Programação Orientada por Objectos (POO)

Semestre de Inverno de 2013-2014

1º Trabalho prático

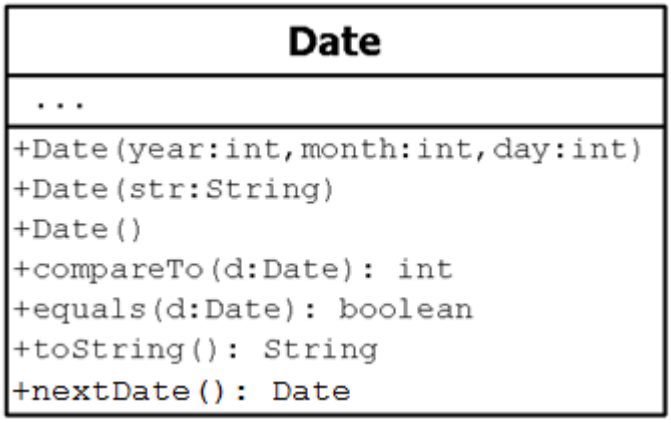
Data de Entrega: 29 de Outubro de 2013

OBJETIVOS: Completar e desenvolver aplicações simples usando o paradigma da Programação Orientada por Objectos.

NOTA: tem que constar todo o código desenvolvido, incluindo os testes unitários que permitem validar a correcção dos métodos e classes realizadas.

Grupo 1

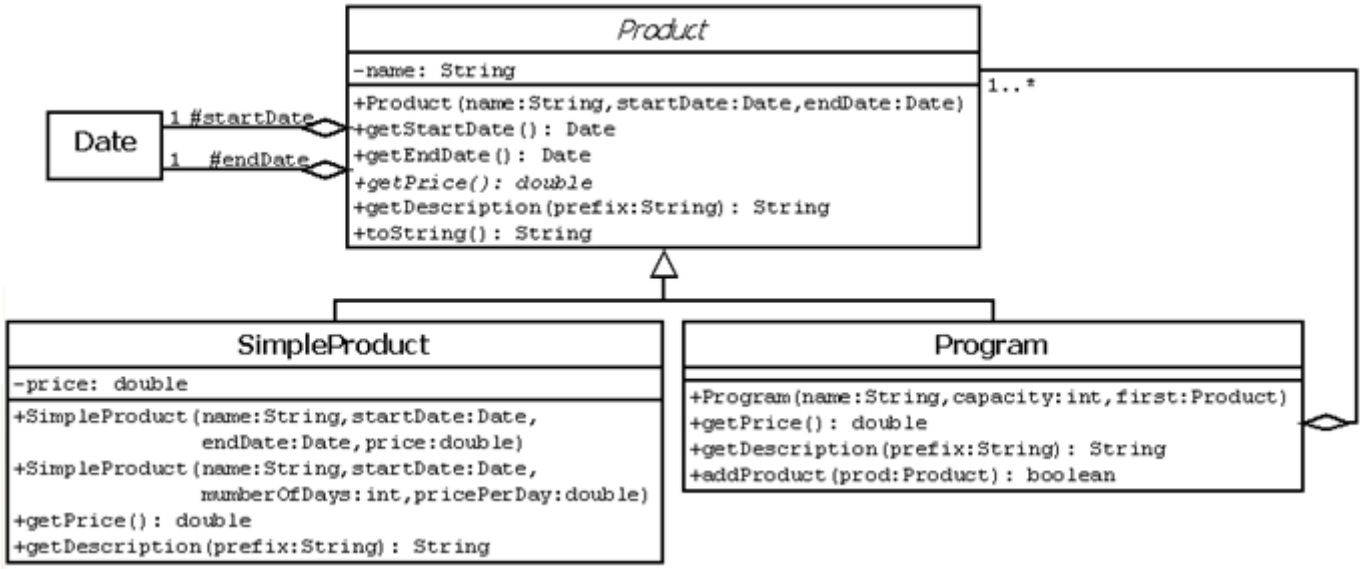
1. Implemente a classe Date tendo em conta que:



- O formato da *string* passada por parâmetro ao construtor com um parâmetro é DD-MM -YYYY.
- O construtor sem parâmetros inicia o dia o mês e o ano com a data corrente. Usar da classe **`java.util.Calendar`**:
  - O método estático **`getInstance()`** para obter um objeto **`Calendar`** com a data corrente;
  - O método de instância **`get(int field)`** para obter o ano, o mês, e o dia do objeto **`Calendar`**.
  - As constantes **`YEAR`**, **`MONTH`** e **`DAY_OF_MONTH`** para passar como argumento ao método **`get`**.
- O método **`compareTo`** define uma relação de ordem sobre as instâncias da classe Date. Sejam **`d1`** e **`d2`** dois objetos do tipo **`Date`** e **`x`** um valor inteiro tal que **`x = d1.compareTo(d2)`**. Se:
  - **`x<0`**, significa que a data **`d1`** é anterior à data **`d2`**;
  - **`x>0`**, significa que a data **`d1`** é posterior à data **`d2`**;
  - **`x==0`**, significa que a data **`d1`** é igual à data **`d2`**.
- O método **`toString`** devolve uma *string* com o formato DD-MM-YYYY.
- O método **`nextDate`** devolve a data do dia seguinte.

Grupo 2

Pretende-se implementar uma solução simplificada para a gestão de produtos de uma agência de viagens. Um produto pode ser simples (uma viagem, um alojamento, etc.), ou um programa, sendo que, um programa é constituído por um conjunto de produtos (produtos simples ou outros programas). O preço de um programa é a soma dos preços dos produtos nele contidos. Para o efeito considere o seguinte diagrama de classes.



- Implemente a classe abstrata **`Product`** exceto os métodos **`toString`** e **`getDescription`** que têm a seguinte implementação. **De notar que o método `getPrice` é abstracto**.

```
public String getDescription( String prefix )
  { return prefix + "De " + startDate + " a " + endDate + ", " + name; }
public final String toString() { return getDescription(""); }
```

- Implemente a classe **`SimpleProduct`** de forma que o seguinte troço de código produza os resultados indicados.

```
SimpleProduct vAcores = new  SimpleProduct("Açores",
                                    new Date(2012,3,1), new Date(2012,3,14), 700);
SimpleProduct vMadeira = new SimpleProduct("Madeira",
                                    new Date(2012, 3, 15), 7, 75);

System.out.println( vAcores );
System.out.println( vMadeira );
```

```
De 1-3-2012 a 14-3-2012, Açores, 700€
De 15-3-2012 a 22-3-2012, Madeira, 525€
```

- Implemente a classe **`Program`** tendo em conta o seguinte:

- O construtor recebe por parâmetro o nome, o máximo de produtos do programa e um dos produtos do programa;
- **`getPrice`** – devolve a soma dos preços de todos os produtos do programa;
- **`addProduct`** – devolve **`false`** se a capacidade do programa estiver esgotada, **`true`** caso contrário. Atualiza as datas, inicial e final, do programa para que incluam as datas do produto adicionado. Este método deve ainda lançar a exceção de *runtime* **`IllegalArgumentException`** caso a referência para o produto recebido por parâmetro seja **`null`**;
- **`getDescription`** – devolve uma *string* com o nome do programa, a descrição de cada produto e o preço do programa. Segue a escrita de um programa com subprogramas, qualquer que seja a ordem com que tenham sido adicionados os produtos.

```
De 1-3-2012 a 30-3-2012, Portugal e ilhas
  De 1-3-2012 a 22-3-2012, Descubra os Açores e a Madeira
    De 1-3-2012 a 14-3-2012, Açores, 700€
    De 15-3-2012 a 22-3-2012, Madeira, 525€
  TOTAL: 1225€
   De 23-3-2012 a 30-3-2012, Lisboa, 600€
TOTAL: 1825€
```

Grupo 3

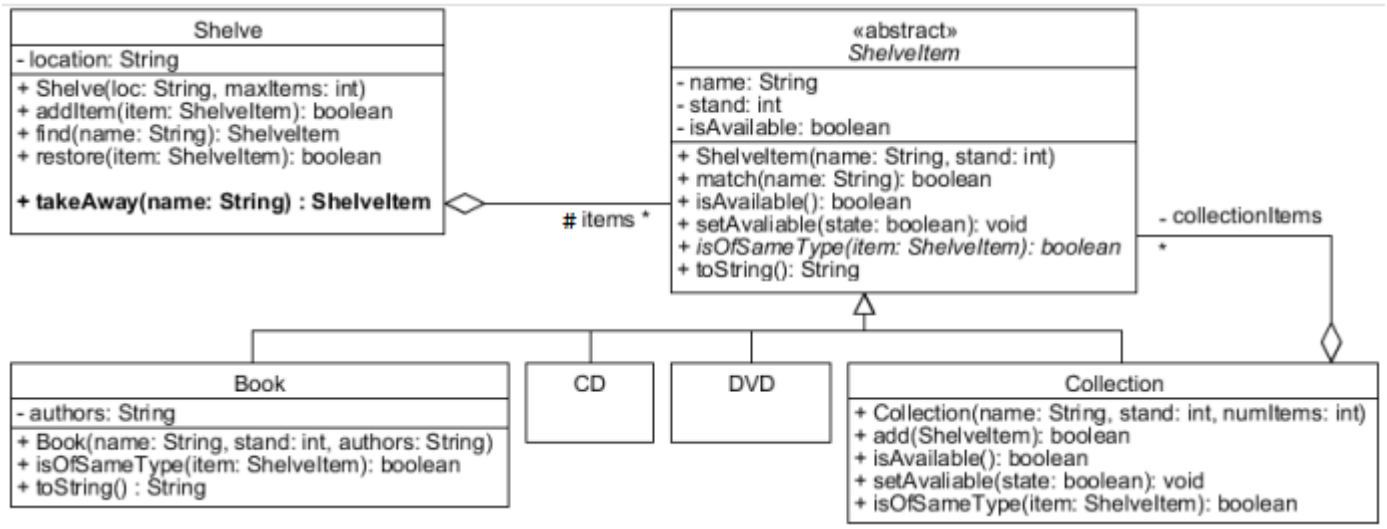
Tendo em conta a seguinte definição dos tipos `Student` e `Filters`:

```
public class Student {
   public String name;
   public int courseId;
   public Student(String name, int courseId) { this.name = name; this.courseId = courseId; }
}
public class Filters {
   public static Student[] filterStudents( Student[] students, FilterCriteria criteria ) {
      Student[] res = new Student[students.length];
      int resCount = 0;
      for (int i = 0; i < students.length; i++) {
         Student student = students[i];
         if (criteria.filter(student, i)) res[resCount++] = student;
      }
      return Arrays.copyRange( res, 0, resCount );
   }
}
```

1. Implemente o tipo **`FilterCriteria`**, usado pelo método **`Filters.filterStudents`**, que define o contrato de filtragem de alunos.
2. Crie o critério de filtragem **`OddFilterCriteria`** que filtra alunos em índices ímpares do *array*.
3. Crie o critério de filtragem **`CourseFilterCriteria`** que filtra os alunos do curso recebido por parâmetro no construtor.
4. Crie o critério de filtragem **`NotFilterCriteria`** que recebendo como parâmetro um critério, filtra os alunos de forma inversa. Por exemplo, se receber como parâmetro uma instância de **`OddFilterCriteria`**, filtra os alunos em índices pares em vez de ímpares mas se receber por parâmetro uma instância de **`CourseFilterCriteria`** filtra os alunos que não pertencem ao curso.
5. Usando os critérios de filtragem implementados nas alíneas 2 e 4, e o método **`filterStudent`**, realize um troço de código que separa o *array* de alunos pooS**`tudents`** em dois *arrays*: um com os alunos em índice par e ímpar respetivamente.
6. Acrescente à classe **`Filters`** o método **`separate`** que recebendo por parâmetro um *array* de alunos e um critério de filtragem retorne o *array* separado em dois *arrays*: um com os alunos que cumprem o critério de filtragem e outro com os que não o cumprem.
7. Usando o método **`separate`** reformule o troço de código da alínea 5.
8. Usando o critério de filtragem implementado na alínea 3 e o método **`separate`**, apresente um troço de código que escreva na consola os alunos que pertencem ao mesmo curso que o que se encontra na primeira posição do *array* de alunos **`iselStudents`** e seguidamente os que não pertencem.

Grupo 4 (opcional)

Pretende-se implementar uma solução para gestão de estantes (**`Shelve`**), sendo uma estante composta por diferentes prateleiras (**`stand`**). A estante contém elementos (**`ShelveItem`**) que podem ser Livros (**`Book`**), CDs, DVDs e coleções (**`Collection`**) de elementos (e.g. A trilogia do Senhor do Anéis). Um elemento é caracterizado por um nome, a prateleira onde tem o lugar reservado e se está disponível (pode estar temporariamente fora do lugar).



Tendo em conta o diagrama estático de classes apresentado em cima, e o troço de código apresentado em baixo, responda às seguintes questões:

```
public abstract class ShelveItem {
   private final String name;              // Nome do elemento
   private int stand;                      // Prateleira em que se encontra
   private boolean isAvailable = true;     // Indica se o elemento está disponível
   public ShelveItem(String n, int stand)  { this.name = n; this.stand = stand; }
   public boolean match(String n)          { return name.equals(n);              }
   public boolean isAvaliable()            { return isAvailable;                 }
   public void setAvaliable(boolean state) { isAvaliable = state;                }
   public String toString()                { return stand + ": " + name;         }
   public abstract boolean isOfSameType(ShelveItem item);
}
```

1. Implemente o tipo **`Book`** tendo em conta que o método **`isOfSameType`** apenas retorna **`true`** se o item recebido como parâmetro for uma instância de **`Book`**; e que o método **`toString`**, deve retornar uma *string* com o número da prateleira, o nome e os autores do livro.

2. Implemente o tipo **`Collection`** tendo em conta:

   - Uma coleção agrupa um conjunto limitado de elementos.
   - Suporta a adição de elementos, caso não exceda o limite. O método **`add`** retorna **`false`** se não foi possível adicionar por a coleção estar completa (cheia). Este método deve ainda lançar a exceção de *runtime* **`IllegalArgumentException`** caso seja adicionado um item de tipo diferente de outro que já exista na coleção (e.g. adicionar um livro e depois um CD).
   - Uma coleção apenas está disponível se todos os elementos nela agrupados também estiverem.
   - Colocar uma coleção disponível ou indisponível (**`setAvaliable`**) altera o estado de todos os elementos agrupados.
   - Qualquer instância de **`Collection`** é sempre de tipo diferente (**`isOfSameType`**) de qualquer outra instância.

3. Implemente o tipo Shelve de forma a que:

   - O método **`find`** retorne a referência para o elemento com nome igual ao passado por parâmetro ou **`null`** se o elemento não existir.
   - O método **`takeAway`** retire da estante (coloque indisponível) o elemento cujo nome é igual ao passado por parâmetro, caso exista e esteja disponível. Este método retorna o elemento retirado da estante ou **`null`** se o elemento não existir ou estiver indisponível.
   - O método **`restore`** coloque na estante (coloque disponível) o elemento cuja referência é passada por parâmetro. Caso o elemento exista na estante só o coloca como disponível, caso contrário se a estante não estiver completa adiciona-o e coloca-o disponível. Este método só retorna **`true`** se o elemento ficar disponível.

Bom trabalho