

Programação Orientada pelos Objectos

Exame de Recurso (Duração 3h)

MIEI 2016/2017

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 uma aplicação para a gestão e armazenamento de fotos na *cloud*. 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 das contas dos utilizadores da aplicação (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.

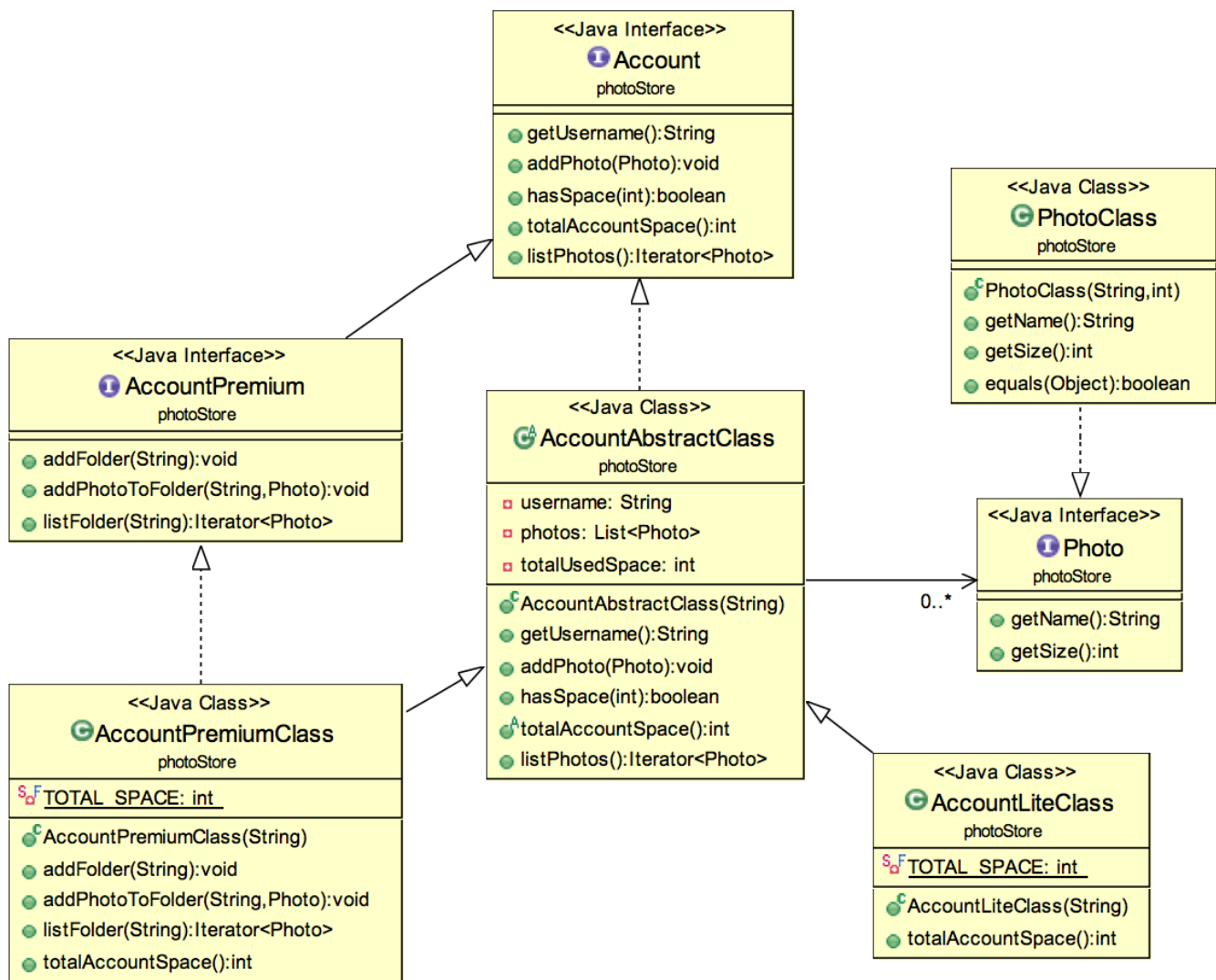


Figura 1 – Conta de um utilizador: interfaces e classes.

A interface `Account` representa a conta de um utilizador registado na aplicação. Cada conta é identificada pelo `username`, tem uma capacidade máxima e mantém as fotos adicionadas pelo utilizador.

Cada foto tem como identificador o nome do ficheiro e tem uma dimensão em KB (kilobytes). Na interface `Account`:

- `getUsername` devolve uma `String` com o `username` da conta.
- `addPhoto` adiciona uma foto à conta do utilizador. Este método recebe como argumento um objecto do tipo `Photo`. O método lança as excepções: `PhotoAlreadyExistsException` no caso dessa foto já existir; `InsufficientAvailableSpaceException` caso de não exista espaço suficiente na conta para armazenar a foto.
- `hasSpace` recebe como argumento uma dimensão (`size`) e devolve `true` caso o espaço livre na conta seja igual a superior a `size` e `false` caso contrário
- `totalAccountSpace` devolve a capacidade total inicial da conta.
- `listPhotos` devolve um iterador para as fotos da conta. Este iterador deve percorrer as fotos por ordem inversa à ordem de inserção (a mais recente primeiro). Caso ainda não existam fotos o método deve lançar a excepção `NoAvailablePhotosException`.

A classe abstracta `AccountAbstractClass` implementa a interface `Account`. Nesta classe o construtor recebe como argumento o `username` da conta (`username`). Os métodos implementados nesta classe obedecem à especificação já apresentada durante a descrição da interface `Account`, sendo que o método

`totalAccountSpace` é abstracto. Esta classe inclui 3 variáveis privadas: `username (String)`, `photos (List<Photo>)` e `totalUsedSpace (int)`.

A aplicação tem dois tipos de contas, as contas *Lite* e as contas *Premium*. As contas *Lite* são gratuitas e tem uma capacidade baixa. Enquanto, as contas *Premium* são pagas e, para além de terem maior capacidade, permitem organizar as fotos em diferentes pastas.

A classe `AccountLiteClass` representa uma conta *Lite*. Esta classe estende a classe abstracta `AccountAbstractClass` e implementa o método `totalAccountSpace`. Este método devolve o valor da constante `TOTAL_SPACE` que define a capacidade das contas *Lite*.

A classe `AccountPremiumClass` representa uma conta *Premium*. Esta classe estende a classe abstracta `AccountAbstractClass` e implementa o método `totalAccountSpace`, para além dos métodos adicionais definidos na interface `AccountPremium`:

- `addFolder` recebe como argumento o nome da pasta e adiciona uma pasta vazia à conta. Caso já exista uma pasta com o nome dado é lançada a excepção `FolderAlreadyExistsException`.
- `addPhotoToFolder` recebe como argumentos o nome da pasta e um objecto do tipo `Photo`. Este método adiciona uma foto à conta do utilizador (tal como a operação `addPhoto`) e à pasta recebida como argumento. O método lança as excepções: `PhotoAlreadyExistsException` no caso desse foto já existir; `FolderDoesNotExistException` no caso a pasta não existir.
- `listFolder` devolve um iterador para as fotos da pasta recebida como argumento. Este iterador deve percorrer as fotos por ordem inversa à ordem de inserção (a mais recente primeiro). Caso a pasta não existe o método deve lançar a excepção `FolderDoesNotExistException`.

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

- O construtor recebe o nome do ficheiro e a dimensão da foto.
- `getName` e `getSize` devolvem respectivamente, o nome e a dimensão da foto.
- `equals` recebe como argumento um objecto do tipo `Object` e devolve `true` caso esse objecto seja uma foto com o mesmo nome.

Grupo I – Implementação das classes `AccountAbstractClass` e `AccountPremiumClass`

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

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

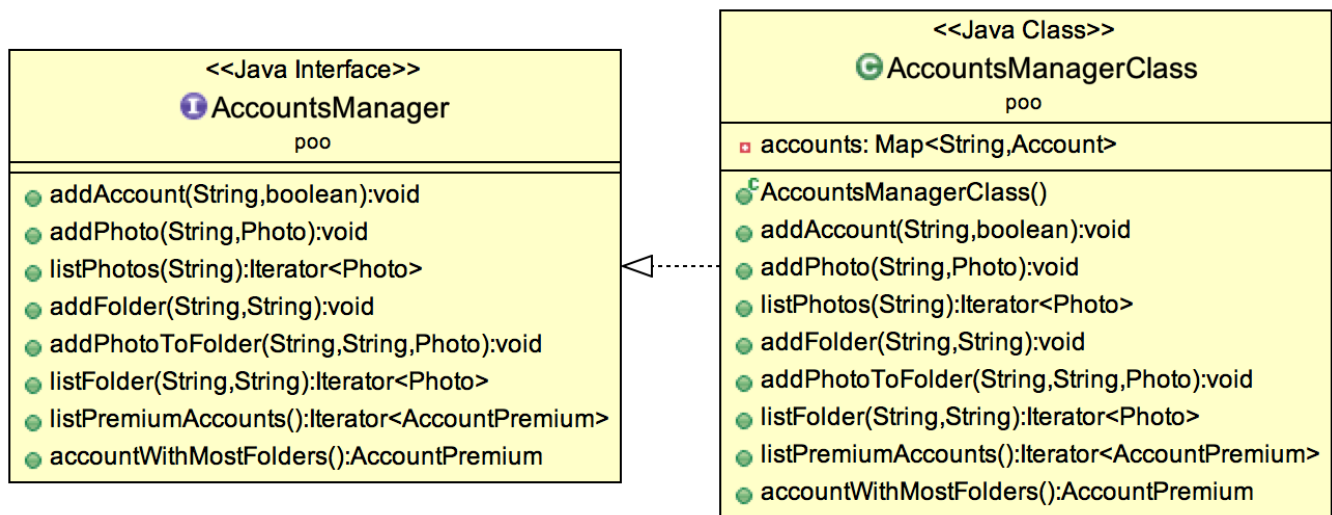


Figura 2: A interface `AccountsManager` e a sua implementação.

A Fig. 2 apresenta a interface e a classe necessária à resolução do Grupo II. A interface `AccountsManager` representa a interface para a gestão de contas.

- `addAccount` adiciona uma nova conta e recebe como argumento o username do utilizador (username) e um booleano (`isLite`) que indica se a conta é uma conta *Lite*. Se já existir uma conta com o username dado é lançada a exceção `AccountAlreadyExistsException`.
- Os métodos `addPhoto`, `listPhotos`, `addFolder`, `addPhotoToFolder` e `listFolder` são similares aos métodos com o mesmo nome definidos no Grupo I, excepto no facto de receberem como argumento adicional o username da conta sobre a qual devem ser aplicados. Em todos estes métodos deve ser lançada a exceção `AccountDoesNotExistException` caso não exista uma conta com o username dado. Esta exceção também deve ser lançada no caso em que se invocam operações da interface `PremiumAccount` sobre contas *Lite*.
- `listPremiumAccounts` devolve um iterador para as contas de tipo `PremiumAccount` ordenadas por ordem alfabética de username. Caso não existam contas deste tipo o método deve lançar a exceção `NoPremiumAccountsException`.
- `accountWithMostFolder` devolve a conta *Premium* com mais pastas. Em caso de empate deve devolver uma das conta com o número máximo de pastas. No caso de não haver nenhuma conta *Premium* com pastas, o método deve devolver `NoPremiumAccountsException`.

Grupo II – AccountsManagerClass

Assuma que o número de contas é elevado e que se pretende otimizar a eficiência das listagens. Nesta classe pode assumir que já existe a variável `accounts`:

```
private Map<String, Account> accounts;
```

sendo que a chave é o username da conta.

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

- b) O modificador

```
void addAccount(String username, boolean isLite)
    throws AccountAlreadyExistsException;
```

- c) O selector

```
Iterator<Photo> listPhotos(String username)
    throws AccountDoesNotExistException, NoAvailablePhotosException;
```

- d) O modificador

```
void addPhotoToFolder(String username, String folder, Photo photo)
    throws AccountDoesNotExistException, FolderDoesNotExistException,
        PhotoAlreadyExistsException, InsufficientAvailableSpaceException;
```

- e) O selector

```
Iterator<AccountPremium> listPremiumAccounts()
    throws NoPremiumAccountsException
```

- f) O selector

```
AccountPremium accountWithMostFolders() throws NoPremiumAccountsException;
```

Grupo III

Considere que existe um método adicional da classe `PremiumAccountClass`:

```
public int oldestPhotoInAFolder().
```

Este método determina qual é a foto mais antiga dentro das pastas da conta e devolve a sua ordem de inserção na conta. Implemente este método.

Exemplo:

Após a seguinte sequência de interacção o método `oldestPhotoInAFolder` iria devolver o número 3, porque a pasta `birthday party` tem a 3ª foto a ser inserida, enquanto a pasta `trip to rome` tem a 5ª foto da conta.

```
PremiumAccount foo = new PremiumAccountClass("foo");  
  
foo.addPhoto(new PhotoClass("photo1.jpg",40));  
  
foo.addPhoto(new PhotoClass("photo2.jpg",120));  
  
foo.addFolder("trip to rome");  
  
foo.addFolder("birthday party");  
  
foo.addPhotoToFolder("birthday party", new PhotoClass("cake.jpg",110));  
  
foo.addPhoto(new PhotoClass("another_photo",90));  
  
foo.addPhotoToFolder("trip to rome", new PhotoClass("collosseum" 140));
```

Grupo IV – Testes unitários

Implemente uma operação de teste à operação `addPhotoToFolder` da classe `AccountPremiumClass`, usando o JUnit. O seu teste deve construir um objecto `AccountPremiumClass`. De seguida deve criar duas pastas na conta e inserir várias fotos nessas pastas. Deve verificar através dos métodos `listFolder` e `listPhotos` que a operação funciona como esperado.

Grupo V – Modelação

Com o aumento do turismo pretende-se implementar um programa que permita gerir as carreiras dos carros eléctricos de Lisboa. Apresente a sua proposta de modelação para o programa, através de um diagrama de classes e interfaces, tendo em atenção os seguintes aspectos:

- Carreiras: cada carreira compreende várias paragens, em número superior a dois.
- Paragens: uma paragem pode pertencer a várias carreiras, embora a sua existência seja independente dos carreiras em operação.
- Horários de uma carreira: um horário caracteriza-se por uma sequência de pares hora/minuto associado à chegada do eléctrico a cada uma das paragens do percurso. Os horários podem ainda ser classificados como horários de semana ou de fim-de-semana/feriado.

As operações fundamentais no programa que gere as carreiras são as seguintes:

- Adição de uma carreira; activação/desactivação do funcionamento de uma carreira.
- Adição de uma paragem; activação/desactivação de uma paragem.
- Consulta carreiras e paragens activas/desactivas.
- Planeamento de uma viagem para um passageiro, dadas as paragens de origem e destino, bem como o momento (tempo) a partir do qual pretende viajar. A resposta deve indicar todas as opções possíveis.

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:

List<E> ● 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[]	Map<K, V> ● 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>
Set<E> ● 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[]	SortedMap<K, V> ● 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> SortedSet<E> ● comparator() : Comparator<? super E> ● first() : E ● headSet(E) : SortedSet<E> ● last() : E ● subSet(E, E) : SortedSet<E> ● tailSet(E) : SortedSet<E>