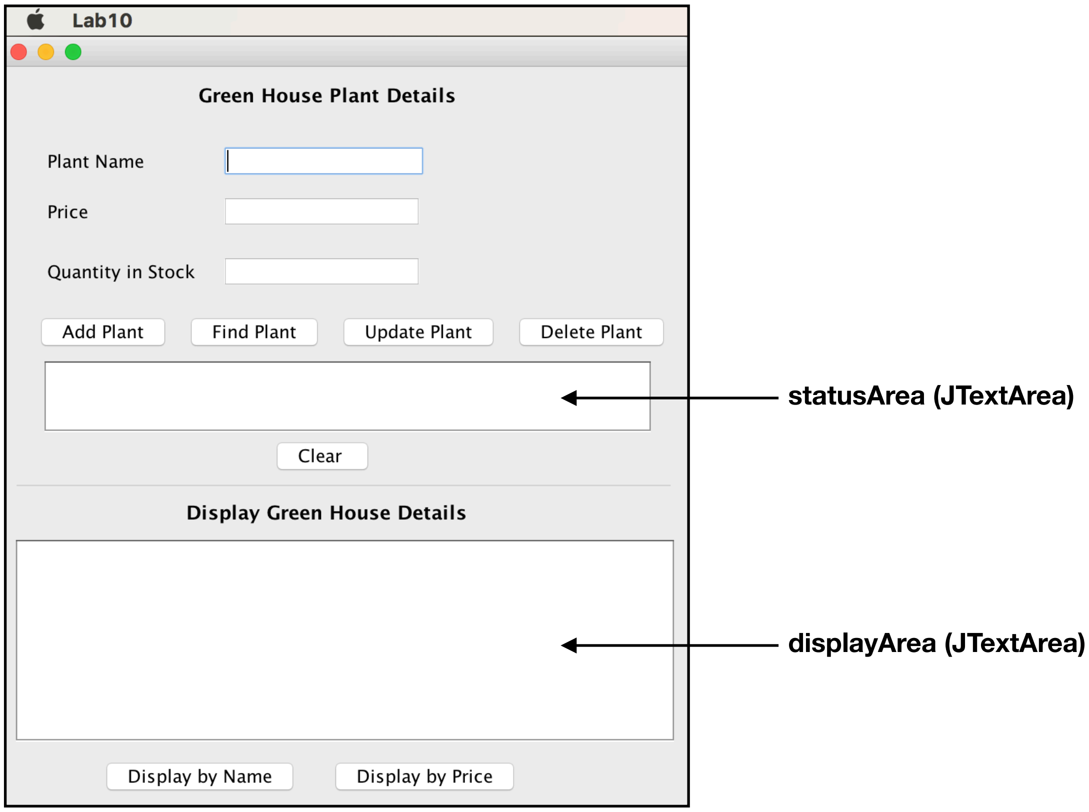
The University of the West Indies, St. Augustine
COMP 2603 Object Oriented Programming 1
2020/2021 Semester 2
Lab 10

In this lab, we will connect GUIs with a domain class and experiment with a few collections.

Download link for the Netbeans IDE (Java SE): https://netbeans.org/downloads/

Part 1: Importing and setting up the Lab10 project in Netbeans

1. Download the zipped Netbeans project file, Lab10.zip, from myElearning. Extract the file on your Desktop and open in it in the Netbeans editor (File -> Open Project)
2. Expand the lab 10 package and observe the GUI class, **GreenhouseGUI.java**, the domain classes **Nursery.java** and **Plant.java**, and the runner class, **Lab10.java**.
3. Run the project and observe how the GUI looks. Observe that action listeners have been set up for the JButton objects already.
4. Familiarise yourself with the code and documentation in the domain classes.



Part 2: Connecting the GUI to a domain class

The view class (**GreenhouseGUI**) needs to be connected to the domain class (**Nursery**) so that the data collected can be sent along for useful processing, and also so that results can be sent back and displayed for the user.

1. Update the **GreenhouseGUI** class so that the constructor is overloaded with one that accepts a **Nursery** object. You need to add a **Nursery** object attribute to the **GreenhouseGUI** class - name it **nursery**. This is the association object that connects the two classes.
2. Create an instance of the **Nursery** class in the main method of the runner class. Pass the instance into the **GreenhouseGUI** constructor. This sets up a relationship between the view layer and domain layer where the view does not know how the processing takes place. Further, the domain is completely unaware of the view layer and is therefore decoupled in terms of processing.
3. We can now invoke the services of the **Nursery** object in the **GreenhouseGUI**.

Part 3: Passing data to and from the GUI and domain class

The action listener method bodies for all of the buttons have been set up in the view layer so that GUI events trigger the appropriate action.

1. In the **GreenhouseGUI** class, let's make the **Add Plant** button work. Add code to:
    a. Collect the String data entered into the textfields when a user fills in a plant name, price and quantity and clicks on the **Add Plant** button.
    b. Pass the String data to the **nursery** object by invoking the **addPlant**(**String name, String price, String quantity)** method.
    c. Collect the String returned by the **addPlant(..)** method, and display it on the JTextArea object called **statusArea**.
2. The button works, but the functionality needs to be added to the domain class now.
3. Note that the **GreenhouseGUI** does not have any references to the **Plant** class. All interactions are done through the **Nursery** class using Strings objects and a collection of Strings.

Part 4: Adding functionality to the domain class

The Nursery class needs to have a collection in which to store the plant objects that will be created and manipulated by the application.

1. In the **Nursery** class, create a new **Collection** object called **plants**, that stores Plant objects. Initialise the collection in the Nursery constructor as an **ArrayList**.
2. Add code to the **addPlant(..)** method that creates a new **Plant** object and inserts it into the **plants** collection. The method should return the default message if these steps fail, otherwise it should return "Plant successfully added".
3. Test whether the **Add Plant** button works properly now. Add error checking code as necessary.

TIP: Converting a String to a primitive type : int

```
import
java.lang.Integer;
String s = "10";
int num =
Integer.parseInt(s);
```

Part 5: ArrayList as a Collection - Duplicates allowed, unsorted, reliance on equals()

1. In the **Nursery** class, add the following lines of code to the **getPlantsByName(**) method before the return statement in the method:

```
if(!plants.isEmpty())
  msg = plants.toString();
```

   What does the method do now?

   Answer:

   If the plant collection is not empty, it would invoke its toString() method. This should do a good job at printing the plants in the collection.

2. In the **GreenhouseGUI** class, add functionality so that the **Display by Name** button presents a list of all the plants (and their details) stored in the collection. This requires code to be added to the **sortByNameButtonActionPerformed(..)** method that invokes the **getPlantsByName( )** on the nursery object and displays the String returned by the method in the **displayArea** JTextArea in the GUI.

3. Run the Project and try adding a few plants e.g.

```
Plant Name: Aloe, price: 10.00, quantity: 50
Plant Name: Penta, price: 10.00, quantity: 12
Plant Name: Hosta, price: 6.00, quantity: 10
Plant Name: Aloe, price: 10.00, quantity: 50
```

4. Are duplicate plants allowed? Why?

   Answer:

   No. Because the plants collection is an ArrayList, which allows duplicates.

5. Click on the **Display by Name** button. Is the list sorted by name? Why not?

   Answer:

   No. Because we did not sort the ArrayList before outputting.

6. In the **Nursery** class, let's add the functionality to find a plant in the collection:
   a. Uncomment the code in the body of the **getPlantObject(String plantName)** method. Observe what the method does:
      - It creates a new **Plant** object with the name of the plant that we are trying to find in the **Collection**.
      - It then uses the **contains(..)** method of the **Collection** interface to test whether the **Plant** object is in the **Collection**.
      - If the **Plant** object is not in the **Collection**, null is returned.
      - If the **Plant** object is indeed in the **Collection**, then the code iterates through the **Collection** and checks each **Plant** object for equality with the one we created. If found, then the **Plant** object in the collection is returned.

   Why couldn't we just use the **get(..)** method of the **ArrayList** to locate the **Plant** object in the collection?

Answer:

Because get() requires an index of the plant that we want to find. We would not know this index beforehand.

   b. In the **getPlant(String plantName)** method, use the **getPlantObject(..)** method to locate the **Plant** object with the corresponding name. Invoke the appropriate method in the Plant class so that the details of the plant are returned as individual Strings stored in an ArrayList.

7. Now, let's make the **Find Plant** button in the GUI functional. When the button is clicked, the application should:
   - Capture the name of the plant typed into the GUI by the user
   - Search the plants collection in the Nursery for the corresponding object
   - Display the price and quantity of the plant if found. In addition, the message "Plant details are shown" should be displayed in the status area.
   - If no plant is found, then the message "No <insert plant name> plant found " should be displayed in the status area.

   Add code to the appropriate action handling method in the **GreenhouseGUI** class that invokes the **getPlant(String plantName)** method on the **nursery** object to achieve this.

   Couldn't we just return the **Plant** object found in the **Nursery** to the **GreenhouseGUI** and let the GUI extract the data directly from the Plant object? Why wasn't this approach used?

Answer:

We could, BUT that would lead to coupling between the Plant and GUI classes. We want to avoid this, hence the reason for the Nursery class' existence.

8. Test whether the **Find Plant** button works properly. Add the following plant:

```
Plant Name: Aloe, price: 10.00, quantity: 50
```

You should be able to add a **Plant** successfully, but the message " No Aloe plant found" is displayed when you try to find the plant. The plant is clearly in the collection (test using the **Display by Name** button) but it is not being found by the **getPlantObject(String plantName).**

How does the **contains(..)** method of the **Collection** interface locate a particular **Plant** object in the **plants** collection? Did we write an **equals( )** method in **Plant**? How does this result in the **getPlantObject(..)** method's failure to locate our Plant?

Answer:

If we do not have an equals() method in our Plant class. The contains() method that was invoked in the getPlantObject() method of the nursery class would always return null. Because the default java equals method would be invoked, which only checks that two plants variables are the same.

9. Override the **equals( )** method in the **Plant** class so that it checks equality based on the name of the plant.

10. Repeat step 8 and determine whether the **Find Plant** button works properly now. Why does it work? What type of **Collection** is **plants**?

Answer:

Yes it does. This is because the equals method of the Plant class allows the getPlantObject method of the nursery class to work as expected.

Additional Activities

- Add code to the application to make the Update Plant button work
- Add code to the application to make the Delete Plant button work.