Object-Oriented Programming. Exercise 2.1

Unit 2. Classes, Objects and Composition

Description of the exercise

In this exercise, we will experiment with the definition of classes and the creation of objects. Also, we will learn how objects are made up of attributes (instance variables, which can also be constants) that store the internal state of the objects. The constructors allow us to create the objects and assign initial values to the object's attributes. The methods allow manipulating and accessing the internal state of the objects.

In addition, in the second part of the exercise, we will practice with the concept of **composition**, as one of the fundamental pillars in the design of classes, in such a way that it allows us to define new more complex classes through the composition of simpler classes previously defined. We will also learn how a constructor can invoke another constructor in the creation of an object.

Exercise 1. (project prJugs)

The goal of this exercise is to create a `Jug` class belonging to the `prJugs` package of the `prJugs` project. We will use this `Jug` class to *simulate* some of the actions we can perform with a jug.

Our jugs will be able to contain a certain amount of water. Thus, each jar has a certain capacity (in liters) that will be the same (constant) during the life of the jar, and its value will be specified at the time of the construction of the object. At a certain time a jug will have an amount of water that can vary over time. The actions we can perform on a jar are:

- Fill the jug completely from a tap.
- Empty the jar completely.
- Fill the jug with water that contains another jug (either until the receiving jug is full or until the jug that we tip over completely empties).
- Access your textual representation.

For example: We have two jugs **A** and **B** with capacities 7 and 4 liters, respectively. We can fill the jar **A** (we cannot pour less than the total of the jar because we would not know for sure how much water it would have). Then, dump **A** on **B** (not everything fits, so in **A** there are 3 liters left and **B** is full). Now empty **B**. Then dump again **A** on **B**. In this situation, **A** is empty and **B** has 3 liters.

We have to define the `Jug` class with the necessary methods to perform the operations just described. Therefore, the `Jug` class contains two instance variables, so that the `capacity` instance variable is constant (`final`) and determines the capacity of the `Jug` object, while the instance variable `content` determines the amount of liquid contained at a given time in the `Jug` object, considering that the contents of the jug may vary over time. In addition, the `Jug` class defines the following constructors and public methods that will allow manipulating objects of the `Jug` class:

- `Jug(int)`

  Builds a new `Jug` object with the *capacity* that is received as a parameter, and the *content* of the empty jug. If the value received as a parameter is less than or equal to zero, then it will throw a `RuntimeException` exception.

- `getCapacity(): int`

  Returns the *capacity* of the jug object.

- `getContent(): int`

Returns the *content* of the jug object.

- `fill(): void`

  Fills the jug object up the maximum of its capacity.

- `empty(): void`

  Completely empties the contents of the jug object.

- `pourFrom(Jug): void`

  Fills the *content* of the current (receiver) jug with the *content* of the jug passed as a parameter (source), either until the target jug is full or until the source jug is completely emptied. If the current object (`this`) is the same object as the object received as a parameter, then the method **will not perform** any action and will throw a `RuntimeException` exception.[1]

- `toString():String  // [@Redefinition]`

  Returns a `String` with the representation of the jug object that receives the message in the format `J(`*cap*`,`*cnt*`)`.

The figure below depicts the UML diagram of the `Jug` class.

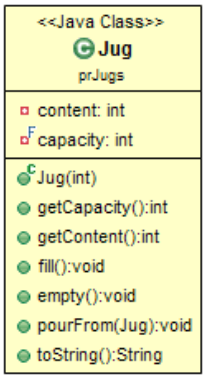

Figura 1: UML class diagram

To test the `Jug` class we will build an application (distinguished class `MainJug`) in the *default* package (*anonymous package*):

- `MainJug`: two jugs will be created, one (`jugA`) with capacity for 7 liters and another (`jubB`) for 4 liters. Once created, we will perform the following operations: fill `jugA`, show both jugs on the screen, pour the contents of `jugA` on `jugB`, show both jugs on screen, empty `jugB`, display both jugs on screen, pour the contents of `jugA` on `jugB`, and show both jugs on the screen. The following out is expected:

```
J(7, 7), J(4, 0)
J(7, 3), J(4, 4)
J(7, 3), J(4, 0)
J(7, 0), J(4, 3)
```

Exercise 2. (project prJugs)

In this exercise, we will define a `Table` class also belonging to the `prJugs` package of the `prJugs` project. This class contains two jugs (defined in the previous exercise). Thus, when a table is constructed, the capacity of each of the two jugs that make up the table object being constructed will be specified. The two jugs will be built with the specified capacity and empty content. In addition, it will also be possible to build a table by providing the two jugs that the table will contain.

[1] The comparison operator `==` may be useful to check if two variables refer to the same object.

We can also perform the following actions with the jugs on the table, specifying the jug we act on using its identification number, either 1 or 2 (in case of receiving an incorrect jug identification, a `RuntimeException` exception will be thrown):

- `Table(Jug,Jug)`

  Builds a new `Table` object that contains two jugs, which are received as parameters. If the received objects are the same object, then a `RuntimeException` exception will be thrown.[2]

- `Table(int,int)`

  Builds a new `Table` object that contains two jugs, which are created with the *capacities* received as parameters, and the *content* of the empty jugs.

- `getCapacity(int): int`

  Returns the *capacity* of the specified jug, which can be either 1 or 2.

- `getContent(int): int`

  Returns the *content* of the specified jug, which can be either 1 or 2.

- `fill(int): void`

  Fills up the *content* of the specified jug, which can be either 1 or 2.

- `empty(int): void`

  Empties the *content* of the specified jug, which can be either 1 or 2.

- `pourFrom(int): void`

  Fills up the *content* of the receiver jug with the *content* of the source jug, specified as parameter, and which may be either 1 or 2, either until the receiving jug is full or until the source jug is completely emptied.

- `toString(): String  // [@Redefinition]`

  Returns a `String` with the representation of the table object that receives the message in the format `M(J(cap,cnt),J(cap,cnt))`.

The figure below depicts the UML diagram of the `Table` class.

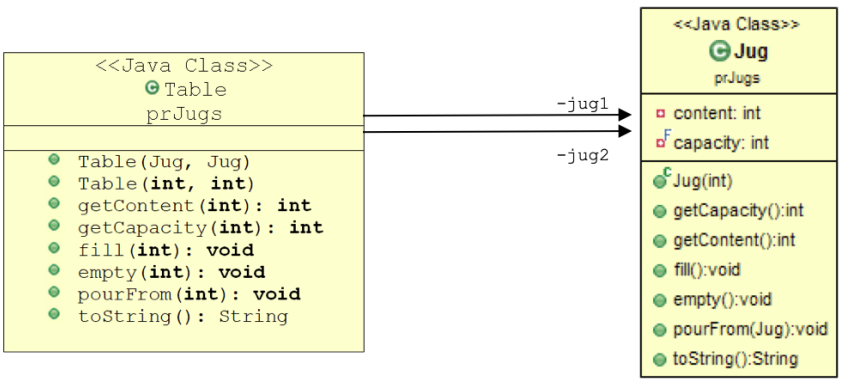

Figura 2: UML class diagram

To test the `Table` class we will build an application (distinguished class `MainTable`) in the *default* package (*anonymous package*):

- `MainTable`: A table with two jugs will be created, one with a capacity for 5 liters and another for 7. Once created, we must carry out the necessary operations to leave exactly one liter of water in one of the jugs.

[2]The comparison operator `==` may be useful for checking whether two variables refer to the same object.

```
M(J(7, 0), J(5, 5))
M(J(7, 5), J(5, 0))
M(J(7, 5), J(5, 5))
M(J(7, 7), J(5, 3))
M(J(7, 0), J(5, 3))
M(J(7, 3), J(5, 0))
M(J(7, 3), J(5, 5))
M(J(7, 7), J(5, 1))
```