

Final Project Report

APRIL 7

Stock Control System

Authored by: Farhaan Beeharry – M00681483



Table of Contents


STOCK CONTROL SYSTEM	3
WHAT IS IT?	3
CLASSES & METHODS	3
DESCRIPTION OF THE FEATURES OF THE APPLICATION	8
CONCEPTS/THEORIES IMPLEMENTED	9
➤ OOP (Object Oriented Programming)	
➤ Files	
➤ Classes	
➤ Methods	
➤ Validation Checks	
CONCEPTS IMPLEMENTED (WITHOUT LECTURER'S HELP)	10
➤ Give the application an icon	
➤ Give buttons an icon	
➤ Use of CSS (Cascading Style Sheet)	
➤ Read the screen resolution	
➤ Regular Expressions (RegEx)	
PROBLEMS ENCOUNTERED AND SOLUTION(S) FOUND	13
HOW WAS THE PROJECT CARRIED OUT?	15
REFERENCES	15


Stock Control System

What is it?

It is a simple, user friendly desktop application designed with a graphical user interface to control the stock quantity of an item. Stock quantity can be increased or decreased, new items can be added, all data can be shown in a well-organized table.

Classes & Methods

-  **AddItem** – it is a class which process the addition of a new item from data input to the inventory/stock and writes it to the file
 - ❖ **addItem** – it is a GUI which allows the user to input data
 - ❖ **processAdd** – it takes the data, validates it and send it to `DataExport.addItemToFile` which writes it to the file

-  **ChangeStock** – it is a class which allows the user to change the stock quantity of a particular item and save the data to the file
 - ❖ **changeStock** – it is a GUI which asks the user to input the item code of the item for which he/she wants to change the stock
 - ❖ **changeStockConfirm** – it is another GUI which displays data of the chosen item code and asks the user for the new stock quantity. At this point, the user has the choice to cancel the process
 - ❖ **process** – it takes the new quantity input from the user and overwrites the existing quantity for the selected item in the array then uses `DataExport.updateFile` to write it to the file

✚ **DataExport** – It is a class which receives data from other methods in forms of variables or array of objects and writes/overwrites it to the file

- ❖ **addItemToFile** – it just contains an algorithm which receives data from other method(s) in the form of parameters (String/integer/double) and writes it on a new line in the file
- ❖ **updateFile** – it is a method which receives data in the form of an array of objects. It then clears the file and writes the array of objects to the file. Thus, any change of data will appear in the file

✚ **DataImport** – It is a class that reads the inventory.txt and/or the default.txt file and process it accordingly upon the request of other methods.

- ❖ **count** – it is a method which counts the number of items in the inventory.txt file
- ❖ **countDefault** – it is a method which counts the number of items in the default.txt file
- ❖ **importDefaultInventory** – it is a method which reads the data from the default.txt file and sends it to other methods in the form of an array so that they can use the data accordingly
- ❖ **importInventory**– it is a method which reads the data from the inventory.txt file and sends it to other methods in the form of an array so that they can use the data accordingly

✚ **Exit** – it is a class which has only a simple GUI which prompts the user that the application is about to exit and asks the user's confirmation

- ❖ **exit** – is contains the GUI and a yes and a no button for the user to confirm about exiting the application

✚ **Items (Class of objects)** – it is a class of objects where the constructors, setters and getters are found

- ❖ **Default constructor** – it is a constructor without parameters used to initialize the instance variables.
- ❖ **Explicit constructor** – it is a constructor with parameters sent by the user to assign to the object's variables
- ❖ **Setters** – this is used to be able to change and set data to the objects
- ❖ **Getters** – this is used to be able to read the data of objects

✚ **MessageBox** – is it a simple GUI which informs the user about errors, success and validation checks

- ❖ **box** – it contains a simple GUI which reads a message passed to this particular method as parameters and displays it with an 'OK' button to inform the user about actions

✚ **RemoveItem** – it is a class which allows the user to remove an item using its item code and overwrites the data to the file

- ❖ **removeItem** – it is a GUI which asks the user to input the item code of the item he/she wants to remove from the inventory
- ❖ **removeConfirmation** – this is another GUI which displays the data of the item the user chose and asks for confirmation from the user whether he/she really wants to remove the item
- ❖ **remove** – this method takes the item code and search it in the inventory, then uses the index of the item in the array and send it to the `removeTheElement` method which will process the deletion of the item from the array and returns the updated array. Then send the array to `DataExport.updateFile` to save the change(s)
- ❖ **removeTheElement** – this method creates a temporary array and copies all data in it except the data for the item to be removed. Thus, resulting in a similar array without the item which the user removed

✚ **RestoreDefault** – it is a class which replaces the data in the inventory.txt with the data in the default.txt if the user confirms the action

- ❖ **Restore** – it is a simple GUI which asks the user to choose YES or NO whether he/she wants to restore the default inventory
- ❖ **restoreDefault** – If the user chooses YES, this method reads the default.txt into an array and send it to DataExport.updateFile to replace the inventory with the default data

✚ **Search** – it is a class which allows the user to search for the item he/she wants using either item code or item name or item price (can show more than one result)

- ❖ **choice** – This is a GUI which asks the user to choose from radio buttons whether he/she wants to search by item code or by item name or by item price. Then asks the user to input the search criteria.
- ❖ **itemCode** – it is an algorithm which uses the user input to compare with the item code data in the file/array and if the data corresponds, it opens Search.result else it displays “No match found” message.
- ❖ **itemName** – it is an algorithm which uses the user input to compare with the item name data in the file/array and if the data corresponds, it opens Search.result else it displays “No match found” message.
- ❖ **itemPrice** – this method displays result in a different way, it uses a table approach. For each item price which matches the user’s criteria, it adds the item to the table and displays it to the user, that is it can have more than one result
- ❖ **result** – this is a GUI which displays the search result item and data, if any, when the user searches by item code or by item name.

✚ **SimilarityCheck** – this class contains simple algorithms which checks if the data input by the user already exists or not. Which means that similar item codes cannot exist

- ❖ **itemCode** – this method checks the user input item code with the item codes already in the file. If there is a similarity, the application displays an error to the user.
- ❖ **itemName** – this method checks the user input item name with the item names already in the file. If there is a similarity, the application displays an error to the user.

✚ **Sort** – this class contains algorithms which compares the data of the file with each other and sort them accordingly using the bubble sorting method

- ❖ **itemCode** – contains algorithms which compares item code with each other. If it is not in the correct order, it swaps the data. Else compares the next two items
- ❖ **itemName** – contains algorithms which compares item name with each other. If it is not in the correct order, it swaps the data. Else compares the next two items
- ❖ **itemPrice** – This one is a multi-level sorting algorithm. Instead of taking data from the file directly, it takes the data from the Sort.itemName method. Which means that when it receives the data, it is already sorted by name alphabetically in ascending order. Then it sorts the data according to the price. Therefore, if there are 3 similar items with the same price, the 3 items will be sorted by item name alphabetically in ascending order in the table

-
- ✚ **StockControlSystemGUI** – this class contains the main method which launches the application. It has the main menu GUI which allows the user to choose what he/she wants to do next
 - ❖ **main** - This method calls the start function to display the main menu GUI
 - ❖ **start** – this contains the GUI of the main menu to allow the user to operate with the different functions of the application such as add, remove, etc....

- ✚ **ViewAllItems** – this class allows the user to view all the data in the inventory.txt file in the form of a table. It also allows the user to choose how he/she wants the data to be sorted in the table
 - ❖ **choice** – This is a GUI which asks the user how does he/she wants the data to be sorted in the table.
 - ❖ **setToTable** – Once the user chooses how he/she wants the data to be sorted, this method calls the Sort. methods accordingly and sets the data to the table then calls the ViewAllItems.table method
 - ❖ **table** – this method creates the table and displays it to the user with the data sorted as requested

Description of the features of the application

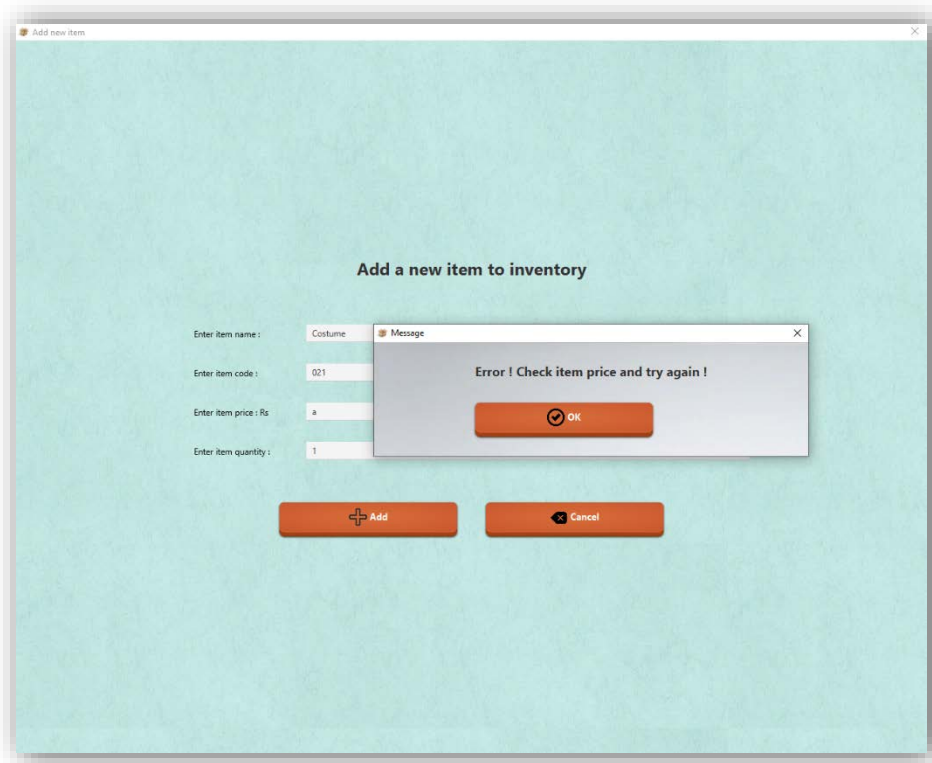
This application has a GUI (Graphical User Interface). It helps the user to control the stock of his/her items. It allows the user to add and remove items from the stock, allows the user to change the stock quantity, allows the user to search for a particular item and also allows the user to view all the data in the stock in the form of a table sorted according to the user's choice. Also, in case of errors, it allows the user to restore the default stock data.

Concepts/theories implemented

- 1) **OOP (Object Oriented Programming)** - Object Oriented Programming is a programming concept that works on the principle that objects are the most important part of your program. It allows users create the objects that they want and then create methods to handle those objects.
- 2) **Files** – Files are used in the application so that the user can keep the data even after the application is closed. It is easier to store data in files than in array. In case the application crashes, the data won't be lost as they will be stored in the file.
- 3) **Classes** – Classes are used to separate the codes. For example, all codes related to the addition of a new item to the stock/inventory/file are in the AddItem class
- 4) **Methods** – Different methods are used to separate codes and easier to edit in case of errors. For example. All the codes related to sorting by item name are in the Sort.itemName method. Thus, allowing easier understanding of the code
- 5) **Validation checks** – this implies that the application checks the input values and displays an error in case the input criteria is not met. For example, if the input criteria is an integer and the user inputs a string value such as "a", the application will show an error message and ask the user to input again.

```
if (itemName.EqualsIgnoreCase("")) {  
    MessageBox.box("Item name field cannot be empty !");  
    return;  
}  
else if (SimilarityCheck.itemName(itemName) == 1) {  
    MessageBox.box("Error ! Item name already exists !\nCheck and try again !");  
    return;  
}  
else if (!itemName.Matches("[a-zA-Z]+$")) {  
    MessageBox.box("Error ! Check item name and try again !");  
    return;  
}  
else if (itemCode.EqualsIgnoreCase("")) {  
    MessageBox.box("Item code field cannot be empty !");  
    return;  
}  
else if (SimilarityCheck.itemCode(itemCode) == 1) {  
    MessageBox.box("Error ! Item code already exists !\nCheck and try again !");  
    return;  
}  
else if (!itemCode.Matches("[a-zA-Z0-9]+$")) {  
    MessageBox.box("Error ! Check item code and try again !");  
    return;  
}  
else if (itemPrice.EqualsIgnoreCase("")) {  
    MessageBox.box("Item price field cannot be empty !");  
    return;  
}  
else if (!itemPrice.Matches("[0-9]+$") || itemPrice.EqualsIgnoreCase("0")) {  
    MessageBox.box("Error ! Check item price and try again !");  
    return;  
}  
else if (itemQuantity.EqualsIgnoreCase("")) {  
    MessageBox.box("Item quantity field cannot be empty !");  
    return;  
}  
else if (!itemQuantity.Matches("[0-9]+$") || itemQuantity.EqualsIgnoreCase("0")) {  
    MessageBox.box("Error ! Check item quantity and try again !");  
    return;  
}  
}
```

This is the error message for the example mentioned:



Concepts implemented (without lecturer's help)

- 1) **Give the application an icon** – In addition to conveying brand personality through color and style, icons must first and foremost communicate meaning in a graphical user interface. Icons are, by definition, a visual representation of an object, action, or idea. Therefore, an icon is used to give the application a meaning.

```
primaryStage.getIcons().add(new Image("file:images/logo.png"));
```

2) **Give buttons an icon** - Icons are most effective when they improve visual interest and grab the user's attention. They help guide users while they're navigating a page. Sometimes, the user can know the function of the button without even reading the label – user-friendly approach.

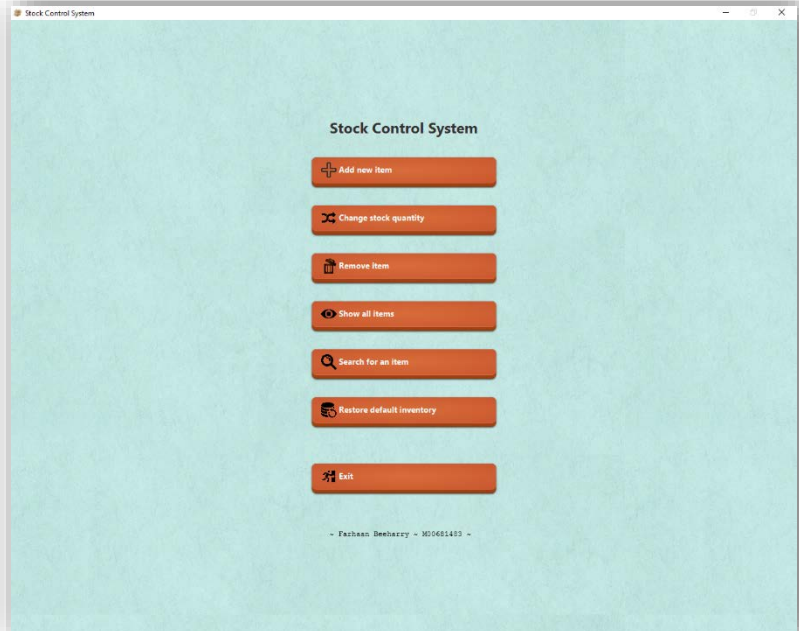
3) **Use of CSS (Cascading Style Sheet)** – Using CSS allows to change the

```
ImageView addIcon = new ImageView(new Image(new FileInputStream("images/addIcon.png")));
```

appearance of the application. Without CSS, each and every button would have to be individually coded. But with CSS, one line of code is enough to set all buttons to the same style. CSS saves lots of time and gives more choice of appearance. It is used to approach user friendliness, easy navigation and user attention.

```
scene.getStylesheets().add("file:stylesheet/stylesheet.css");
```

```
.root {
    -fx-background-image: url("../images/background.jpg");
}
.button {
    -fx-padding: 8 15 15 15;
    -fx-background-insets: 0,0 0.5 0, 0 0.6 0, 0 0.7 0;
    -fx-background-radius: 8;
    -fx-background-color:
        linear-gradient(from 0% 93% to 0% 100%, #a34313 0%, #903b12 100%),
        #9d4024,
        radial-gradient(center 50% 50%, radius 100%, #d86e3a, #c54e2c);
    -fx-effect: dropshadow( gaussian , rgba(0,0,0,0.75) , 4,0,0,1 );
    -fx-text-fill: white;
    -fx-text-effect: dropshadow( gaussian , #a30000 , 0,0,0,2 );
    -fx-font-weight: bold;
    -fx-font-size: 1.1em;
}
.button:hover {
    -fx-background-color:
        linear-gradient(from 0% 93% to 0% 100%, #a34313 0%, #903b12 100%),
        #9d4024,
        radial-gradient(center 50% 50%, radius 100%, #ea7f4b, #c54e2c);
}
.button:pressed {
    -fx-padding: 10 15 13 15;
    -fx-background-insets: 2 0 0 0,2 0 3 0, 2 0 4 0, 2 0 5 0;
}
.label {
    -fx-font-size: 24px;
    -fx-font-weight: bold;
    -fx-text-fill: #333333;
    -fx-effect: dropshadow( gaussian , rgba(255,255,255,0.5) , 0,0,0,1 );
}
.text-field {
    -fx-background-color: -fx-text-box-border, -fx-background ;
    -fx-background-insets: 0, 0 0 1 0 ;
    -fx-background-radius: 0 ;
}
.text-field:focused {
    -fx-background-color: -fx-focus-color, -fx-background ;
}
```



-
- 4) **Read the screen resolution** – This is used to read the screen's resolution and divide it accordingly to be used in the `ViewAllItems.table` method so that the table fit the screen with equal widths for all 4 columns as different monitors have different resolution.

```
double screenWidth = Screen.getPrimary().getBounds().getWidth();  
double columnWidth = (screenWidth - 4) / 4;
```

- 5) **Regular Expressions (RegEx)** – it defines a search pattern for strings. The abbreviation for regular expression is *regex*. The search pattern can be anything from a simple character, a fixed string or a complex expression containing special characters describing the pattern. It is used in this application for validation checks.

Integer only check: `if (!itemQuantity.matches("^([0-9])+$"))`

String only check: `if (!itemName.matches("^([a-zA-Z])+$"))`

Double only check: `if (!itemPrice.matches("^([0-9.])+$"))`

String and integer check: `if (!itemCode.matches("^([a-zA-Z0-9])+$"))`

Problems encountered and solution(s) found

- 1) A method `Exit.exit` was created to ask the user whether he/she is sure about exiting the application. When clicking on the close icon on the top right of the window, the method is called. But even if the user clicks on “NO”, the application still exits.

Solution: `event.consume()` was used to cancel the action of the close icon and allows only the “YES” in the `Exit.exit` method to close the application

```
primaryStage.setOnCloseRequest(event -> {  
    try {  
        event.consume();  
        Exit.exit();  
    } catch (Exception ex) {  
    }  
});
```

- 2) When adding icons to buttons, the buttons were distorted. The size of the icon was too big for the buttons.

Solution: `.setFitWidth()`; and `.setFitHeight()`; were used to change the size of the icons so that they fit inside the button.

```
yesIcon.setFitWidth(25);  
yesIcon.setFitHeight(25);
```

- 3) On different computers with different screen resolutions, the application runs differently. Sometimes the height is more than that of the screen.

Solution: To fix this, `.setMaximized(true)`; is used so that the application takes full advantage of the screen size and fits the screen.

```
searchWindow.setMaximized(true);
```

- 4) Buttons sometimes were misaligned. They were either too far on the right or not centered at all. This was a major issue in designing the GUI.

Solution: To fix this, HBox was used to place the buttons. Then the HBox was centered in the GUI. This fixed the issue everywhere in the application.

```
HBox Hbtns = new HBox(40);  
Hbtns.getChildren().addAll(btnAdd, btnCancel);  
Hbtns.setAlignment(Pos.CENTER);
```

- 5) As ArrayList was not permissible in this coursework, to display the data in the form of a table was a big issue.

Solution: Read from file into an array of objects. Then use TableView to display the array of objects in the form of a well-organized table.


```
TableView tableItems = new TableView();
```

- 6) When using arrays, each time the application is closed, only hard-coded values stays in the application. All the user manipulated data were lost.

Solution: Files were used to read and write the data which prevented loss. This method helped the user to keep his stock updated and no need to start from scratch. All updated information is stored in the file.

```
import java.io.*;
```

These two classes are used to write to the file and read from the file respectively.



DataExport.java
DataImport.java

How was the project carried out?

First step was to create a mini project with only console input and output which had simple functions such as add new item, show all and check for similarities. Then I removed the console inputs and outputs and replaced them with JOptionPane and added more features such as delete an item from the inventory. Then I started to use files to read and write data so that no data is lost when the application is closed. When the main features of the application were good, I started building a GUI so that the user can easily use the application. After building the GUI, I added more options to some features (mini project had only search by item code, the final one has search by item name and item price). After the building of the GUI and the functions were done and the application was usable, CSS was used to beautify the application, to make it attractive and prevents the user from getting bored when dealing with it.

References

- **Vogella. (2019). [online]**
Available at: <https://www.vogella.com/tutorials/JavaRegularExpressions/article.html>
- **Oracle Help Centre. (2019). [online]**
Available at: <https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/Button.html>
- **GeeksforGeeks. (2019). [online]**
Available at: <https://www.geeksforgeeks.org/java/>
- **Stack Overflow. (2019). [online]**
Available at: <https://stackoverflow.com/>
- **Jenkov.com. (2019). [online]**
Available at: <http://tutorials.jenkov.com/javafx/index.html>