IS1220 - Object Oriented Software Design

Quiz 02

Paolo Ballarini

General instructions:

- If you have not done so already, create a workspace for your tutorial `IS1220/TDs`.
- Within the workspace, create a new package for this TD called `fr.ecp.IS1220.TD02/`.
- Carefully document your code (JavaDoc), i.e., explain the general idea of your algorithm, explain the relevant steps in the code implementing the algorithm, document assumptions (if any), corner cases, and error conditions.

Learning outcomes:

- Inheritance
- Methods overriding and hiding

Exercise 1. A program to manage bank accounts

Q1) You are required to conceive a program (in Java) for managing bank accounts. The program should allow to represents different kind of information, such as, *customers*, *banks*, different kind of *back accounts*. Based on the specifications given below you should come up with a proper architecture of Java classes to deal with this kind of information. Classes should obey the *encapsulation* principle.

Specifications

- A bank's customer is represented by a unique numeric ID and a name.
- A *current account* (compte courant) is represented by a unique account number, a balance, a threshold of authorized overdraft and a reference to the client owner of the account.
- A *savings account* (compte d'èpargne) is represented by a unique account number, a balance, an interest rate and a reference to the owner of the customer account.
- numeric identifiers for clients (i.e client number) and accounts (i.e., current/savings account number) should be ensured to be assigned a unique value for each client and customer: how could you achieve this? (**hint**: consider using *class variables* i.e., `static` variables).
- a bank should be capable of dealing with a number of clients and both kinds of accounts (current accounts and saving accounts). Furthermore the user of a bank should be capable of adding customers as well as current and saving accounts to a bank (hint: think about equipping the bank class with methods `addCustomer(...)`, `addCurrentAccount(...)`, and `addSavingsAccount(...)`).
- For each class provide a *constructor* that initializes each attribute with given parameters. Write the constructor of class `Customer` using the keyword "this" and the constructors of all remaining classes without using this.
- For each class provide also a `toString()` method, that is, a method that returns a `String` object which contains informations about the current value of (a subset of) the class attribute, as sketched by the following example:

```
public String toString() {

String s = new String();
```

```
s = s + "attribute1 = " + this.attribute1 + "attribute2 = " + this.attribute2
    + ...
}
```

- for each class define the *getters* and *setters* method for each `private` attribute. As a reminder *getters* and *setters* are characterized as follows:
  - *getter*: a `public` method associated to a `private` attribute of a class and whose execution returns the value of the associated attribute
  - *setter*: a `public` method associated to a `private` attribute. It has one argument (whose type is the same as the type of the associated attribute) and its execution assigns the associated attribute with the value passed on as argument on invocation.

  Hint: to add constructor(s), getters/setters and `toString()` use the Eclipse "Source" generation facility: right-click on the Eclipse frame of the .java file of the class you working on and select the "Source" item in the pop-up menu, and from the "Source" sub-menu select the proper item e.g. "Generate getters/setters"

Testing. To test your program add a `main()` method to the Bank class in which you perform the following instructions:

- create two clients: Smith and Davies and add them to the bank.
- create a current account for Mr. Smith (account number 1) with a threshold of authorized overdraft -500 Euros. Add 1000 Euros to his account. Remove 100 Euros. Remove 2000 Euros.
- create a savings account for Mr. Smith (account number 1) with an interest rate of 3.
- create a current account to Mr. Davies (account number 2). Add 800 Euros.
- display the content of the bank: accounts and customers.

Exercise 2. Bank management with inheritance

Define an alternative solution to the *bank management* problem (Exercise 1) based on inheritance. In order to identify a proper hierarchy for the "Account" classes you should consider the following questions:

- A ..... IS-A ..... (for example "A Car IS-A Veihcle ")
- A ..... HAS-A ..... (for example "A Car HAS-AN Engine ")

Based on such scheme identify the superclasses and subclasses for the Bank accounts problem as well as the attributes that should be included in each superclass and subclass. In the Tutorial2 project add a new package `fr.ecp.is122o.tutorial2.ex1q2` in which you will add the classes corresponding to this exercise.

Exercise 3. Shapes

Add a package `fr.ecp.is1220.tutorial2.ex3` add a JAVA implementation of the class hierarchy depicted in the class-diagram of Figure 1. In particular consider the following characteristics for the superclass Shape and all of its derived classes.

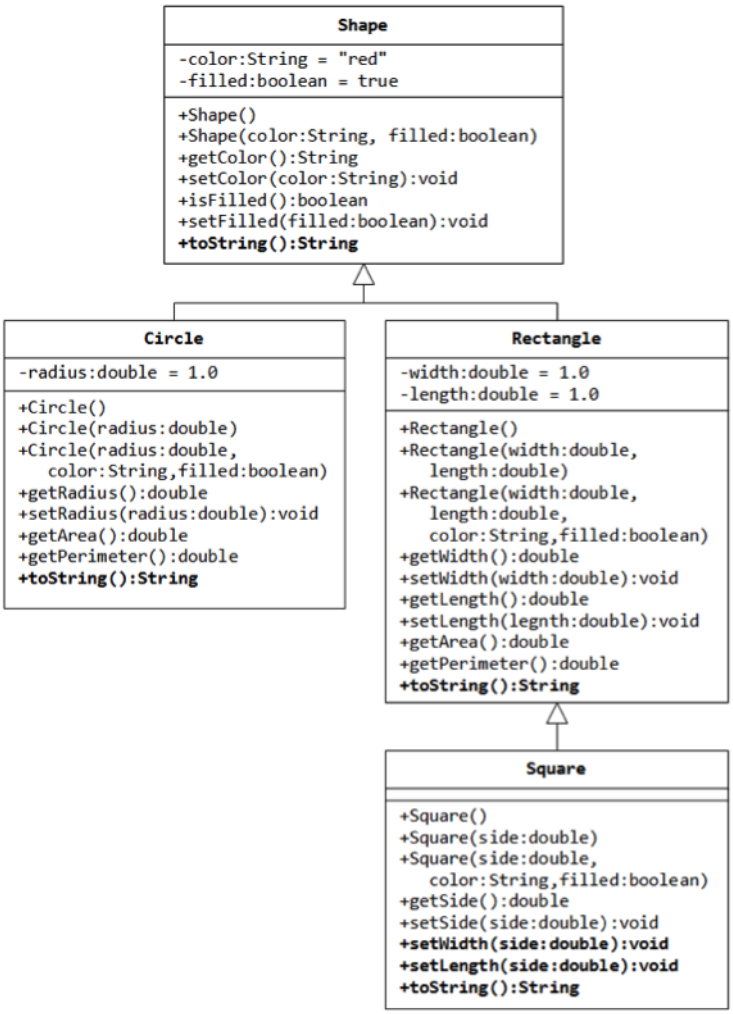

Figure 1: Class diagram for the Shape hierarchy

Shape (superclass):

- Has two (private) instance variables color (`String`) and filled (`boolean`)

- Defines two constructors: a no-argument constructor that initializes color to "green", and a constructor with two arguments which initializes both attributes.
- Define getters and setters for all attributes of Shape
- Define a `toString()` method for Shape which returns a shape with color "xxx" and filled/not-filled
- Write a test program to test the Shape class
- Write two subclasses os Shape, namely Circle and Rectangle.

Circle

- Instance variable radius (double)
- Three constructors (the no-arg constructor initializes radius to 1.0)
- Getters and setters for the radius
- Extend the Shape class with Write the `getArea()` and `getPerimeter()` methods for Circle
- Override the `toString()` method (inherited from `Shape`) so that it returns a circle with radius=xxx which is a subclass of "yyy" where yyy is the output of the `toString()` method from the superclass.

Rectangle

- Instance variables `length` (having `double` as type) and `width` (having `double` as type)
- Three constructors (the no-arg constructor initializes length and width to 1.0)
- Getters and setters for the width and length
- Extend the Shape class with Write the `getArea()` and `getPerimeter()` methods for Circle
- Override the `toString()` method (inherited from `Shape`) so that it returns a rectangle with length=xxx which is a subclass of "yyy" where yyy is the output of the `toString()` method from the superclass.
- Write a `Square` class which is a subclass of `Rectangle`
- Provide the appropriate constructors of Square

- Override the `toString()` method (inherited from Shape) so that it returns a square with side=xxx which is a subclass of "yyy" where yyy is the output of the `toString()` method from the superclass
- Do you need to override the `getArea()` and `getPerimeter()` methods? (try them out)
- Override the `setLength()` and `setWidth()` methods to set the length of the length and width so to maintain the square geometry.