# Object-Oriented Programming 1 – Graded Exercises, Series 3 (DOJ)

**Exercise 1: Callbacks**

Unzip the given file `Countries.zip` and use the `CountryLoader.getCountries` method to load the file `WorldCountries.txt` containing information about world countries (their names, populations and areas) into an `ArrayList<Country>`. Then make some processing of that information as explained below, by using the callback technique.

- First, select the data you want to process: either the *populations* or the *areas.* Use for this the interface:

```
public interface ValueSelector {
  public int getValue(Country country);
}
```

  whose implementations (`PopulationSelector` or `AreaSelector`) return the data values you want to process. Also provide a filter to limit the number of values processed (e.g. process only countries with at least 1 million inhabitants or only countries with at least 100000 $km^2$). This filter should implement the interface:

```
public interface Filter {
  public boolean accept(Object obj);
}
```

  and be activated each time a data is selected.

- Then, select the operation you want to execute on the selected data: either computing the *sum* or the *average* of the data values. Use for this the interface:

```
public interface Operator {
  public long calculate(java.util.List<Country> countries,
                        ValueSelector selector);
}
```

  whose implementations (`Adder` or `AvgOperator`) carry out the operation you want to process and return the result.

- Now, process the country information according to your selections (e.g. calculate the sum of the world population or the average of all country areas that have at least 100000 $km^2$) and display the corresponding result. Also print a list of the processed countries, sorted according to the selected data values (use the `Collections.sort` method with an appropriate comparator) with their names, populations, areas and population densities.

To control your program, i.e. to make your selections and activate the processing, build a little GUI with 6 buttons:

- *Population* and *Area*, to select the data to process. (To enter a filter value, call a `TextInputDialog`).
- *Sum* and *Average,* to select the operation to execute
- *Execute,* to activate the processing
- *Print,* to print the list of the processed countries

**Exercise 2: A Company**

A company has four types of employees: the manager, the vendors, the trainees and the workers, which can be hourly-workers or salaried-workers. All employees have a name (a string), an entry date (a `LocalDate` object containing the date when the employee entered the company) and get paid at the end of month.

- The manager gets a salary that fully depends on the value of the company shares at pay-day (simulate this with a random value between a minimum and a maximum amount).
- The vendors get a fixed salary plus a commission of $200 on every item they sold during the month (for example, if a vendor sell 15 items, it will receive $3000 commission).
- The trainees are vendors, too, but they are not within the company since very long time. Therefore, they get a fixed salary but don't get the whole commission as a normal vendor. During the first year, they do not get any commission. After one year in the company, they receive only 20% ($40) thereof pro item sold, after two years, 40 % ($80) , ..., and after five years, 100% ($200).
- The workers do not have a fixed salary. They get paid the hourly wage for the actual number of hours worked. If an hourly-worker worked more than 170 hours a month, it gets the excess hours paid time and a half. A salaried-worker never gets excess hours more paid than normal ones, but after 15 years in the company, it gets a special prime of $100 each month.

Of course, the company has not only employees, but also computers, vehicles, buildings, etc., that we do not consider here. The important thing is that all these elements must be insured and therefore insurance premiums must regularly be paid. In particular, all employees, except trainees, have a health insurance and must pay each month an insurance premium. The basic monthly premium is $200 and it grows by $10 each year (i.e. after one year in the company, it is $210; after two years, it is $220, etc.). For workers however, the premium does never grow above $300. For all employees (except trainees), 50% of the insurance premium is withdrawn from the salary.

Build a class/interface hierarchy to describe this company and implement it. In your classes, implement constructors to initialize employee instances and appropriate methods to calculate how many the employees of each category earn each month. Write JUnit test classes to check these methods.

Then, write a main class that creates a few instances of every employee category and stores these instances in an array list. Read the list and print out, for every employee, the employee's category (i.e. the name of the employee's class), his name, the number of years he has been in the company, and the amount of money he actually gets at the end of the month (i.e. salary – insurancePremium / 2). To produce this output, use `toString` methods and define good readable formats.

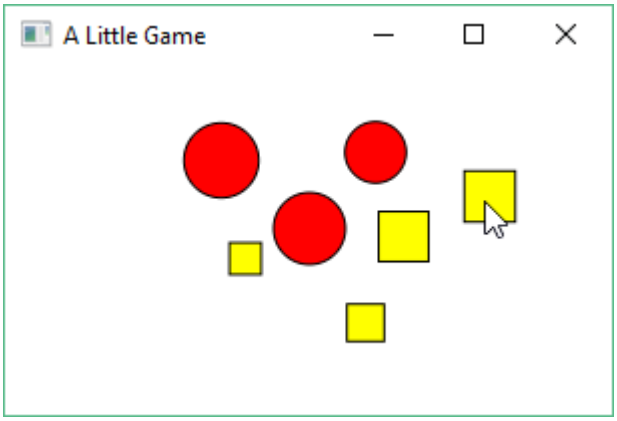
**Exercise 3: A Bank (Overriding Object Methods)**

For this exercise, use the bank account files that we developed during the lectures (you find them on Moodle stored in the file `Bank.zip`).

1. Extend the account classes as follows: Every bank account should receive a (unique) ID-number (an `int`), an owner (a mutable(!) `Person` object) and have the possibility to get an initial balance when the account is created. In one of the account types, define two constructors (one with an initial balance and one without).
2. Create a `Bank` class and register a few accounts in that bank.
3. Implement appropriate `toString, equals, hashCode` and `clone` methods in all your classes. Also write copy constructors.
4. Write JUnit test classes to verify that your methods (except `toString`) work correctly in all your classes. Note: two banks should be considered as equal, if they contain the same accounts, but not necessarily in the same order!
5. Write a small `Main` class that print all the accounts contained in a bank, using the `toString` methods.

**Exercise 4: A Little Animated Game**

The following scene contains moveable shapes (circles and squares) and builds a little game:




**Game Rules:**

- When the mouse key is pressed in the window, a shape is created, centred at the mouse pointer. The dimension of the shape (the circle radius or the square side length) is selected randomly between fixed minimum and maximum values.
- When the mouse key is released, the shape starts moving in a random direction at a random velocity.
- When the mouse key is pressed on an already existing (moving) shape, the shape stops and can be dragged by the mouse at any other position on the panel. As soon as the mouse key is released, however, the shape starts moving again by itself in a (new) random direction at a (new) random velocity.
- The goal of the game is to create as many shapes (circles and squares) as possible, avoiding however that a shape leaves the window, by dragging it back, each time it tries to "escape" from the window.
- If a shape succeeds to leave the window, the game terminates. You win, if no shape could escape after a given amount of time (e.g. one minute). Of course, your score should be better, if you have created many shapes instead of only one or two (imagine a way to calculate your score).

In your solution, you should use a suitable interface `MoveableShape` that is implemented by the shape classes (`Circle` and `Square`).

Enter the time duration for the game as a command line parameter and read it with the method `javafx.application.Application.getParameters`. Check the input and throw an exception if it is incorrect.