Universidad Autónoma de Madrid

Escuela Politécnica Superior

Software Analysis and Design 2019-2020

Assignment 3: Introduction to Object Oriented Programming

Starting date:	Monday 24th of February.
Duration:	3 weeks.
Submission:	**1 hr before the start of the next assignment, via Moodle.**
Weight in the grade:	20 %

The goal of this assignment is to learn basic concepts of object oriented programming with Java, by incrementally developing the Java classes (with their testers and documentation) of a small application for sales management.

This assignment will be based on the following Java concepts:

- *Primitive data types, String, Array, and reference types (objects)* defined by the programmer.
- *Classes* defined by the programmer to implement abstract data types using *instance variables, class variables, instance methods, class methods, and constructors.*
- *Basic input/output* to read text files and display text information on the console.
- *Inheritance, overloaded methods*
- A first look at *collections*
- *Good programming style and clear documentation using the Javadoc tools.*

Part 0. Introduction

This assignment will go on developing the sales management application for a home appliances shop, whose design you started in **part 2 of previous assignment**. Initially, we will work from the description given in assignment 2, and then we will add new functional requirements.

Part 1. Object creation and basic calculations (3 points)

The tester shown below creates some home appliances and uses them to create some sales with the purpose of testing the calculation of the final price of each sale. In this part you ought to adapt your design (class diagram), and the code developed so far, so that they become compliant with that tester.

For example, you must use the same class names used in the tester, add a method `getTicket()` which may not have initially included in your design, etcetera. Also you will notice that refrigerators and washing machines (also called washers) can be created with optional information which was not mentioned in the previous assignment: in particular, refrigerators have an attribute indicating whether they are *no frost* and washers have two attributes indicating *kilograms of load* and *revolutions per minute for spin-dry*.

In addition to those attributes, you must implement all the constructors and methods required to make the execution of the tester produce the expected output shown below. The new attributes must use the *final* positions in the parameter list of the constructors, so that it is easier to distinguish when a constructor is call using all possible parameters (including the optional ones) from when it is called with no optional parameters. In the tester, you can see that the washer constructor has different number of parameters when called to create the *Indesit* washer and when call to create the *Superkin* washer.

Recall that all details for computing discounts, delivery costs and final price were given in Part 2 of previous assignment. Make sure that your computations produce exactly the same output and with the format show in the expected output below.

Follow the Java Programming Style Guide available in Moodle. Include comments, especially those in Javadoc format, in the code of the classes you program.

Tester 1 (available in Moodle):

```
package ads.assignment3;

/**
 * Tester for Part 1 of lab assignment 3
 * @author Course Profs
 */
public class Tester1 {
          public static void main(String[] args) {
            Appliance tv1 = new Television("Loewe", "Art 48 4K Plata", 1699.00,
                                        EnergyEfficiency.A, 48);
            Appliance tv2 = new Television("LG", "32LF592U", 271.92,
                                        EnergyEfficiency.C, 32);
            Appliance wash1 = new Washer("Indesit", "XWE 91483 X", 329.0,
                                        EnergyEfficiency.A, new Dimension(59.5, 60.5, 85.0), 72, 9.5, 1500);
            Appliance frigo1 = new Refrigerator("Bosch", "KGN39VW21", 599.0,
                                        EnergyEfficiency.A, new Dimension(60, 65, 201), 83.0, true);

            Sale[] sales = new Sale[6];
            sales[0] = new Sale(tv1);
            sales[1] = new HomeDeliverySale(tv1);
            sales[2] = new Sale(tv1,
                              new Washer("Superkin", "", 0.0, EnergyEfficiency.UNKNOWN,
                                         new Dimension(59.5, 60.5, 85.0), 100)); // no load nor RPM
            sales[3] = new HomeDeliverySale(wash1, frigo1);
            sales[4] = new Sale(tv2,
                              new Television("Telefunken", "", 0.0, EnergyEfficiency.D, 32));
            sales[5] = new HomeDeliverySale(wash1, tv2);
            for (Sale s : sales)
                System.out.print(s.getTicket());
          }
}
```

Expected output for Part 1:

```
-----------------------------------------
Sold product: Loewe Art 48 4K Plata, 1699.00 Euros
-----------------------------------------
Product price:          1699.00 Euros
Shipping discount:         0.00 Euros
TOTAL:                  1699.00 Euros
-----------------------------------------
Sold product: Loewe Art 48 4K Plata, 1699.00 Euros
-----------------------------------------
Product price:          1699.00 Euros
Shipping discount:         0.00 Euros
Shipping cost:            43.00 Euros
TOTAL:                  1742.00 Euros
-----------------------------------------
Sold product: Loewe Art 48 4K Plata, 1699.00 Euros
-----------------------------------------
Product price:          1699.00 Euros
Shipping discount:        10.00 Euros
TOTAL:                  1689.00 Euros
-----------------------------------------
Sold product: Indesit XWE 91483 X, 329.00 Euros
-----------------------------------------
Product price:           329.00 Euros
Shipping discount:        25.00 Euros
Shipping cost:            46.00 Euros
TOTAL:                   350.00 Euros
-----------------------------------------
Sold product: LG 32LF592U, 271.92 Euros
-----------------------------------------
Product price:           271.92 Euros
Shipping discount:        40.00 Euros
TOTAL:                   231.92 Euros
-----------------------------------------
Sold product: Indesit XWE 91483 X, 329.00 Euros
-----------------------------------------
Product price:           329.00 Euros
Shipping discount:        55.00 Euros
Shipping cost:            46.00 Euros
TOTAL:                   320.00 Euros
```

Part 2. Basic reading of text files (2 points)

In this part, you must develop the class `ApplianceReader` with a method `readAppliances` which reads line by line a text file that contains all data (including optional data) describing the home appliances to be used for testing, and stores them into a data structure returned by the method. As you can see in the following tester, the structure returned by `readAppliances` must be compatible with `List<Appliance>`, and so it can be for example `ArrayList<Appliance>`. Recall that to be able to use those structures you must include an **`import`** similar to the one included in the tester.

The tester also shows how the method `readAppliances()` expects one parameter with the name of the text file to be read, whose contents are display next.

Input text file for tester 2 (available in Moodle, file name `products.txt`):

```
Loewe=Art 48 4K Plata=1699=A=48
LG=32LF592U=271.92=C=32
Indesit=XWE 91483 X=329=A=59.5=60.5=85=72=9.5=1500
Bosch=KGN39VW21=599=A=60=65=201=83=NoFrost
Loewe=Art 48 4K Plata=1800=A=48
Indesit=XWE 91483 Z=329=A=59.5=60.5=85=72=10.0=1500
Bosch=KGN39VW21=599=A=59.5=60.5=85=72=10.0=1500
```

Each file line consists of a sequence of fields separated by an **=** sign (which we may safely assume will never appear within any field). Fields on any given line describe all the attributes of certain home appliance (refrigerator, washer or television), including the optional data that were added in Part 1.

After reading each line and splitting it into its fields, the method will create an instance of the type of home appliance corresponding to the information read, and will add it to the data structure to be returned after processing the whole text file. It will be necessary to auxiliary methods to create the different types of appliances, one method for each type. Think carefully about what is most convenient class to contain those methods. The execution of this tester must generate an output identical to the expected output shown after the tester

Tester 2 (available in Moodle)**:**

```
package ads.assignment3;
import java.util.List;
/**
 * Tester for Part 2 of lab assignment 3
 * Receives the file name through the command line
 * @param args[0] First and only command line argument must be the name of the text file describing products
 * @author Course Faculty
 */
public class Tester2 {
    public static void main(String[] args) {
        List<Appliance> products = ApplianceReader.readAppliances( args[0] );
        for (Appliance p : products) {
                System.out.println(">> " + p );
        }
    }
}
```

Expected output for Part 2:

```
>> Loewe Art 48 4K Plata, 1699.00 Euros
>> LG 32LF592U, 271.92 Euros
>> Indesit XWE 91483 X, 329.00 Euros
>> Bosch KGN39VW21, 599.00 Euros
>> Loewe Art 48 4K Plata, 1800.00 Euros
>> Indesit XWE 91483 Z, 329.00 Euros
>> Bosch KGN39VW21, 599.00 Euros
```

Part 3. Defining equality of appliances with `equals` to avoid duplicates (2 points)

In this part you will use the same tester used in Part 2, but the expected output will be different (as shown below). In particular, you will modify your previous class `ApplianceReader` so that when home appliances are read from the file, the method detects if the appliance just read is equal to any of the other appliances previously read and store. In case the list of previously read appliances already contains one equal to the one just read, you must avoid adding the latter and maintain the former in the list. **Two appliances are considered equal if and only if both are of the same type and have identical brand and model**, regardless of their other attributes. Although it may be unusual, it is possible that two appliances could be of different types (e.g., refrigerator and washer) while having the same brand and model. Thus, it is necessary to verify whether their types are identical (or not) before they can be granted (or not) as equal appliances.

Note that in the expected output the duplicated appliance does not have the same price as the other appliance with the same type, brand and model. Analogously, the appliance described in the last line of the file is taken to be different from the other one with same brand and model.

Java´s mechanism to define the equality of objects is based on the method `public boolean equals(Object obj)` which is predefined in the class `Object` and inherited by all classes. However, any class may override that method to provide a new implementation of how equality should be decided particularly for instances of that class. To that purpose you must include, in the proper class, your own implementation of a method with exactly the same header of `equals` (described above) and preceded by `@Override` to indicate that your definition of the method overrides a previous definition.

Input text file for Part 3:
Same as for Part 2

Tester 3:
Same as tester 2

Expected output for Part 3:
```
Duplicated not added:
Loewe Art 48 4K Plata, 1800.00 Euros
>> Loewe Art 48 4K Plata, 1699.00 Euros
>> LG 32LF592U, 271.92 Euros
>> Indesit XWE 91483 X, 329.00 Euros
>> Bosch KGN39VW21, 599.00 Euros
>> Indesit XWE 91483 Z, 329.00 Euros
>> Bosch KGN39VW21, 599.00 Euros
```

Part 4. Storage and deletion of sales within class `Sale` (2 points)

As you can see in the tester of this part, the first instruction executes the tester of Part 1 again. After that there are invocations to new methods that you will develop and add to class `Sale`. The goal of these methods is to manage globally the collection of all sales created during the execution of tester 1.

The first new method is the class method `salesSummary`, which creates a returns a string describing the sales summary to be printed (see the tester code and the expected output). That string will contain the following information, in chronological order: one line per sale (not cancelled) which corresponding to the final line of the sale ticket. This implies that at some point in time all sales will have been stored within the class `Sale`. *(Note: this may not always be the best design decision but for now we will use it).* If in part 1 you decomposed the method `getTicket` in four methods, one for each line of the ticket, then the current task will be very much simplified by reusing now the method you created to generate the final line of the ticket. Otherwise, you may have to repeat code (a bad practice).

The second method is the class method `cancel`, which removes from the collection of stored sales the last sale added to the collection, and returns the removed sale. If the method is invoked repeatedly, it will remove one by one several sales in the reverse order of their creation. If there are no more sales to remove, it will simply return **`null`**. Also, you will need to add a method that returns the last sale added to the collection, but without removing it. The tester uses these methods to cancel two sales and then verify that they no longer appear in the sales summary.

Finally, you will need to add two forms of the method `salesSummary`; the first with one numeric parameter which show only the sales with *final price not less than the parameter value*, and the second with one `String` parameter which will show only the sales of *products whose brand contains the text given* as parameter.

Tester 4 (available in Moodle)**:**

```
package ads.assignment3;

/**
 * Tester for Part 4 of lab assignment 3
 * @author Course Faculty
 */
public class Tester4 {
  public static void main(String[] args) {
        // first execute tester1 to load appliances and sales
        Tester1.main(null);

        System.out.println( Sale.salesSummary() );
        System.out.println( Sale.last().getTicket() );

        Sale cancelled = Sale.cancel();  // cancell the last sale
        System.out.println("Sale cancelled:\n" + cancelled.getTicket());

        Sale.cancel();  // cancel another sale

        System.out.println( Sale.salesSummary(500) ); // without last two sales nor the one with price < 500
        System.out.println( Sale.salesSummary("Indes") ); // only sales not cancelled and with brand Indesit

  }
}
```

Expected output for Part 4: We expect the same output from Part 1 followed by this output:

```
SALES SUMMARY
TOTAL:                  1699.00 Euros
TOTAL:                  1742.00 Euros
TOTAL:                  1689.00 Euros
TOTAL:                   350.00 Euros
TOTAL:                   231.92 Euros
TOTAL:                   320.00 Euros

-------------------------------------------
Sold product: Indesit XWE 91483 X, 329.00 Euros
-------------------------------------------
Product price:           329.00 Euros
Shipping discount:        55.00 Euros
Shipping cost:            46.00 Euros
TOTAL:                   320.00 Euros

Sale cancelled:
-------------------------------------------
Sold product: Indesit XWE 91483 X, 329.00 Euros
-------------------------------------------
Product price:           329.00 Euros
Shipping discount:        55.00 Euros
Shipping cost:            46.00 Euros
TOTAL:                   320.00 Euros

SALES SUMMARY
TOTAL:                  1699.00 Euros
TOTAL:                  1742.00 Euros
TOTAL:                  1689.00 Euros

SALES SUMMARY
TOTAL:                   350.00 Euros
```

Part 5. Extending the original design (1 point)

The strongest test for any software is to see how well it supports maintenance tasks, even when new requirements are added. That is why in this part you are to **add two new classes**: one to represent *sales to The Canary Islands*, another to represent *curved televisions*.

Sales to The Canary Islands do not accept that customers turn in an old used appliance when the sold one is delivered to their home, and the shipping cost is always calculated as a 7% of the base price of the sold product. For example, the sale of a TV Loewe Art 48 4K Plata, with a base price of 1699.00 Euros implies a shipping cost of 118.93 Euros.

For the curved televisions we must store, in addition to features common to all televisions, their weight and dimensions (analogously to dimensions of washers and refrigerators), since their shipping cost is computed by adding an extra 25 Euros per cubic meter to the shipping cost previously described for televisions.

Modify your previous code – improving it if necessary – to offer the new functionality, and add your own test data and tester code to test the new functionality. Of course, do not introduce any error or unexpected output in the previous testers.

While performing those modification, analyze how many of the changes you are making on your code could have been avoided had you designed your initial code better (more flexibly).

Part 6 (optional). (1 point)

Extend and modify the code and design of the Java classes generated at the end of part 5, in order to address new requirements regarding *stock management* for the home appliances shop.

In particular, you must implement the following functional requirements: add the description of a new appliance (refrigerator, washer or television) to the catalog of products in stock; maintain updated the amounts in stock (after sales are made or cancelled, or new appliance units are received in the store); discontinue offering sales of certain appliance by removing it from the catalog (and from the stock); and generate an inventory of the appliance stocks, which could include all appliances or only those of a selected.

In addition to develop and document the classes added in this optional part, you must create your own test data and tester code to test the newly added functionality. **If in this part you modify the classes you developed in parts 1 thru 5 so significantly that the testers used in those parts do not generate the expected output, then you must save your work before modifying it in this optional, and submit both code versions separately (see instructions below).**

How to submit:

- The submission should be packaged in one ZIP or RAR file, that has the following name: `GR<group_number>_<student_names>.zip`, For example, the group composed by Marisa and Pedro, of group 2261, should submit a file named: `GR2261_MarisaPedro.zip`
- Only ONE member for each team will submit ONE compressed file resulting from compressing a directory named **`P3_`**`GR<group_number>_<student_names>` (e.g., **`P3_`**`GR2261_MarisaPedro`) with the following contents:
    - directory **src** with all Java code in its **final version described in Part 5**, and in case you have solved the optional part, **the code version as described in Part 6**, including test data and all testers used or developed for each part
    - **doc** folder with the documentation generated using Javadoc
    - **txt** folder with all text files used or generated during testing
    - one PDF file with the **class diagram** and a brief rationale justifying the decisions you have taken in the course of this assignment, the problems that you faced and how you solve them, as well as any problems pending to be solved satisfactorily.