

Name: Anais Ulrichs
Module: 4WCM0022-0901-2019 - Programming (COM)
Module Leader: Mark Martin
Student ID: 17070779
Date of Submission: 20.12.2019

Practical Coursework: Revenue Statement of a Small Shop

The goal of the coursework was to create a Java Program, using Object Oriented Programming Techniques introduced throughout the module. This paper provides documentation for the program that was created. First, the high-level overview of the project will be outlined, before explaining in detail the design of the project, and of the individual classes and methods.

The Java program was written in BlueJ [1], a Java Development Environment, installed on Ubuntu. Additionally, a private GitHub repository was used for version control.

1. Program Description

The program allows a store manager to input inventory items into the system, whereby the distinction is made between three different categories of items. The first category is 'tools', which refers to any small utility item. The second is 'grocery', which refer to any food item; and the last one is 'stationery', which categorises any office item that the store might have in stock. Once all items are placed into the system, a report will be generated, stating the following:

Category (tools, grocery, stationery)	Name of the item	Number ordered	Number in shop	Number sold	Revenue Profit/Loss per item
--	---------------------	-------------------	-------------------	-------------	------------------------------------

Table 1: List of variables printed in the report

At the end of the report, an additional section will highlight the total revenue of each category and the total revenue of the shop for the given month.

When inputting the data for each item, the shop owner will be asked to provide the following information:

- The name of the inventory (tools, grocery, stationery); stored in variable *inventoryType*.
- The name of the item purchased; stored in variable *itemName*.
- The number of items ordered; stored in variable *numOrdered*.
- The number of items in the shop; stored in variable *numInShop*.
- The order cost of the individual item; stored in variable *orderCost*.
- The shop price of the individual item; stored in variable *priceSoldAt*.
- The total number of items sold this month; stored in variable *numSold*.

2. Classes

The program consists of three classes:

1. The *Report* class: Calls the methods that are responsible for creating the inventory, generating the revenue for each item, each category of item and the overall revenue of the shop, printing the report and saving the report to a txt file.
2. The *Inventory* class: The *Inventory* class creates an object of type *Inventory* when called. It has stored all the variables that define the class *Inventory*, as well as the methods that allow

the user to create a new instance of the class *Inventory*. When the *Inventory* class is called, it creates a new instance of the class; depending on the user input, the instance will be in a particular state. The state hereby refers to all attributed to the class *Inventory*, which is stored in the variables. [2]

3. The *CalculateRevenue* class: This class will take the variables of each *Inventory* class and calculate the revenue for the individual categories and the total revenue for the shop.

Additionally, the program contains a test class, called *ReportTest*. Once compiled, the user can call the *InventoryTestOne()* method to execute the entire program.

The following sections will walk through the separate classes in the program.

2.1 ReportTest Class

The *ReportTest* Class is used to execute the program. [3] To do so, the program has to be compiled and the *InventoryTestOne()* method called. First, it will create a new instance of the *Report* class, to then call several methods from the *Report* class. The methods are called in the following order:

```
Report inventoryReportOne = new Report(); // creates a new Report instance for the test
inventoryReportOne.createInventory(); // calls the createInventory() method of the new Report
inventoryReportOne.calculateRevenue(); // calls the calculateRevenue() method of the new Report
inventoryReportOne.calculateTotalRevenue(); // calls the calculateTotalRevenue() method of the new Report
inventoryReportOne.printInventory(); // calls the printInventory() method of the new Report
inventoryReportOne.saveReport(); // calls the saveReport() method of the new Report
```

Figure 1: Methods called in the *InventoryTestOne()* method of the *ReportTest* class

The next section will look at the *Report* class to explain each individual method.

2.2 Report class

The *Report* class contains the *main()* method of the program. In Java, the *public static main()* method is used by default to execute a program.

```
public static void main(String args[]) throws IOException
{
    Report report = new Report(); // creates a new Report instance

    report.createInventory(); // calls the createInventory() method of the new Report
    report.calculateRevenue(); // calls the calculateRevenue() method of the new Report
    report.calculateTotalRevenue(); // calls the calculateTotalRevenue() method of the new Report
    report.printInventory(); // calls the printInventory() method of the new Report
    report.saveReport(); // calls the saveReport() method of the new Report
}
```

Figure 2: *main()* method in the *Report* class

The *main* method applies a similar execution logic to the *ReportTest* class in that it first creates a new instance of the class *Report*. Each following method is then called of this instance with *report.method*.

Taking a step back, each *Report* Method will have an instance of the following variable, which are declared before the *main()* method. (Figure 3)

```

int inventorySize = 1; // variable to store the array size; the number of inventory items
Inventory inventory[] = new Inventory[inventorySize]; // new inventory array of type Inventory
double revenues[] = new double[inventorySize]; //new revenue array

/*
 * List of new RevenueCalculator objects set per category of inventory items
 */
RevenueCalculator tools = new RevenueCalculator();
RevenueCalculator stationery = new RevenueCalculator();
RevenueCalculator grocery = new RevenueCalculator();

/*
 * Each Report will create the following variables of datatype double since they store the revenue
 */
private double totalToolRevenue = 0;
private double totalStationeryRevenue = 0;
private double totalGroceryRevenue = 0;
private double totalRevenue = 0;

```

Figure 3: Instance variables of the *Report* class

The decision was made to have two separate arrays; one array to hold all *Inventory* objects and the other array to hold the revenues per *Inventory*. The size of the arrays is defined through the variable *inventorySize* of datatype *int*. Changing *inventorySize* allows to increase or decrease the number of items stored in both arrays.

After declaring the arrays, a new instance of the *RevenueCalculator* Class is created for each category. The state of each instance is stored in the respective variable of type *RevenueCalculator*. Lastly, each report will create several variables, which are unique to the *Report* and can only be accessed and changed within the *Report* class.

The *main()* method (Figure 2) will first call the *createInventory()* method. As the name suggests, this method is used to create the *Inventory*.

```

public void createInventory() {
    for(int i = 0; i < inventory.length; i++) { // iterate through the length of the array starting at 0
        inventory[i] = new Inventory(); // for every i in inventory[i] create a new Inventory object
    }
}

```

Figure 4: *createInventory()* method of the *Report* class

The *createInventory()* method (Figure 4) contains a for loop that iterates through the length of the *inventory* array. For each place in the *inventory* array, it will create a new object of the type *Inventory*. The *Inventory* class is explained further in section 2.3.

The next method that is called by *main()* is *calculateRevenue()*. This method will iterate through the newly created array of *Inventories* and check whether the category recorded for the given *Inventory* item equals either tools, grocery, or stationery. To do so, the category is stored in the type variable and then compared to the value of each category. In case it does match either of the categories, it will calculate the given revenue of that *Inventory* item and store the result at the same place of the *revenue* array as the *Inventory* item is stored in the *inventory* array.

```

public void calculateRevenue() {
    for(int i = 0; i < inventory.length; i++) { // iterate through the length of the array starting at 0
        String type = inventory[i].inventoryType; // type is the itemType per Inventory object in inventory[]
        Inventory inventoryItem = inventory[i]; // inventoryItem is the current inventory object

        if(type.equals("tools")) { // identify the category of type
            double revenue = this.tools.getRevenuePerItem(inventoryItem); // set revenue to the calculated revenue of category tools
            revenues[i] = revenue; // set the calculated revenue to the current place in revenue[]
        } else if(type.equals("stationery")) { // identify the category of type
            double revenue = this.stationery.getRevenuePerItem(inventoryItem);
            revenues[i] = revenue; // set the calculated revenue to the current place in revenue[]
        } else if(type.equals("grocery")) { // identify the category of type
            double revenue = this.grocery.getRevenuePerItem(inventoryItem);
            revenues[i] = revenue; // set the calculated revenue to the current place in revenue[]
        }
    }
}

```

Figure 5: *calculateRevenue()* method of the Report class

After the *createInventory()* and *calculateRevenue()* method, both the *inventory* array and the *revenues* array is populated with data. Based on the data recorded, the total revenue will be calculated in the *calculateTotalRevenue()* class.

```

public void calculateTotalRevenue() {
    this.totalToolRevenue = this.tools.getTotalRevenue(); // calculates the total revenue through the getTotalRevenue() class
    this.totalStationeryRevenue = this.stationery.getTotalRevenue();
    this.totalGroceryRevenue = this.grocery.getTotalRevenue();
    this.totalRevenue = this.totalToolRevenue + this.totalStationeryRevenue + this.totalGroceryRevenue; // adds up all revenues
}

```

Figure 6: *calculateTotalRevenue()* method of the Report class

Every time the '*getTotalRevenue()*' method is called, it is called from the instance of the *RevenueCalculator* class, which state is stored in tools, grocery, and stationery variable of the type *RevenueCalculator*. The total calculated revenue is stored in the *total Revenue* variables created at the beginning of the class. (Figure 3)

After the total revenue is calculated, the program has all the data to print the desired report on the screen and into a txt file. The *main()* method in the Report class will now call the *printInventory()* method. (Figure 8)

```

public void printInventory() {
    for(int i = 0; i < inventory.length; i++) { // iterate through the length of the array starting at 0
        System.out.print(inventory[i].inventoryType.toUpperCase() + "\t");
        System.out.print(inventory[i].itemName + "\t");
        System.out.print(inventory[i].numOrdered + "\t");
        System.out.print(inventory[i].numInShop + "\t");
        System.out.print(inventory[i].numSold + "\t" );
        if (revenues[i] > 0){ // check whether the revenue per item is bigger than 0 print the following
            System.out.print("£" + revenues[i] + " profit \n" );
        } else{ // else, the revenue is below 0 print the following
            System.out.print("£" + revenues[i] + " loss \n" );
        }
    }
    System.out.print("\n" ); // Next line to show clear division between reports
    System.out.println("TOTAL TOOLS: " + "\t \t \t \t £" + this.totalToolRevenue); // Print totalToolRevenue
    System.out.println("TOTAL STATIONERY: " + "\t \t \t \t £" + this.totalStationeryRevenue); // print totalStationeryRevenue
    System.out.println("TOTAL GROCERY: " + "\t \t \t \t £" + this.totalGroceryRevenue); // print totalGrocery Revenue
    System.out.println("TOTAL REVENUE: " + "\t \t \t \t £" + this.totalRevenue); //print totalRevenue
}

```

Figure 8: *printInventory()* method of the Report class

This method will iterate again through the array of *inventory* items. For each Inventory Object, it will print the variables as specified in Table 1. The last part is to record the revenue of each individual item and state the loss or profit made for that item. An if statement checks whether the revenue for the item is above 0, in which case, it will post a 'profit' after the revenue or if it is below 0, in which case, it

will post a 'loss' statement after the revenue. After it has printed all *inventory* items, it will post a line break to format the report. Below the report, the summary of the total revenue per category and the total revenue for the shop is posted.

Last, the `main()` method will call the `saveReport()` method. (Figure 9)

```
public void saveReport() throws FileNotFoundException { // throws FileNotFoundException in case the file is not found
    PrintWriter pw = new PrintWriter("RevenueReport.txt"); // create new PrintWriter
    for(int i = 0; i < inventory.length; i++) { // iterate through the length of the array starting at 0
        pw.print(inventory[i].inventoryType.toUpperCase() + "\t");
        pw.print(inventory[i].itemName + "\t");
        pw.print(inventory[i].numOrdered + "\t");
        pw.print(inventory[i].numInShop + "\t");
        pw.print(inventory[i].numSold + "\t" );
        if (revenues[i] > 0){ // check whether the revenue per item is bigger than 0 print the following
            pw.print("£" + revenues[i] + " profit \n" );
        }else{ // else, the revenue is below 0 print the following
            pw.print("£" + revenues[i] + " loss \n" );
        }
    }

    pw.print("\n"); // Next line to show clear division between reports
    pw.print("TOTAL TOOLS: " + "\t \t \t \t £" + this.totalToolRevenue + " \n");
    pw.print("TOTAL STATIONERY: " + "\t \t \t \t £" + this.totalStationeryRevenue + " \n");
    pw.print("TOTAL GROCERY: " + "\t \t \t \t £" + this.totalGroceryRevenue + " \n");
    pw.print("TOTAL REVENUE: " + "\t \t \t \t £" + this.totalRevenue + " \n");

    pw.flush(); // ensures that all data has been written to the stream of output into the file
    pw.close(); // closes the output stream
}
```

Figure 9: `saveReport()` method of the *Report* class

This method creates a new object of type *PrintWriter*, which is responsible to save all variables that have been printed to the screen in the previous method, into a txt file. The structure of the `saveReport()` method (Figure 9) is similar to the `printInventory()` method (Figure 8). Ultimately, the trade-off was made between separation of concern between each method and having an efficient program. Both methods could have been written as one but that would have resulted in one method being responsible for two tasks, writing the data into the console and printing the data into a txt file.

2.3 Inventory Class

The *Inventory* class is called in the `createInventory()` method in the *Report* class (Figure 4). Additionally, the array *inventory* in the *Report* class is of type *Inventory* since it stores all created *Inventory* objects. A new object of type *Inventory* is created for every place in the *inventory* array; '`new Inventory()`' is called in the `createInventory()` method, which then creates a new *Inventory()* object. The decision was made to have one object of type *Inventory* that is called, every time that a new *Inventory* item is added to the *inventory* array. All categories, tools, grocery, and stationery share the same variables. Thus, it was unnecessary to create separate objects per category. Every *Inventory* Object has a constructor that creates the following variables: (Figure 10)

```
public Inventory()
{
    inventoryType = getInventoryType();
    itemName = getItemName();
    numOrdered = getNumOrdered();
    numInShop = getNumInShop();
    orderCost = getOrderCost();
    priceSoldAt = getPriceSoldAt();
    numSold = getNumSold();
}
```

Figure 10: *Inventory* constructor of the *Inventory* class

Every variable is public. This allows them to be accessed by the *Report* and the *ReportCalculator* class. To create the value of each variable a method is called that allows the user to write the value into the console. Since all methods in the class *Inventory()* are fairly similar, this report will focus on three of them.

```
private String getInventoryType() { //method returns a variable of the datatype String
    System.out.println("Please enter the name of the inventory (tools, grocery, stationery): ");
    inventoryType = scan.nextLine();
    while (!inventoryType.equals("tools") && !inventoryType.equals("stationery") && !inventoryType.equals("grocery")){ // validates the user input
        System.out.println("Invalid input! Please enter the name of the inventory (tools, grocery, stationery): ");
        inventoryType = scan.nextLine();
    }
    return inventoryType; // return the newly created inventoryType
}
```

Figure 11: *getInventoryType()* method of the *Inventory* class

The first one is the *getInventoryType()* method (Figure 11), which asks the user to record the type of the Inventory item (tools, stationery, or grocery). A while method is used to check whether the input equals either of the categories. If it does not, it will post an error message to the user to request the correct input. (Figure 12) Once the input is recorded, the value is returned and can then be saved by the constructor into the *inventoryType* variable of datatype String.

```
Please enter the name of the inventory (tools, grocery, stationery):
wrong category
Invalid input! Please enter the name of the inventory (tools, grocery, stationery):
tools
```

Figure 12: Console user input and error message for the wrong input

```
private String getItemName() {
    System.out.println("Please enter the name of the item purchased: "); // print request to the user
    String itemName = scan.nextLine(); // Read user input
    return itemName; // returns the newly created itemName
}
```

Figure 13: *getItemName()* method of the *Inventory* class

The *getItemName()* method (Figure 13) allows the user to input any value which they see fit. Depending on the shop, shop items are not always identified via a name (String) but often a number (int). Thus, this method does not verify the content provided. Instead, all input is saved to a String. In future iterations of this program, it would make sense to define the input and the length of the input further to prevent the user from inputting data that cannot be processed.

```
private int getNumOrdered() {
    try { // try to receive the right user input; otherwise, terminates the program
        System.out.println("Please enter the number of items ordered: "); // print request to the user
        int numOrdered = scan.nextInt(); // Read user input
        return numOrdered;
    } catch (InputMismatchException e) { // print exception in the case of invalid input
        System.out.println("Invalid input! Please enter the number of items ordered.");
    }
    return numOrdered;
}
```

Figure 14: *getNumOrdered()* method of the *Inventory* class

The following methods vary in the variable name and in the datatype of the variable. In particular, the datatype int has been used for inventory size variables, such as the number of items ordered, and double has been used for any variable that refers to the price of an item. The input is validated through a try and catch statement (Figure 14). If an input is provided, other than what type int can hold, the program will terminate and the error messages will be printed to the screen. The error

messages provide user-readable information that is easier to understand than the default error messages, which are logged into the terminal when the program terminates.

2.4 RevenueCalculator Class

The *RevenueCalculator* class is used to separate the logic of calculating the *revenue* per Inventory item from the *Report* class. Additionally, it would have not been practical to place the calculation logic into the Inventory object, since the data for an Inventory object is populated when a new Inventory object is created. Before all Inventory Objects are created, the total *revenue* per category cannot be calculated.

The first time the *RevenueCalculator* class is used when a new instance of the object is created in the beginning of the *Report* class. The Report class uses in total three instances, one per category. Once a new instance is created, the constructor in the *RevenueCalculator* class (Figure 15) creates a new list of variables for that instance.

```
public RevenueCalculator() {  
    this.totalRevenue = 0;  
    this.revenuePerItem = 0;  
    this.totalOrderCost = 0;  
    this.orderCostPerItem = 0;  
    this.salePerItem = 0;  
    this.totalSale = 0;  
    this.revenue = 0;  
}
```

Figure 15: *RevenueCalculator* constructor of the *RevenueCalculator* class

The *RevenueCalculator* class consists of four methods. The *calculateRevenue()* method (Figure 5) in the *Report* class calls the *getRevenuePerItem()* method in the *RevenueCalculator* class and passes as parameter the current inventory item from the *inventory* array.

```
public double getRevenuePerItem(Inventory inventoryItem) {  
    setRevenuePerItem(inventoryItem); // call method setRevenuePerItem() with the values of the current Inventory item  
    return this.revenuePerItem;  
}
```

Figure 16: *getRevenuePerItem()* method of the *RevenueCalculator* class

The *getRevenuePerItem()* method (Figure 16) will call the *setRevenuePerItem()* method (Figure 17) in the same class. The latter will then return the *revenuePerItem* for this instance of the *RevenueCalculator* class. The *setRevenuePerItem()* method is a private method since it can only be called from the corresponding getter method of the same class.

```
private double setRevenuePerItem(Inventory inventoryItem) {  
    System.out.println(inventoryItem.numSold);  
    orderCostPerItem = inventoryItem.numOrdered * inventoryItem.orderCost; // multiply individual order cost with the number of items ordered  
    salePerItem = inventoryItem.numSold * inventoryItem.priceSoldAt; // multiply the number of items sold with the price the item is sold at  
  
    totalOrderCost = totalOrderCost + orderCostPerItem; // calculates the totalOrderCost for the category  
    revenuePerItem = salePerItem - orderCostPerItem; // calculate the revenue per item  
  
    revenuePerItem = Double.parseDouble(df2.format(revenuePerItem)); // formats the revenue to 2 decimal numbers  
  
    return revenuePerItem;  
}
```

Figure 17: *setRevenuePerItem()* method of the *RevenueCalculator* class

Furthermore, the *RevenueCalculator* class deals with calculating the total revenue of each category. The *calculateTotalRevenue()* method in the *Report* class (Figure 6) calls the *getTotalRevenue()* method (Figure 18) in the *RevenueCalculator* class. *getTotalRevenue()* calls *setTotalRevenue()*

(Figure 19) in the *RevenueCalculator* class, which is a private method since it can only be called and modify variables in the *RevenueCalculator* class.

```
public double getTotalRevenue() {
    setTotalRevenue(); // call method setTotalRevenue()
    return totalRevenue;
}
```

Figure 18: *getTotalRevenue()* method of the *RevenueCalculator* class

```
private double setTotalRevenue() {
    totalSale = totalSale + salePerItem; // calculates the totalSale of the category
    totalRevenue = totalSale - totalOrderCost; // calculates the Revenue/profit made from this category

    totalRevenue = Double.parseDouble(df2.format(totalRevenue)); // formats the revenue to 2 decimal numbers
    return totalRevenue;
}
```

Figure 19: *setTotalRevenue()* method of the *RevenueCalculator* class

3. User Interaction

This section will outline the user flow from creating the Inventories to receiving the report.

First, the classes have to be compiled. This can either be done by compiling the *ReportTest* class and then selecting the *InventoryTestOne()* method or by compiling the *Report* class and then selecting the *main()* method. Next, the terminal opens and the user will be asked to provide their input.

In this case, the *inventorySize* array in the *Report* class is set to equal '3' to allow for three Inventory objects to be added to the *inventory* array, and three revenues recorded in the *revenues* array. The screenshot below provides the example of adding one of those Inventory objects.

```
Please enter the name of the inventory (tools, grocery, stationery):
grocery
Please enter the name of the item purchased:
soy milk
Please enter the number of items ordered:
40
Please enter the number of items in the shop:
0
Please enter the order cost of the individual item:
0.8
Please enter the shop price of the individual item:
1.99
Please enter the number sold this month:
40
```

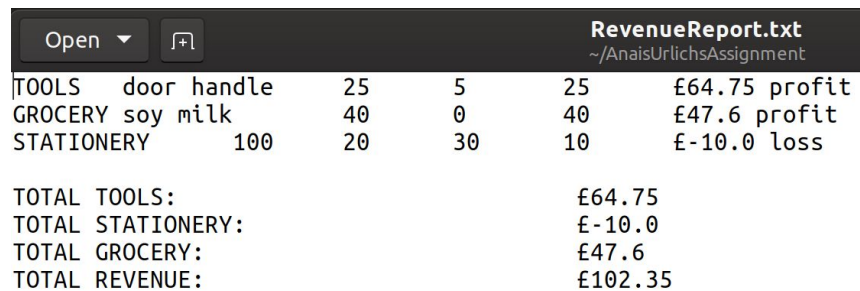
Figure 20: *setTotalRevenue()* method of the *RevenueCalculator* class

Once all three inventory items have been added to the *inventory* array (Figure 3), the following report will be printed to the console through the *printInventory()* method in the *Report* class (Figure 8):

TOOLS	door handle	25	5	25	£64.75 profit
GROCERY	soy milk	40	0	40	£47.6 profit
STATIONERY	100	20	30	10	£-10.0 loss
TOTAL TOOLS:					£64.75
TOTAL STATIONERY:					£-10.0
TOTAL GROCERY:					£47.6
TOTAL REVENUE:					£102.35

Figure 21: Inventory Report printed to the console

The same report is saved in the file 'RevenueReport.txt' through the *saveRevenue()* method in the Report class. (Figure 22)



The screenshot shows a text editor window titled 'RevenueReport.txt' with the path '~/.AnaisUrlichsAssignment'. The file contains an inventory report with the following data:

TOOLS	door handle	25	5	25	£64.75 profit
GROCERY	soy milk	40	0	40	£47.6 profit
STATIONERY	100	20	30	10	£-10.0 loss
TOTAL TOOLS:					£64.75
TOTAL STATIONERY:					£-10.0
TOTAL GROCERY:					£47.6
TOTAL REVENUE:					£102.35

Figure 22: Inventory Report saved in txt file 'RevenueReport.txt'

4. Proposed changes for future iterations of the program

This section outlines several design considerations that should be made in further iterations of the program.

- Iterating several times through the same array as done in Figure 4, Figure 5 and Figure 7 may create a computational overhead in the long term, depending on the size of the array. Thus, the goal will be to change the code and find an alternative way to only have to iterate once through the array.
- Additional iterations of the program will look into data verification to prevent faulty data, such as invalid user input, to be saved in variables.
- More comments have been added to the code than necessary. The goal of a good design is that the naming of methods, classes and variables is self-explanatory. Thus, several comments are unnecessary for understanding the program.
- Once the program becomes more advanced, for example by storing additional variables and methods to each inventory category, the inventory class should become an interface. The interface will define the common attributes of all categories. In the separate object classes, each category implements the Inventory class and may have its own variables and methods, which are specific to that category.

This assignment outlined one solution of creating Inventory objects, storing those to an array, calculating the revenue per object, category and for the shop, printing a report to the console and saving the same data into a txt file. The report looked into each class and outlined the connections between classes, as well as the reasoning for the given design. Additionally, the console output have been included to highlight some of the code features. Lastly, the drawbacks of the current implementation and thereof resulting considerations for further iterations of the program have been outlined.

Resources:

- [1] Bluej.org. (2019). *Bluej*. [online] Available at: <https://www.bluej.org/> [Accessed 3 Dec. 2019].
- [2] GeeksforGeeks. (2019). *Classes and Objects in Java - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/classes-objects-java/> [Accessed 5 Dec. 2019].

- [3] Bluej.org. (2019). [online] Available at: <https://www.bluej.org/tutorial/testing-tutorial.pdf> [Accessed 10 Dec. 2019].
- [4] Csis.pace.edu. (2019). *The Java Main Method*. [online] Available at: <https://csis.pace.edu/~bergin/KarelJava2ed/ch2/javamain.html> [Accessed 8 Dec. 2019].
- [5] Docs.oracle.com. (2019). *PrintWriter (Java Platform SE 7)*. [online] Available at: <https://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html> [Accessed 15 Dec. 2019].
- [6] B. Downey, A. and Mayfield, C. (2019). *Think Java*. [online] Greenteapress.com. Available at: <http://greenteapress.com/thinkjava6/html/index.html> [Accessed 1 Dec. 2019].