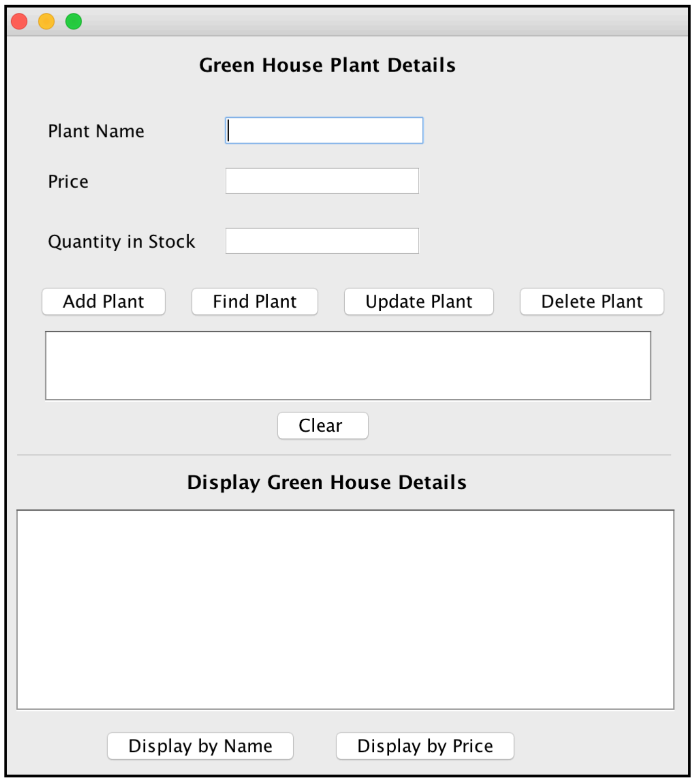
The University of the West Indies, St. Augustine
COMP 2603 Object Oriented Programming 1
Lab 9

In this lab, we will connect GUIs with a domain class and experiment with a few collections: TreeSets and HashSets, and the Comparable interface.



Part 1: Importing and setting up the Lab9 project

1. Download the zipped Netbeans project file, Lab9.zip, from myElearning. Extract the file on your Desktop and open in it in the Netbeans editor (File -> Open Project)
2. Open and observe the GUI class, **GreenhouseGUI.java**, the domain classes **Nursery.java** and **Plant.java**, and the runner class, **Lab9.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:

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:

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

   Answer:

Part 6: HashSet as a Collection - Duplicates not allowed, unsorted, reliance on hashCode( ) and equals( )

Let's deal with the unwanted duplication of the plants now. We can fix this easily using a different collection - a HashSet. A **HashSet** is backed by a hash table (actually a HashMap instance) and it does not allow duplicate objects to be stored.

1. In the **Nursery** class, change the *dynamic* type of **plants** to **HashSet**. Why can we do this without needing to change any of the code in the **Nursery** class?

   Answer:

2. Try adding the following plants using the GUI:

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

   Did it work? Explain what you observe and why it occurs

   Answer:

3. Override the **hashCode( )** method in the **Plant** class so that it generates a hash code using the String produced by the toString( ) method. Repeat step 2. What do you observe?

   Answer:

4. Override the **hashCode( )** method in the **Plant** class so that it generates a hash code using the same criteria that is used to test equality in the **equals( )** method. Repeat step 2 again. What do you observe this time? Why must these methods use the same criteria on which to base their functionality?

   Answer:

5. Add a few plants:

```
        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
```

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

   Answer:

Part 7: TreeSet as a Collection - Duplicates not allowed, sorted, reliance on hashCode( ) and equals( ), requirement of Comparable objects

Let's deal with the sorting of the plants by name now. We can fix this easily again using a different collection - a TreeSet. A **TreeSet** does not allow duplicate objects to be stored. The elements are ordered using their natural ordering, or by a `Comparator` provided at set creation time, depending on which constructor is used.

1. In the **Nursery** class, change the *dynamic* type of **plants** to **TreeSet**. Why can we do this again without needing to change any of the code in the **Nursery** class?

   Answer:

2. Try adding the following plants using the GUI:

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

   Did it work? Explain what you observe and why it occurs.

   Answer:

3. Modify the **Plant** class so that it implements the **Comparable** interface. What method are you required to add to the **Plant** class now?

   Answer:

4. Add a **int compareTo(Object obj)** method to the **Plant** class that compares the name of two **Plant** objects and returns 0 if the names are identical, returns 1 if the plant's name is alphabetically higher than the one being compared to, or returns -1 if the plant's name is alphabetically lower than the one being compared to. A **java.lang.IllegalArgumentException( )** should be thrown if a non-Plant object is supplied for comparison.

5. Try adding the following plants using the GUI:

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

   Did it work? Why did this work now?

   Answer:

6. Click on the **Display by Name** button. Is the list sorted by name? How about if we wanted to make it sorted in descending order instead. What would you change in the compareTo( ) method?

   Answer:

Part 8: TreeSet as a Collection - Duplicates not allowed, sorted, reliance on hashCode( ) and equals( ), use of Comparators for sorting

Let's deal with the sorting of the plants by price now. We can do this using a separate Collection to the plants collection, and a **Comparator** class.

1. In the **Nursery** class, create a private inner class called **PriceComparator** that implements the **Comparator** interface. Which method must the **PriceComparator** class provide?

   Answer:

2. Write the code for the **compare(..)** method in the **PriceComparator** class with the signature: **public int compare(Object o1, Object o2)**

   The method should compare the prices of two **Plant** objects and return 0 if the prices are identical, return 1 if plant 1's price is larger than plant 2's, or return -1 if the plant 1's price is smaller than plant 2's.
   A **java.lang.IllegalArgumentException( )** should be thrown if a non-Plant object is supplied for comparison.
3. In the **Nursery** class, in **getPlantsByPrice( )** method, create a new **PriceComparator** object. Create a new **TreeSet** collection called **plantsByPrice** that orders elements based on the new **PriceComparator** object. Add all of the elements in the current **plants** collection to the **plantsByPrice** collection. The method should now return the result of invoking the toString( ) method on the **plantsByPrice** collection if it is not empty.

   What does the **getPlantsByPrice( )** method do now?

   Answer:

4. Try adding the following plants using the GUI:
   ```
   Plant Name: Aloe, price: 10.00, quantity: 50
   Plant Name: Penta, price: 10.00, quantity: 12
   Plant Name: Hosta, price: 6.00, quantity: 10
   ```

   Does everything still work? Try out the Display by Price button. Did it sort by price? Why not? What do you need to do?

   Answer:

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

6. Repeat step 4. Toggle between clicking on the **Display by Price** button and the

Display by Name button. What do you observe? Explain what is happening.

Answer:

7. We need to fix the **getPlantsByPrice( )** method so that the new **PriceComparator** object used with a different collection. Delete the **TreeSet** from step 3 and all related code for that object. Create a new **ArrayList** collection called **plantsByPrice** that is initialised with all of the elements in the current **plants** collection.

   Explore the Collections interface for a method that will sort the **ArrayList** using the **PriceComparator**.
   What is the name of this method and how is it invoked?

   Answer:

   Add code to the **getPlantsByPrice( )** method based on your answer above.

TIP: There is a difference between Collection and Collections (plural) interfaces in Java

8. Repeat step 4 again. Toggle between clicking on the **Display by Price** button and the **Display by Name** button. What do you observe now?

   Answer:

Additional Activities - At Home Practice

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