCard`. Esta classe inclui 4 variáveis privadas: `cardId` (`String`), `balance` (`int`), `trips` (`List<Trip>`) e `current` (`Trip`).

A interface `TransportCard` é especializada pela interface `PlusZoneTransportCard` que representa um cartão de transporte para utilizadores que fazem viagens mais longas (com maior número de zonas). Este tipo de cartão paga uma taxa de carregamento, mas em contrapartida o custo de uma viagem corresponde ao custo de apenas uma zona, independentemente das zonas percorridas. Esta taxa é definida pela constante `PLUSZONE_FEE`. Estes cartões guardam também todos os carregamentos efectuados com o cartão. Na interface `PlusZoneTransportCard`:

- `amountSaved` determina qual a quantia poupada com a utilização deste cartão. Esta quantia é determinada fazendo a diferença entre:
 - o somatório do custo de cada viagem registada no cartão `PlusZoneTransportCard` calculado como no cartão normal (ou seja, considerando o número de zonas) e
 - o somatório do custo de cada viagem registada no cartão `PlusZoneTransportCard` calculado como no cartão `PlusZoneTransportCard` (ou seja, considerando que cada viagem só percorre uma zona), adicionando o custo da taxa de carregamento do cartão `PlusZoneTransportCard` pago em cada carregamento.
- `listRecharges` devolve um iterador para os valores de todos os carregamento efectuados no cartão. Este iterador deve percorrer os carregamentos por ordem de inserção. Caso ainda não tenham sido feitos carregamentos o método deve devolver `null`.

Note que a classe `PlusZoneTransportCard` redefine os seguintes métodos: `enterBus` e `recharge`. No método `recharge` se a quantia de carregamento for inferior ao custo da taxa de carregamento, o método não faz nada. Para esta classe vai poder definir as variáveis de instância que achar necessárias.

Embora não tenha que implementar a classe `TripClass`, segue-se uma descrição resumida do construtor e métodos públicos:

- O construtor recebe o número de zonas e o local de origem da viagem.
- `getOrigem`, `getDestination`, `getZones` devolvem respectivamente, a origem, o destino e o número de zonas da viagem.
- `isFinished` devolve `true` se a viagem já tiver um destino definido e `false` caso contrário.
- `setFinish` recebe como argumento uma localização e regista essa localização como o destino da viagem.

Grupo I – Implementação das classes `TransportCardClass` e `PlusZoneTransportCardClass`

Neste grupo tem que apresentar a **implementação completa** das seguintes classes:

- a) Implemente a classe `TransportCardClass`. Note que as variáveis desta classe já estão definidas na Fig. 1 (variáveis privadas `cardId`, `balance`, `trips` e `current`). Não pode adicionar mais variáveis a esta classe ou alterar a sua visibilidade.
- b) Implemente a classe `PlusZoneTransportCardClass`. Ao contrário da classe `TransportCardClass` que já incluía a declaração das variáveis, para a classe `PlusZoneTransportCardClass` pode definir as variáveis de instância que achar necessárias.

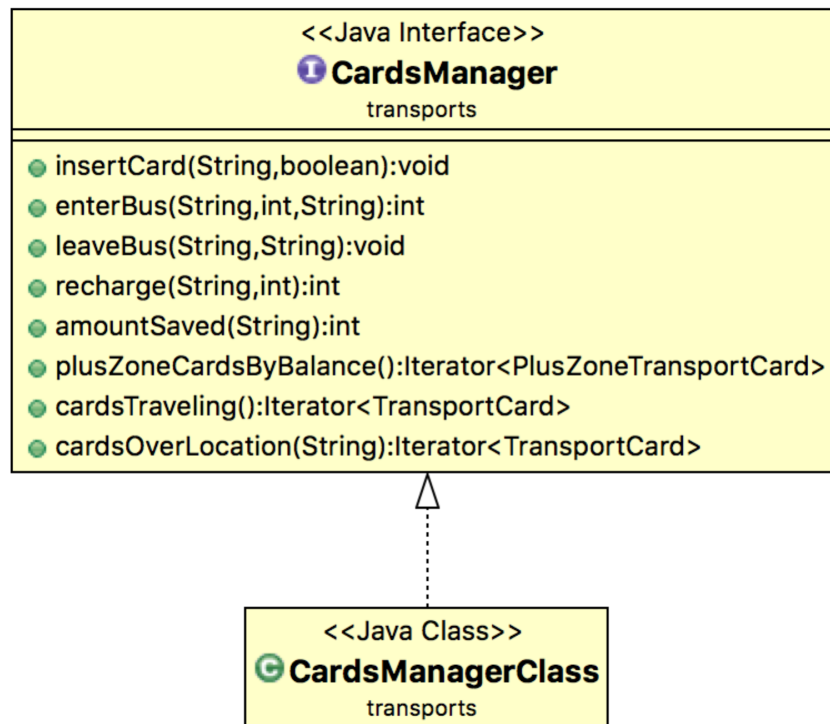


Figura 2: A interface CardsManager e a sua implementação.

A Fig. 2 apresenta a interface e a classe necessária à resolução do Grupo II. A interface `CardsManager` representa a interface para a gestão dos cartões de transporte.

- `insertCard` adiciona um novo cartão de transporte e recebe como argumentos o identificador do cartão (`cardId`) e um booleano (`isPlusCard`) que indica se o cartão é do tipo `PlusCardTransporteCard`. Se já existir um cartão com o identificador dado é lançada a exceção `ExistingCardException`.
- `enterBus` inicia uma viagem para um dado cartão. Este método recebe como argumentos o identificador do cartão, o número de zonas (`zones`) a percorrer e a localização (`location`) de entrada no autocarro, e devolve o saldo após o pagamento da viagem. Esta operação lança as exceções: `NonExistingCardException` no caso de não existir um cartão com o identificador dado; `AlreadyTravelingException` no caso de já existir uma viagem iniciada e não terminada; `NoBalanceException` no caso de não existir saldo suficiente para a viagem.
- `leaveBus` regista o fim da viagem para um dado cartão. Este método recebe como argumentos o identificador do cartão e a localização (`location`) de saída. Esta operação lança as exceções: `NonExistingCardException` no caso de não existir um cartão com o identificador dado; `NotTravelingException` no caso de não existir uma viagem iniciada.
- `recharge` carrega um dado cartão. Este método recebe como argumentos o identificador do cartão e a quantia a carregar nesse cartão. Pode assumir que a quantia é um valor inteiro positivo. Esta operação lança a exceção `NonExistingCardException` no caso de não existir um cartão com o identificador dado.
- `amountSaved` determina qual a quantia poupada com a utilização de um dado cartão. Este método recebe como argumento o identificador do cartão. Esta operação lança as exceções: `NonExistingCardException` no caso de não existir um cartão com o identificador dado; `NonPlusCardException` no caso do cartão não ser um cartão do tipo `PlusCardTransporteCard`.

- `plusZoneCardsByBalance` devolve um iterador para os cartões de tipo `PlusCardTransporteCard` ordenados por ordem decrescente de saldo. Caso não existam cartões deste tipo o método deve devolver `null`.
- `cardsTraveling` devolve um iterador para os cartões que estão actualmente numa viagem num autocarro. No caso (insólito) de não haver nenhum cartão nesta situação, o método deve devolver `null`.
- `cardsOverLocation` devolve um iterador para os cartões com viagens com origem ou destino numa dada localização recebida como argumento. No caso de não haver nenhum cartão na situação descrita o método deve devolver `null`.

Grupo II – CardsManagerClass

Assuma que o número de cartões de transporte é elevado e que se pretende otimizar as pesquisas por identificador e a eficiência das listagens. Nesta classe implemente apenas:

- a) O construtor de modo a que no início não existam cartões. Apresente também a declaração das variáveis que achar necessárias (deve colocar a declaração das variáveis acima do construtor).

- b) O modificador

```
void insertCard(String idCard, boolean isPlusZone)
    throws ExistingCardException;
```

- c) O modificador

```
int enterBus(String idCard, int zones, String location)
    throws NonExistingCardException, AlreadyTravelingException,
    NoBalanceException;
```

- d) O modificador

```
int amountSaved(String idCard)
    throws NonExistingCardException, NonPlusCardException;
```

- e) O selector `Iterator<PlusZoneTransportCard> plusZoneCardsByBalance()`

Assuma que está disponível uma classe `CardsByBalanceComparator` (que implementa a interface `Comparator<TransportCard>`), que permite estabelecer uma relação de ordem parcial entre os cartões com base no saldo (ordem decrescente do saldo).

- f) O selector `Iterator<TransportCard> cardsTraveling()`.

- g) O selector `Iterator<TransportCard> cardsOverLocation(String location)`.

Grupo III – Testes unitários e asserções

- a) Implemente uma operação de teste à operação `amountSaved` da classe `PlusZoneTransportCard`, usando o JUnit. O seu teste deve construir um objecto `PlusZoneTransportCardClass`. De seguida deve fazer alguns carregamentos e viagens sobre esse cartão. Após cada uma destas operações (carregamentos e viagens) invoque o método `amountSaved` e verifique que o valor devolvido é o esperado.
- b) Considere a seguinte classe:

```
public class AssertionsClass {  
  
    public static void main(String[] args) {  
        int sum = 0;  
        int k = 4;  
        List<Integer> ns = new ArrayList<Integer>(k);  
        for(int i=0; i<k; i++) {  
            ns.add(i);  
        }  
        for(int i=0; i<k; i++) {  
            sum += i;  
        }  
        ns.clear();  
        assert ns.isEmpty();  
        System.out.println("O sumatório é: " + sum);  
        assert sum != 6: sum;  
        System.out.println("Fim.");  
    }  
}
```

Assumindo que o mecanismo de asserções está activado, qual é o resultado obtido na execução do programa acima? Justifique a sua resposta.

Grupo IV – Modelação

Pretende-se implementar uma nova versão do Pac-Man. Este jogo é definido sobre uma grelha e deve permitir:

- Criar o labirinto recebendo o número de linhas e colunas.
- Adicionar personagens de diferentes tipos: heróis e inimigos. Cada personagem tem um nome, uma posição na grelha e uma cor. Os inimigos têm uma velocidade associada (Slow, Fast);
- Adicionar obstáculos de diferentes tipos: fixos ou móveis. Cada obstáculo tem uma posição. Os obstáculos móveis têm uma direcção (Left, Right, Up, Down);
- Mover um herói dado o nome da personagem, a direcção e o número de passos.
- Consultar o número de pontos obtidos;
- Listar todas as personagens de um dado tipo (herói ou inimigo);
- Listar todos os inimigos por proximidade a um dado herói.

Apresente a sua proposta de modelação para o programa, através de um diagrama de classes e interfaces, tendo em atenção que deve incluir na sua resposta:

- A interface de topo com a qual o programa principal irá interagir. Esta interface deve ser completamente especificada.
- As variáveis de instância da classe que implementa a interface de topo.
- Para as restantes componentes do diagrama, isto é, para as interfaces e classes que não a interface de topo, omita a indicação das operações e das variáveis de instância.

Nota 1: Não é necessário implementar nenhuma das operações.

Nota 2: Apresente de forma distinta classes e interfaces, identifique caso existam classes abstractas e indique a visibilidade das variáveis de instância.

Nota 3: Para efeitos de clareza da apresentação represente as relações com uma seta etiquetada, por exemplo

$A \xrightarrow{\text{extends}} B$ significa *A extends B*. Analogamente para implements, eventualmente com linha tracejada mas sempre etiquetada com *implements* e no caso do contains, uma seta no sentido dos elementos da colecção etiquetada com *contains*.

Algumas das interfaces abaixo reproduzidas poderão ser úteis na resolução deste exame:

<div> List<E> </div> <ul style="list-style-type: none"> ● add(E) : boolean ● add(int, E) : void ● addAll(int, Collection<? extends E>) : boolean ● addAll(Collection<? extends E>) : boolean ● clear() : void ● contains(Object) : boolean ● containsAll(Collection<?>) : boolean ● equals(Object) : boolean ● get(int) : E ● hashCode() : int ● indexOf(Object) : int ● isEmpty() : boolean ● iterator() : Iterator<E> ● lastIndexOf(Object) : int ● listIterator() : ListIterator<E> ● listIterator(int) : ListIterator<E> ● remove(int) : E ● remove(Object) : boolean ● removeAll(Collection<?>) : boolean ● retainAll(Collection<?>) : boolean ● set(int, E) : E ● size() : int ● subList(int, int) : List<E> ● toArray() : Object[] ● toArray(T[]) <T> : T[] 	<div> Map<K, V> </div> <ul style="list-style-type: none"> ● clear() : void ● containsKey(Object) : boolean ● containsValue(Object) : boolean ● entrySet() : Set<Entry<K, V>> ● equals(Object) : boolean ● get(Object) : V ● hashCode() : int ● isEmpty() : boolean ● keySet() : Set<K> ● put(K, V) : V ● putAll(Map<? extends K, ? extends V>) : void ● remove(Object) : V ● size() : int ● values() : Collection<V>
<div> Set<E> </div> <ul style="list-style-type: none"> ● add(E) : boolean ● addAll(Collection<? extends E>) : boolean ● clear() : void ● contains(Object) : boolean ● containsAll(Collection<?>) : boolean ● equals(Object) : boolean ● hashCode() : int ● isEmpty() : boolean ● iterator() : Iterator<E> ● remove(Object) : boolean ● removeAll(Collection<?>) : boolean ● retainAll(Collection<?>) : boolean ● size() : int ● toArray() : Object[] ● toArray(T[]) <T> : T[] 	<div> SortedMap<K, V> </div> <ul style="list-style-type: none"> ● comparator() : Comparator<? super K> ● entrySet() : Set<Entry<K, V>> ● firstKey() : K ● headMap(K) : SortedMap<K, V> ● keySet() : Set<K> ● lastKey() : K ● subMap(K, K) : SortedMap<K, V> ● tailMap(K) : SortedMap<K, V> ● values() : Collection<V> <div> SortedSet<E> </div> <ul style="list-style-type: none"> ● comparator() : Comparator<? super E> ● first() : E ● headSet(E) : SortedSet<E> ● last() : E ● subSet(E, E) : SortedSet<E> ● tailSet(E) : SortedSet<E>