Week 2, Lab 2

<u>Learning Objectives</u>

- Define custom classes with state (primitive variables) and behaviour (accessors, mutators, methods)
- Create no-argument and simple constructors
- Implement a toString( ) method that makes use of accessor methods to return an appropriate description of an object.
- Instantiate classes by creating objects in a main class
- Import and use Java API classes (File reading, String tokenisation)

Create a new BlueJ Project called Lab2. Add the vehicles.txt file from myElearning to your project.

<u>FuelStation Class</u>

1. Write the code for a **FuelStation** class with:
   a. the following attributes:
      - **fuelType** (String) - default is gasoline
      - **fuelVolume** (double) - default is 75000
      - **fuelRate** (double) - default is 2
      - **fuelSales** (double) - default is 0

```
public class FuelStation {
   private String fuelType = "gasoline";
   private double fuelVolume = 75000;
   private double fuelRate = 2.00;
   private double fuelSales = 0;
}
```

Note: Ensure that information hiding principles are not violated. Use private access modifiers for all attributes.

   b. a **no-argument constructor** that initialises state of all attributes to their default values.

```
public FuelStation(){
   String fuelType = "gasoline";
   double fuelVolume = 75000;
   double fuelRate = 2.00;
   double fuelSales = 0;
}
```

   c. **accessor** methods for all attributes. Follow the camelCase convention.

```
   public String getFuelType(){
      return fuelType;
   }
   public double getfuelVolume(){
      return fuelVolume;
   }
   public double getfuelRate(){
      return fuelRate;
   }
   public double getfuelSales(){
      return fuelSales;
   }
```

d. a t**oString( ) method** that produces and returns a string as follows: FUEL: gasoline VOL: 75000.0L PRICE PER L: $2.00 SALES: $0.00

```
public String toString(){
      String output;
      output= "Fuel: " + getFuelType() + " Vol: " + getfuelVolume() + " Price: " + getfuelRate() + "
Sales: " + getfuelSales();
      return output; }
```

2. Add the following methods (error checking is required in all cases) to the FuelStation class:

Method Signature	**Return Type**	**Purpose**
sellFuel(double volume)	boolean	Decreases the fuelVolume of the FuelStation by the supplied volume, if possible, updates the sales values, returns true if successful, false otherwise. This method is private.
canDispenseFuelType (String fuelType)	boolean	Returns true if the FuelStation dispenses fuel of the supplied type, false otherwise. This method is public.
dispense(String fuelType, double volume)	boolean	Returns true if the FuelStation can dispense the supplied type and volume of fuel, false otherwise. This method is public.

```
private boolean sellFuel (double volume){
   if (this.fuelVolume > volume && volume > 0){
      fuelVolume = fuelVolume - volume;
      fuelSales += volume * fuelRate;
      return true;
   }
   return false;
}
public boolean canDispenseFuelType (String fuelType){
   if (fuelType != null){
      return fuelType.equals(getFuelType());
   }
   return false;
}
public boolean dispense (String fuelType, double volume){
   if (canDispenseFuelType(fuelType) == true) {
      return sellFuel(volume);
   }
return false; } }
```

Main Class

3. Create a main class, **StationSimulation**, that creates an instance of the **FuelStation** class and prints the result of invoking the **toString( )** method of the instance. Observe the relationship created in the BlueJ editor.

```
import java.util.Random;
import java.io.File;
```

```
public class StationSimulation{

public static void main (String [] args){
      //3
      FuelStation classObj = new FuelStation();
      System.out.println(classObj);
}}
```

4. Test the three methods created in Step 2 using randomly generated values.

```
 import java.util.Random;
Random r = new Random ();
      int i = r.nextInt (999);

      classObj.dispense("gasoline", i);
      System.out.println(classObj);
```

File Reading using Scanner

5. Import the **Scanner** and **File** classes from the Java API into the **StationSimulation** class.

```
import java.util.Scanner;
import java.io.File;
```

6. Add code to read the data from the **vehicles.txt** file and print out each line of data to the screen. Remember to use try/catch blocks.

```
 try{
         File dataFile = new File ("vehicles.txt");
         Scanner scanner = new Scanner (dataFile);
         String carData = " ";

         while (scanner.hasNextLine()){
            carData = scanner.nextLine();
            System.out.println (carData.spilt);
           Vehicle v = createVehicle (CarData.split(" , "));
         }
      }
      catch (Exception e){
      }
```

Vehicle Class

7. Write the code for a **Vehicle** class with:
   a. the following attributes:
      - **tankCapacity** (int)
      - **fuelType** (String)

```
public class vehicle {
 private int tankCapacity;
 private String fuelType;
}
```

b. a **constructor** that accepts **3 int values** representing the **length**, **width**, **breadth** of the Vehicle's fuel tank. These variables are used to calculate and initialise the **tankCapacity** variable. The **fuelType** should be assigned to either gasoline (if the tankCapacity is even) or diesel (if the tankCapacity is odd).

```
public Vehicle(int length, int width, int breadth){
     tankCapacity = length * width * breadth;
     if (tankCapacity % 2 ==0){
        fuelType = "gasoline";
     }
     else {
        fuelType = "diesel";
     }
  }
```

c. **accessor** methods for all attributes. Follow the camelCase convention.

```
public int getTankCapacity(){
     return tankCapacity;
  }
  public String getFuelType(){
     return fuelType;
  }
```

d. a **toString( )** method that produces and returns a string as follows:
VEHICLE TANK CAPACITY: 60 FUEL TYPE: gasoline

```
 public String toString(){
     String info = "VEHICLE TANK CAPACITY: " + getTankCapacity() + "FUEL TYPE: " +
getFuelType();
     return info;
  } }
```

Putting it all together

8. Modify your code block from Step 6 in the **StationSimulation** class to create **Vehicle** objects for each line of data. The three values on each line represent the length, width, breadth of a Vehicle's fuel tank. You will need to parse and extract the values from the line (Tip: Use StringTokenizer or a String's split(..) method) .Test your code works by printing out the details of each Vehicle object (using the toString( )).

```
public static Vehicle createVehicle(String[] dimensions){
          int length = Integer.parseInt(dimensions[0]);
         int breadth = Integer.parseInt(dimensions[1]);
         int width = Integer.parseInt(dimensions[2]);
        Vehicle v = new Vehicle( length, breadth, width) ;

         return v;  }
```

9. Write code in the StationSimulation class to fill up the tanks of each Vehicle and print the outcomes. Diesel vehicles should not get any fuel. Sample output below:

```
FUEL: gasoline VOL: 75000.0L PRICE PER L: $2.00 SALES: $0.00

VEHICLE TANK CAPACITY: 60 FUEL TYPE: gasoline
Filled up: true
FUEL: gasoline VOL: 74940.0L PRICE PER L: $2.00 SALES: $120.00

VEHICLE TANK CAPACITY: 42875 FUEL TYPE: diesel
Filled up: false
FUEL: gasoline VOL: 74940.0L PRICE PER L: $2.00 SALES: $120.00

VEHICLE TANK CAPACITY: 400 FUEL TYPE: gasoline
Filled up: true
FUEL: gasoline VOL: 74540.0L PRICE PER L: $2.00 SALES: $920.00
```

```
Public static void serviceVehicle (Vehicle v, FuelStation f){
        System.out.println(v);
        System.out.println("Filled Up: " + f,dispense(v.getFuelType(), v.getTankCapacity()));
        System.out.println(f);
}
```

Week 3, Lab 3

Learning Objectives

- Create and use class and instance variables
- Create and use overloaded methods, constructors
- Implement an equals(..) method in a class
- Implement relationships: association, composition
- Use an ArrayList to manage multiple objects

Create a new BlueJ Project called Lab3. Add the Vehicle.java and VehicleDriver.java files provided on myElearning to your project. You may use the Vehicle class you completed in Lab 2.

Part 1: Class and instance variables

1. Modify the **Vehicle** class so that all new vehicle objects are given a unique license plate ID with the form XXXYY where X is an uppercase letter and Y is a digit from 0 to 9. Examples are shown in the sample output in step 3.

   - Add a new class variable **plateNumberCounter** (int): initialised to 1

   ```
   private static int plateNumberCounter = 1;
   ```

   - Add a new instance variable **plateID** (String)

   ```
   private String plateID;
   ```

   - Add a private mutator method, **setPlateID( )**, that constructs a plateID String by appending ‘TAB’ to the appropriate number counter value and sets the **plateID** variable. The **plateNumberCounter** increments by 1 each time.

   ```
   private void setPlateID(){
       if  (plateNumberCounter < 10)
          plateID = "TAB0" + plateNumberCounter;
       else
          plateID = "TAB" + plateNumberCounter;
       plateNumberCounter++;  }
   ```

   - Add a public accessor method, **getPlateID( )** for the **plateID** variable.

   ```
   public String getPlateID(){
       return plateID;    }
   ```

2. Modify the **Vehicle** toString( ) method to include the plateID in the string description.

```
public String toString(){
return ("VEHICLE TANK CAPACITY: " + getTankCapacity() + "FUEL TYPE: " + getFuelType() +
"PLATEID: " + getPlateID());  }
```

3. Create a new main class, **StationSimulation**, and use a loop to create 10 **Vehicle** objects and print their details. Use random int values for the **Vehicle** constructor parameters. See here for steps: https://www.baeldung.com/java-generating-random-numbers-in-range

```
VEHICLE TANK CAPACITY: 144 FUEL TYPE: gasoline PLATE ID: TAB01
   VEHICLE TANK CAPACITY: 72 FUEL TYPE: gasoline PLATE ID: TAB02
   VEHICLE TANK CAPACITY: 96 FUEL TYPE: gasoline PLATE ID: TAB03
   VEHICLE TANK CAPACITY: 108 FUEL TYPE: gasoline PLATE ID: TAB04
   VEHICLE TANK CAPACITY: 20 FUEL TYPE: gasoline PLATE ID: TAB05
   VEHICLE TANK CAPACITY: 48 FUEL TYPE: gasoline PLATE ID: TAB06
   VEHICLE TANK CAPACITY: 16 FUEL TYPE: gasoline PLATE ID: TAB07
   VEHICLE TANK CAPACITY: 44 FUEL TYPE: gasoline PLATE ID: TAB08
   VEHICLE TANK CAPACITY: 180 FUEL TYPE: gasoline PLATE ID: TAB09
   VEHICLE TANK CAPACITY: 153 FUEL TYPE: diesel PLATE ID: TAB10
```

```
public class StationSimulation{
  public static void main (String[] args){
     for (int i =0; i <10; i++){
        Vehicle v = new Vehicle (StationSimulation.getRandomNumber(1,20),
        StationSimulation.getRandomNumber(1,5),
        StationSimulation.getRandomNumber(1,5),
        StationSimulation.getRandomNumber(1,5));
```

```
        System.out.println(v);   }
 public static int getRandomNumber(int min, int max) {
   return (int) ((Math.random() * (max - min)) + min);
   }}}
```

Part 2: Overloaded methods and constructors

In the **Vehicle** class:

4. Add a new attribute, **vehicleClassification** (int), that stores the vehicle's classification, together with appropriate **accessor** and **mutator** methods (error checking should be done - valid values are shown in Table 1). Add the vehicle classification to the **toString( )** description using the accessor **getVehicleClassification( )**.

```
public int getVehicleClassification(){
        return vehicleClassification; }

public void setVehicleClassification(int value){
  if (value ==1 || value ==4)
      vehicleClassification = value;

   else
         vehicleClassification = 3;  }
```

The toString method would be modified to:

```
public String toString(){
return ("VEHICLE TANK CAPACITY: " + getTankCapacity() + "FUEL TYPE: " + getFuelType() +
"PLATEID: " + getPlateID() + "CLASSIFICATION: " + getVehicleClassification());  }
```

5. Add an overloaded accessor **getVehicleClassification( )** method that accepts a **vehicleClassification** (int) value and returns the corresponding type of vehicle according to Table
   2. Tip: Use a switch block https://www.w3schools.com/java/java_switch.asp

Vehicle Classification	**Type of Vehicle**	**Assignment**
1	Motorcycle	4th parameter in overloaded Vehicle constructor
3	Light motor vehicle	Default in existing Vehicle constructor
4	Heavy motor vehicle	4th parameter in overloaded Vehicle constructor

Table 2

```
public String getVehicleClassification(int vehicleClassification){
      switch (vehicleClassification){
         case 1: return "Motorcycle";
         case 3: return "Light Motor Vehicle";
         case 4: return "Heavy Motor Vehicle";
      }
      return null;
 }
```

6. Modify the **toString( )** method so that the overloaded accessor method is also invoked

Sample output at this point:

```
VEHICLE TANK CAPACITY: 441 FUEL TYPE: diesel PLATE ID: TAB01 VEHICLE CLASSIFICATION: 3 Light Motor Vehicle
```

```
public String toString(){
return ("VEHICLE TANK CAPACITY: " + getTankCapacity() + "FUEL TYPE: " + getFuelType() +
"PLATEID: " + getPlateID() + "CLASSIFICATION: " + getVehicleClassification() + " " +
getVehicleClassification(this.vehicleClassification) );
   }
```

7. Add an overloaded constructor **Vehicle(…)** that accepts a 4th parameter (int) which is used to initialise the vehicleClassification attribute**.** Observe how the original 3-argument constructor is invoked using the keyword 'this'

```
public Vehicle (int length, int breadth, int width, int vehicleClassification){
        this(length, breadth, width);
        setVehicleClassification(vehicleClassification);

   }
```

Part 3: Object equality: equals(..)

8. In the **Vehicle** class, add the following **equals(Object obj)** method that checks object equality using the vehicle **plateID** and returns true if the plateIDs are equal, false otherwise.

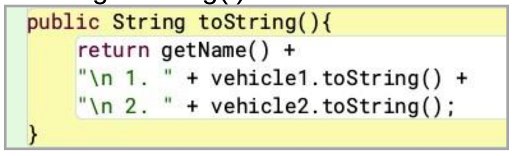
```
public boolean equals (Object obj){
    if (obj instanceof Vehicle){
      Vehicle v = (Vehicle) obj;
      String otherVehiclePlateID = v.getPlateID();
      boolean result = this.plateID.equals(otherVehiclePlateID);
    }

    return false;
  }
```

Part 4: Implement association relationships

10. Modify the supplied **VehicleDriver** class to have 2 **Vehicle** objects, **vehicle1** and **vehicle2**, as private attributes. This sets up an association relationship between a **VehicleDriver** and a **Vehicle** where a driver can drive (up to) two specific vehicles. These should be set to null in the constructor.

```
public class VehicleDriver {
  private String name;
  private Vehicle vehicle1;
  private Vehicle vehicle2;

  public VehicleDriver(String name){
    this.name = name;
    vehicle1 = null;
    vehicle2 = null;
  }}

 public String getName(){
    return name;
  }
}
```

11. Add the following toString( ) method to the class:



```
public String toString(){
    return getName() +
    "\n 1. " + vehicle1.toString() +
    "\n 2. " + vehicle2.toString();
}
```

12. Write a method **addVehicle(..)** that accepts a **Vehicle** object and if valid (not null), sets **vehicle1** or **vehicle2** (accordingly if vehicle1 is already initialised) to the supplied vehicle object, and returns true if successful, false otherwise (meaning both variables have been set already). Note: vehicle1 and vehicle2 should be unique.
Tip 1: Check if an object obj is initialised as follows: if(obj == null)
Tip 2: Use the equals(..) method to check if vehicle1 has already been set to the suppled vehicle object.

```
 public boolean addVehicle (Vehicle v) {
   if (v != null){
     if (vehicle1 == null){
       vehicle1 = v;
       return true;
     }
     if (vehicle2 == null){
       vehicle2 = v;
       return true;
     }
   }

    return false;
  }
```

Part 5: Use an ArrayList to manage objects

13. In the StationSimulation class, create two **ArrayLists**: **drivers** and **vehicles** which hold 5 **VehicleDriver** objects and 10 **Vehicle** objects respectively using Generic Types.

```
import java.util.ArrayList;
ArrayList <Vehicle> vehicles = new ArrayList <Vehicle>(10);
    ArrayList <VehicleDriver> drivers = new ArrayList <VehicleDriver>(5);
    String [] names = { "Lou" , "Sue", "Drew", "Koo", "Murphy"};

    for (int i = 0; i < 5; i++){
      drivers.add(new VehicleDriver(names[i]));
    }
```

14. Traverse the **drivers** ArrayList and randomly allocate objects from the **vehicles** ArrayList to the driver objects. Tip: use a do-while loop to set the other one.

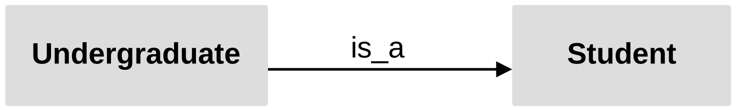
```
 for (VehicleDriver driver: drivers){
      int index = StationSimulation.getRandomNumber(0,10);
      Vehicle v = vehicles.get(index);
      driver.addVehicle(v);

      do {
        index = StationSimulation.getRandomNumber(0,10);
        v = vehicles.get(index);
      }
      while (driver.addVehicle(v));

      System.out.println(driver);
    }
  }
```

Expected Output

```
Lou
1. VEHICLE TANK CAPACITY: 36 FUEL TYPE: gasoline PLATE ID: TAB12 VEHICLE CLASSIFICATION: 1 Motorcycle
2. VEHICLE TANK CAPACITY: 48 FUEL TYPE: gasoline PLATE ID: TAB13 VEHICLE CLASSIFICATION: 4 Heavy Motor Vehicle Sue
 1. VEHICLE TANK CAPACITY: 48 FUEL TYPE: gasoline PLATE ID: TAB16 VEHICLE CLASSIFICATION: 3 Light Motor Vehicle  2.
VEHICLE TANK CAPACITY: 27 FUEL TYPE: diesel PLATE ID: TAB14 VEHICLE CLASSIFICATION: 3 Light Motor Vehicle Drew
 1. VEHICLE TANK CAPACITY: 72 FUEL TYPE: gasoline PLATE ID: TAB11 VEHICLE CLASSIFICATION: 4 Heavy Motor Vehicle  2.
VEHICLE TANK CAPACITY: 36 FUEL TYPE: gasoline PLATE ID: TAB12 VEHICLE CLASSIFICATION: 1 Motorcycle
Koo
1. VEHICLE TANK CAPACITY: 36 FUEL TYPE: gasoline PLATE ID: TAB12 VEHICLE CLASSIFICATION: 1 Motorcycle
2. VEHICLE TANK CAPACITY: 128 FUEL TYPE: gasoline PLATE ID: TAB19 VEHICLE CLASSIFICATION: 3 Light Motor Vehicle
Murphy
1. VEHICLE TANK CAPACITY: 132 FUEL TYPE: gasoline PLATE ID: TAB15 VEHICLE CLASSIFICATION: 3 Light Motor Vehicle
2. VEHICLE TANK CAPACITY: 72 FUEL TYPE: gasoline PLATE ID: TAB11 VEHICLE CLASSIFICATION: 4 Heavy Motor Vehicle ,
```

Lab 4

Inheritance, Method Refinement and Method Replacement



1. Create a new project in BlueJ called Lab 4.
2. Retrieve the following classes from the website: **Student.java** and **StudentApp.java** Compile both java files. Run the **StudentApp** file and observe the output.

3. Create a subclass of the **Student** class called **Undergraduate.**
   public class Undergraduate extends Student{}

4. Create a new **Undergraduate** object in the **StudentApp** class with the following details:

TIP: Public methods in the parent class are inherited and can be used by a child

Object	**Name**	**ID** (auto-gen)
u1	Barry Allen	30

Undergraduate u1 = new Undergraduate();
u1.setName("Barry Allen");

5. Print the details of the **Undergraduate** object **u1** using the **toString( )** method. What do you observe? Where did the state that is printed come from?

System.out.println(u1);
//the state is being printed from super

6. Now, add the following features to your **Undergraduate** class:
   a. Attributes: **minor (String)**, major **(String)**, credits **(int)**

```
 private String minor;
   private String major;
   private int credits;
```

   b. Accessors and mutators as necessary for each attribute

```
//mutators - setters
   public void setMinor (String minor){
      this.minor = minor;
   }
   public void setMajor (String major){
      this.major = major;
   }
   public void setCredits (int credits){
      this.credits = credits;
   }

   //mutators - getters
   public String getMinor (String minor){
      return minor;
   }
   public String getMajor (String major){
        return major;
    }
    public int getCredits (int credits){
        return credits;
    }
```

c. A default no-argument constructor for the **Undergraduate** class.

```
    public Undergraduate(){
     }
```

7. Create and/or modify the following **Undergraduate** objects in the **StudentApp** class:

Object	**Name**	**ID** (auto-gen)	**Major**	**Minor**	**Credits**
u1	Barry Allen	30	Forensics	Athletics	25
u2	John Rambo	40	Conflict Analysis	International Affairs	20
u3	Ellen Ripley	50	Astrobiology	Conflict Analysis	15

```
Undergraduate u1 = new Undergraduate();

u1.setName("Barry Allen");
u1.setMajor("Forensics");
u1.setMinor("Athletics");
u1.setCredits(25);

System.out.println(u1);

Undergraduate u2 = new Undergraduate();
u2.setName("John Rambo");
u2.setMajor("Conflict Analysis");
u2.setMinor("International Affairs");
u2.setCredits(20);

Undergraduate u3 = new Undergraduate();
u3.setName("Ellen Ripley");
u3.setMajor("Astrobiology");
u3.setMinor("Conflict Analysis");
u3.setCredits(15);

System.out.println(u1.toString());
System.out.println(u2.toString());
System.out.println(u3.toString());
```

8. Print the details of the **Undergraduate** objects **u1, u2, u3** using the **toString( )** method. What do you observe? Did all of the state print properly? Why not?
//the details will not be printed because there is no toString() method to do so

9. Let us now *refine* the **toString( )** method in the **Undergraduate** class. Write a **toString( )** in the **Undergraduate** class which returns the full details of an **Undergraduate** object (ID, name, fees, graduated, major, minor, credits). This method should call the **toString( )** method inherited from the parent class (Student) to achieve this.

TIP: Invoking a parent method from a child class

```
public String toString(){
    String details = super.toString();
    details += "\nUGRAD " + " MAJOR:" + major + " MINOR: " + minor + " CREDITS: " + credits;
    return details;    }
```

10. Run the **StudentApp** class again. What do you observe? Did all of the state print properly this time? How does it differ for the objects **s1**, **s2** compared to **u1**,**u2**, **u3**?

//it did print properly
//compared to the s objects, the minor and major are printed for the u objects

11. Overload the constructor of the **Student** class as follows:
public Student(String name)
The overloaded constructor should still set the state as in the original constructor.

```
public Student (String name){
        fees = 275.00;
        graduated = false;
        studentID = IDGenerator;
        IDGenerator = IDGenerator + 10;
        this.name = name;
    }
```

12. Compile and run the **StudentApp** class. Did it work? Did you have to change the code for the **Student** objects **s1** and **s2** in the **StudentApp** class to suit the new constructor? Why or why not?
//it will compile because if a parameter is not passed in the Student constructor it would just use the constructor will no parameter

13. What happens when you compile your **Undergraduate** class? Which of the superclass constructors do you think is called by the Undergraduate constructor?
//the no argument constructor will be used

14. Comment off the no-argument constructor in the Student class. Observe the (compilation) error generated in the Undergraduate class. Explain why this happens. Explain how it can be fixed. (remove your commented code when finished
//when commented there will be an error due to no parameter being passed, in order to fix this use super(" "); to pass in a string into the constructor

15. Create a subclass of the **Student** class called **Postgraduate** with the following:
public class Postgraduate extends Student{ }
   a. Attributes: **supervisor (String), thesisTitle (String), status (String)**

```
 private String supervisor;
   private String thesisTitle;
   private String status;
```

   b. A constructor that accepts a **name**, **supervisor** and **thesisTitle**.

```
public Postgraduate (String name, String s, String t){
        super(name);
        supervisor = s;
        thesisTitle = t;
        status = "Full Time" ; //part d
```

```
    }
```

c. A mutator for **status**

```
public void setStatus(String s){
    status = s;
  }
```

d. The default status for a student is full-time.

Note: Do not change the access modifiers of the Student class from private. You need to invoke the appropriate Student class constructor in order to set the name.

16. Create two **Postgraduate** objects in the **StudentApp** class with the following state

Object	Name	ID	Supervisor	Thesis Title
p1	John McClain	60	Prof. Asp Pirin	How to Die Hard
p2	Brian Mills	70	Dr. No Kia	Mobile Usage Patterns in Hostage Situations

```
Postgraduate p1 = new Postgraduate("John McClain", "Prof. Asp Pirin",
"How to Die Hard");
Postgraduate p2 = new Postgraduate("Brain Mills", "Prof. No Kia",
"Mobile Usage Patterns in Hostage Situations");
```

17. In the **Postgraduate** class, *refine* the inherited **toString( )** method from the **Student** so that the full details of a **Postgraduate** object are returned. Print the details of the **Postgraduate** objects **p1** and **p2** in the **StudentApp** class using your refined **toString( )**.

```
String details = super.toString();
 details += (" SUPERVISOR: " + supervisor + " THESIS TITLE: " + thesisTitle  "
STATUS: " + status);
 return details;   }
```

18. Override the method **calculateFees( )** in the **Undergraduate** class so that **Undergraduate** tuition fees are calculated based on the number of credits. Each credit costs $200.00.

```
 public void calculateFees(){
      setFees(credits *200);
```

19. Override the method **calculateFees( )** in the **Postgraduate** class so that
    - part-time **Postgraduate** tuition fees amount to $1,325.00
    - full-time **Postgraduate** tuition fees amount to $2,650.00

```
 public void calculateFees(){
      if(status.equals("part-time"))
         setFees(1325);
```

20. Change the status of the **Undergraduate** and **Postgraduate** objects in the **StudentApp** class as follows:
    a. John McClain: full-time
    b. Brian Mills: part-time
    c. Barry Allen: full-time

d. John Rambo: 25 credits
e. Ellen Ripley: 20 credits

Calculate their fees, and print their details once more.What do you observe?

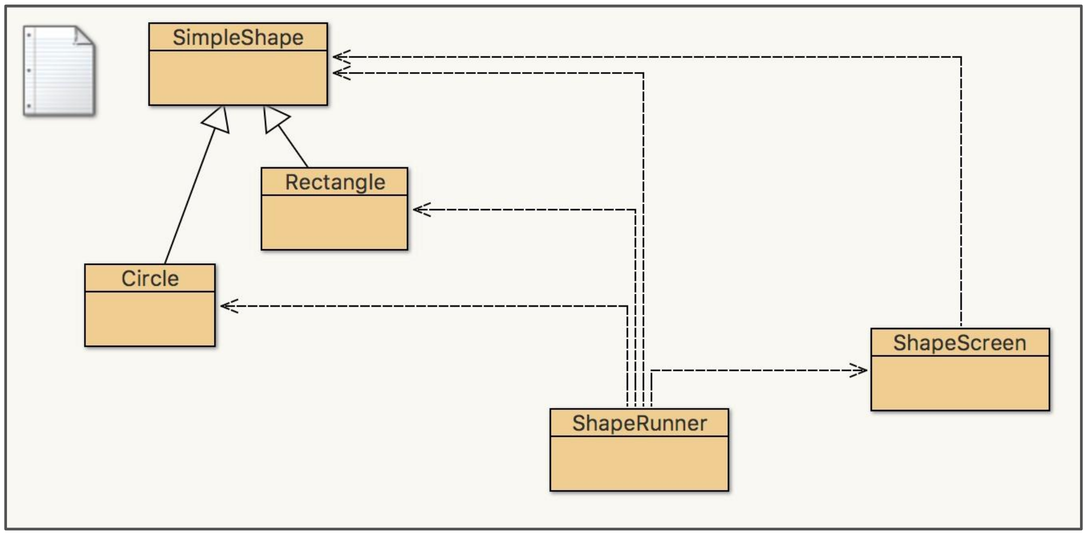
```
p1.setStatus("full-time");
p1.calculateFees();
System.out.println(p1);


p2.setStatus("part-time");
p2.calculateFees();
System.out.println(p2);

u1.setStatus(“full-time”);
u1.calculateFees();
System.out.println(u1);

u2.setCredits(25);
u2.calculateFees();
System.out.println(u2);

u3.setCredits(20);
u3.calculateFees();
System.out.println(u3);
```

21. What happens when you invoke the **calculateFees( )** method on the **Student** objects **s1** and **s2**?
    //nothing will be printed because the method in the Student class was not refined

Lab 5

In this lab, we will explore the polymorphic behaviour of subclass and superclass instances. This lab builds on the concepts of Inheritance, method overriding and replacement.

Part 1: Polymorphism, Method Binding, Principle of Substitutability



1. Create the following instances in the **ShapeRunner** class , invoke the **toString( )** method on them and print the output.

TIP: Invoking a parent method from a child class

Object	Object Type	Features
s1	SimpleShape	
s2	Rectangle	Length = 50, Breadth = 100

```
 SimpleShape s1 = new SimpleShape();
 SimpleShape s2 = new Rectangle(50, 100);

System.out.println(s1.toString());
 System.out.println(s2.toString());
```

2. Modify the **toString( )** method (inherited from the **SimpleShape** class) in the **Rectangle** class so that it prefixes the word “Rectangle” to the String produced in the parent **toString( )** method. Execute **ShapeRunner**, and observe the output.

```
public String toString(){
	String s = "Area: " + String.format("%.2f", area);
	return s; }
//area is now printed but is 0 because no calculation was done
```

3. Modify the **toString( )** method (inherited from the **SimpleShape** class) in the **Circle** class so that it prefixes the word "Circle " to the String produced by the parent **toString( )** method from SimpleShape. Execute **ShapeRunner.**

```
public String toString(){
     return "Circle" + super.toString();
  }
```

4. Create the following instance in the **ShapeRunner** class but declare it to be of type **SimpleShape** and instantiate them as the respective Object type in the table.

TIP: Declaration vs Instantiation

Object	Object Type	Features
s3	Circle	Radius = 50

SimpleShape s3 = new Circle(50);

Object	Object Type	Features
s4	Circle	Radius = 30
s5	Rectangle	Length = 300, Breadth = 100

5. Create the following instances in the **ShapeRunner** class but declare and instantiate them as the respective Object type in the table.

```
Circle s4 = new Circle(30);
Rectangle s5 = new Rectangle(300, 100);
```

6. Invoke the **toString( )** method on the instances print the output. Observe the outcome and identify which **toString( )** method (from the subclass or the superclass) is being called by each instance.

```
System.out.println(s3.toString());
System.out.println(s4.toString());
System.out.println(s5.toString());
```

7. Let's try to reduce the 5 print statements to run in a loop.
   (a) Create an array of 5 **SimpleShape** objects called **shapes**

   ```
   SimpleShape[ ] shapes = new SimpleShape[5];
   ```

   (b) Insert the 5 objects (**s1..s5**) into the array. Did this work? Why?

   ```
   /* e.g. */   shapes[0] = s1;
   ```

```
shapes[0] = s1;
shapes[1] = s2;
shapes[2] = s3;
```

```
shapes[3] = s4;
shapes[4] = s5;
```

(c) Type the following code to iterate through the array and print the details of the objects in the array. This is a different way of writing a `for` loop in Java.

```
for (SimpleShape ss: shapes){
System.out.println(ss.toString());
          }
```

8. Override the **calculateArea( )** methods in the **Rectangle** and **Circle** classes so that the **toString( )** method works more correctly.

```
public void calculateArea(){  // for rectangle
      area = length * breadth;
   }

public void calculateArea(){ //for circle
      area = Math.PI * radius * radius;
   }
```

9. Invoke the **calculateArea( )** method on the instances within the loop from 11(c).

```
ss.calculateArea();
System.out.println(ss.toString());
```

Part 2: Reverse Polymorphism

The **ShapeScreen** class has a method that will render the shapes specified in the array on the Applet window. However, the method requires that all **SimpleShape** objects provide a **draw( )** method that returns a **java.awt.Shape** object.

1. Type the following line of code in the **ShapeRunner**:

```
ShapeScreen screen = new ShapeScreen(shapes); //pass array as param
```

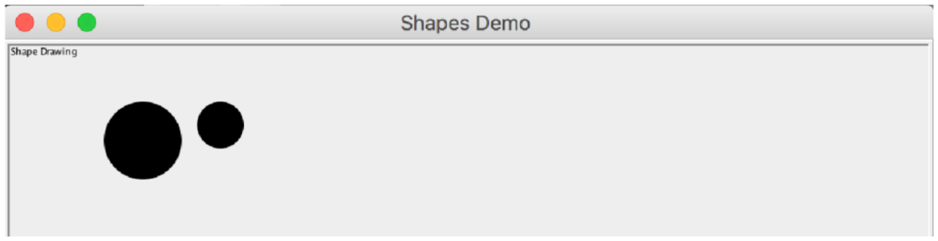
   Observe the Applet window displayed. No shapes are displayed. Why not?

   //because the draw method needs to be overridden

2. How would you override the **draw( )** method in the **Circle** class so that it returns an **Ellipse2D.Double** object with the appropriate dimensions? Examine the constructor of the **Ellipse2D.Double** class, Ellipse2D.Double(double x, double y, double w, double h). It constructs and initialises an **Ellipse2D** object from the specified coordinates.

```
public Ellipse2D.Double draw(){
     return new Ellipse2D.Double(x,y,radius,radius);
  }
```

3. Run the **ShapeRunner** class. You should see the following output if your **draw( )** method works properly in the **Circle** class.

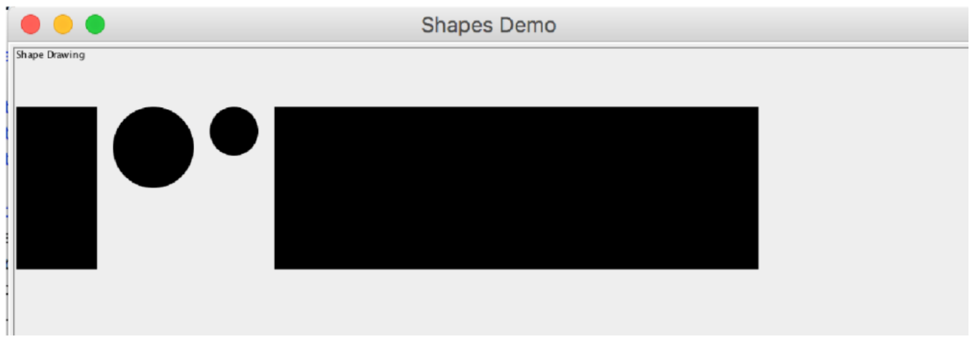


4. How would you override the **draw( )** method in the **Rectangle** class so that it creates and returns a **RoundRectangle2D.Double** object with the appropriate dimensions from the Rectangle class?

   Examine the constructor of the **RoundRectangle2D.Double** class:
   RoundRectangle2D.Double(**double x, double y, double w, double h, double arcw, double arch) .** It constructs and initialises a **RoundRectangle2D** object from the specified `double` coordinates.

```
public RoundRectangle2D.Double draw(){
     return new RoundRectangle2D.Double(x,y, length, breadth, edgeRoundness, edgeRoundness);
  }
```

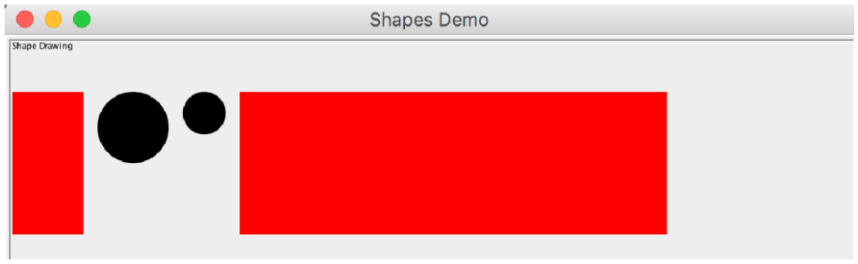
5. Run the **ShapeRunner** class. You should see the following output if your **draw( )** method works properly in the **Rectangle** class.



6. Let's try to change the colours of the **SimpleShape** objects.
   - In a `for` loop, change the colour of the **SimpleShape** objects to red.
   - Use the mutator to set the colour using Color.red as the parameter.
   - Try some other colours for fun.

```
for (SimpleShape ss: shapes){
    ss.setColor(Color.red);

    System.out.println(ss.toString());

    }
```

7. How would you modify your code to generate this colour pattern?



```
for (SimpleShape ss: shapes){

        if (ss instanceof Rectangle){
           Rectangle a = (Rectangle) ss;
           ss.setColor(Color.red);
      } }
```

8. Write a method in the **Rectangle** class called **roundEdge(int curve)** that sets the **edgeRoundness** variable to the incoming value. This would allow us to be change a Rectangle object's edges to rounded.

```
public void roundEdge(int curve){
        this.edgeRoundness = curve;
    }
```

9. Test your **roundEdge( )** method by invoking it on the instances **s2** and **s5** in the **ShapeRunner** class with a curve of **35**.

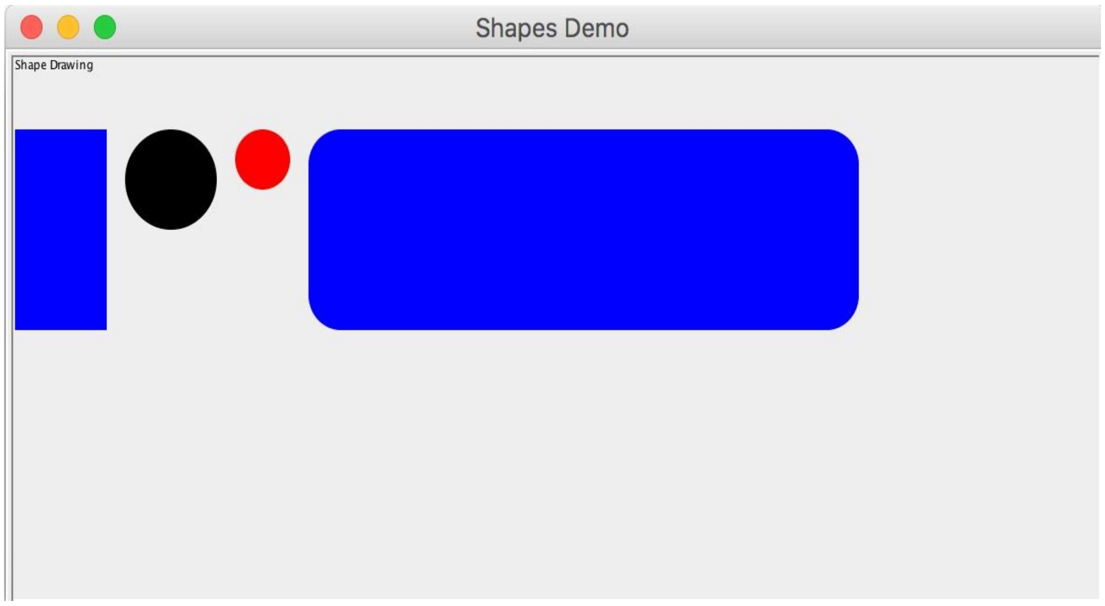
```
s2.roundEdge(35);
  s5.roundEdge(35);
```

10. How can you get your **roundEdge( )** method to work on the **Rectangle** objects in the **shapes** array using a loop? Why do you need to cast here?

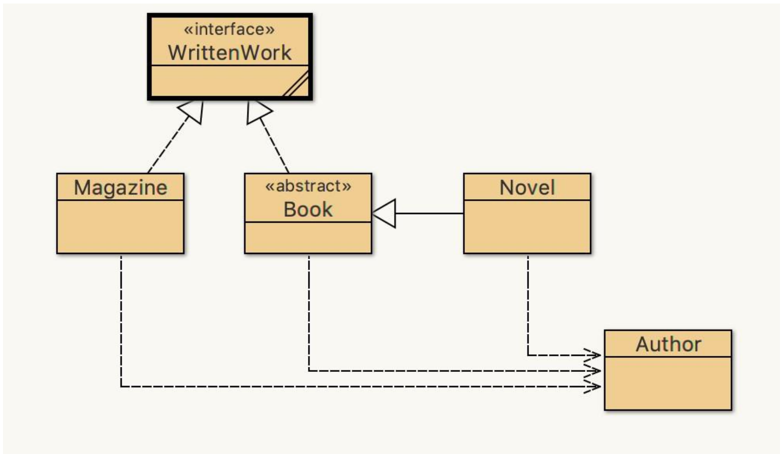
```
 for (SimpleShape ss: shapes){
         ss.calculateArea();
         System.out.println(ss.toString());

         if (ss instanceof Rectangle){
            Rectangle a = (Rectangle) ss;
            a.roundEdge(35);
            ss.setColor(Color.red);

            if (a.getArea() == 30000){
            a.roundEdge(35);
            }
         }
```

11. Try to get your code to generate this colour pattern in the **for** loop for the various shapes

```
for (SimpleShape ss: shapes){
    ss.calculateArea();
    System.out.println(ss.toString());

    if (ss instanceof Rectangle){
       Rectangle a = (Rectangle) ss;
       ss.setColor(Color.blue);

       if (a.getArea() == 30000){
       a.roundEdge(35);
       }}

    if (ss instanceof Circle){
       Circle c = (Circle) ss;
       ss.setColor(Color.black);

       if (c.getRadius() == 30){
       c.setColor(Color.red);
       }
    }
```



Worksheet - Week 6 Part 1: Abstract Classes, Interfaces, ArrayLists



1. Create an interface called **WrittenWork** with the following methods:

   public interface WrittenWork {

   - `public int getNumPages();`
   - `public String getAuthorName();`
   - `public String getTitle();`
   - `public double getPrice();`
   - `public void addAuthor(Author author);` **}**

   Replace the semi-colon symbol from the **getPrice(**) method with curly brackets as for a normal method { }. What do you notice? Why does this happen? Restore the semi-colon when you are finished answering the questions.
   Methods in an interface are abstract by default; no method body; they are subtypes to be defined and implemented.

2. Create a runner class called **BookStore**. Try to create a direct instance of type **WrittenWork**. What do you notice? Why does this happen?

   WrittenWork w = new WrittenWork(); You can't directly instantiate interfaces directly.

3. Create an **abstract** class called **Book**. Try to create a direct instance of type **Book** in the **BookStore** class. What do you notice? Why does this happen?
   Book b = new Book (); You can't directly instantiate abstract classes directly.

4. Modify the class signature of **Book** so that it implements the **WrittenWork** interface. Why is the **Book** class allowed to skip the implementation of the **WrittenWork** interface methods?
   public abstract class Book implements WrittenWork {} ; Only concrete classes need to adhere to implementing method bodies for abstract methods.

5. Remove the **abstract** keyword from the **Book** class signature. Does the class compile? Explain what happened using the terms abstract class, concrete class and interface.
   No, it does not compile. Reason: Concrete classes must provide implementation of methods from the interfaces whereas abstract classes do not.

   i.e an Interface allows a group of unrelated objects to be treated similarly.

6. Fill in the following details of the **Book** class:

Attribute	**Type**	**Purpose**	**Method**
author	Author	The author of the Book	**private Author author;**
title	String	The title of the Book	**private String title;**
numPages	int	The total number of pages in the Book	**private int numPages;**
price	double	The price of the Book	**private double price;**

Method	**Return Type**	**Purpose**	**Method**

Book (String title, int numPages)		Book class constructor	**public Book(String title, int numPages){
			this.title = title;
			this.numPages = numPages;
			}**

addAuthor(Author a)	void	Sets the author of the Book	**public void addAuthor(Author a){
			this.author = author;
			}**

getAuthorName( )	String	Returns the Book author's name	**public String getAuthorName(){
			return this.author.getName();
			}**

getTitle( )	String	Accessor for the title attribute	**public String getTitle(){
			return this.title;
			}**

getNumPages( )	int	Accessor for the numPages attribute	**public int getNumPages(){
			return this.numPages;
			}**

7. Create a new class called **Novel** and make it a subclass of **Book**.
   a. What is the first error flagged by the compiler? Why does this happen?

      public class Novel extends Book {} REASON: it's not abstract and does not override abstract method getPrice() in Written Work.

   b. Fix the error by inserting a getPrice( ) method in the **Novel** class that calculates and returns the price of the Novel where each page costs 0.75 cents.

```
public double getPrice(){
return getNumPages() + 0.75;
}
```

c. What is the second error flagged by the compiler now? Why does this happen?
Constructor in Book class calls two parameters, and it calls super by default.

d. Fix the error by inserting a constructor in the **Novel** class that accepts three parameters (author, title, numPages) and invokes the parent constructor using super(..)

```
public Novel(Author author, String title, int numPages){
    super(title, numPages);
    addAuthor(author);
  }
```

e. Add a t**oString( )** method that returns "NOVEL: " appended with the title, author name, price and number of pages on separate lines.

```
public String toString(){
    String details = "NOVEL : ";
    details += getTitle() + "\n";
    details += getAuthorName() + "\n";
    details += getPrice() + "\n";
    details += getNumPages() + "\n";
    return details;
  }
```

8. Create and associate the following **Author** and **Novel** objects in the **BookStore** runner class:

Object	Declared Type	Features	
a1	Author	Malcolm Gladwell	**Author a1 = new Author ("Malcolm GladWell");**
a2	Author	Steven Johnson	**Author a2 = new Author ("Steven Johnson");**
a3	Author	Mathias Johansson	**Author a3 = new Author ("Mathias Johanssen");**
a4	Author	Evan Ackerman	**Author a4 = new Author ("Evan Ackerman");**
a5	Author	Erico Guizzo	**Author a5 = new Author ("Erico Guizzo");**
a6	Author	Fan Shi	**Author a6 = new Author ("Fan Shi");**
w1	WrittenWork	What the Dog Saw and other	**WrittenWork w1 = new Novel(a1, "What the dog saw and othe**

		adventures, 503 pages, Author: Malcolm Gladwell	
w2	WrittenWork	How We Got to Now: Six Innovations That Made the Modern World, 320 pages Author: Steven Johnson	**WrittenWork w2 = new Novel(a2, "How we got to now", 320);**
w3	WrittenWork	Everything Bad is Good for You: How Today's Popular Culture is Actually Making Us Smarter, 254 pages, Author: Steven Johnson	**WrittenWork w3 = new Novel(a2, "WEverything bad is good f**

Print the details of all **Novel** objects using the **toString( )** method.

System.out.println(w1);

System.out.println(w2);

System.out.println(w3);

9. In the **BookStore** runner class: Create an ArrayList of **WrittenWork** objects called **products**, and add the appropriate objects from Step 10. Example:

ArrayList<WrittenWork> products = new ArrayList<>( );
products.add(w1);

products.add(w2);

products.add(w3);

10. In the **BookStore** runner class: Print the details of the objects in the **products** ArrayList using a for loop:

```
for( WrittenWork w: products)
  System.out.println(w.toString( ) );
```

11. In the **BookStore** runner class: Create an **ArrayList** of **Author** objects called **authors**, and add the appropriate objects from Step 10.

```
ArrayList<Author> authors = new ArrayList<>();

      authors.add(a1);

      authors.add(a2);

      authors.add(a3);

      authors.add(a4);

      authors.add(a5);

      authors.add(a6);
```

12. In the **BookStore** runner class: Print the details of the objects in the **author** ArrayList using a for loop. What do you notice?

```
for (Author p: authors)
System.out.println(p.toString());
```

The attributes are still 0 because it isn't set. Add in the following method below to fix.

13. In the **BookStore** runner class: Write code that would go through the ArrayLists, check whether an author has written a book, and update the appropriate count in the author object. Repeat Step 14 to verify that your code works.

```
for (Author p: authors){

   for (WrittenWork w: products){

      if (p.getName().equals(w.getAuthorName()))

      p.addNewBook();

   }

}
```

14. Create a new concrete class called **Magazine** and make it a subtype of **WrittenWork**. What do you notice?

```
public class Magazine implements WrittenWork{}
```

   a. Introduce one attribute in the **Magazine** class: a **title**

      private String title.

b. Introduce the following methods in the **Magazine** class:

15. In the **Magazine** class:

a. Introduce an **ArrayList** called **authors** that holds **Author** objects. A Magazine can have multiple authors and we want to be able to reflect this.

private ArrayList <Author> authors;

Method	**Return Type**	**Purpose**	**Method**
Magazine (String title)		Magazine class constructor (sets the title)	**public Magazine(String title){ this.title = title; authors = new ArrayList<>(); }**
addAuthor (Author a)	void	Adds an author to the Magazine.	**public void addAuthor(Author a){ }**
getAuthorName( )	String	Returns a list of the names of all authors who contributed to the magazine.	**public String getAuthorName(){ String s = " "; for (Author a: authors) s+= a.getName(); return s; }**
getTitle( )	String	Accessor for the Magazine title attribute	**public String getTitle(){ return title; }**
getNumPages( )	int	Returns the total number of pages in the magazine.	**public int getNumPages(){ return 0; }**
getPrice( )	double	Returns 50.00 (Same price for all Magazines).	**public double getPrice(){ return 50.00; }**

b. Modify the **addAuthor(..)** method so that a new author is added to the authors ArrayList when the method is invoked. (See ArrayList add method in the API).

```
public void addAuthor(Author a){
        this.authors.add(a);
   }
```

Test that your method works by creating a **Magazine** object in the **BookStore** class as follows:

```
WrittenWork mag = new Magazine("IEEE Spectrum”);
products.add(mag);  mag.addAuthor(a3);
```

```
mag.addAuthor(a4); mag.addAuthor(a5);
mag.addAuthor(a6);

System.out.println (mag.getAuthorName());

System.out.println (mag.getNumPages());
```

c. Modify the **getAuthorName( )** method so that a list of all of the authors of a Magazine is returned. Test that your method works by calling the appropriate method in the BookStore class. **SEE TABLE**

d. Modify the **getNumPages( )** method so that the total number of pages in the magazine is determined by the total number of authors where each author contributes 3 pages. (See ArrayList size method in the API). Test that your method works by calling the appropriate method in the BookStore class.

```
public int getNumPages(){

    int size =+ this.authors.size();

   return size*3;

  }
```

16. Modify the WrittenWork interface to include one more method:

```
    public boolean hasAuthor(Author a){

          if (a.getName().equals(this.getAuthorName())

          return true;

 }

       return false;

  }
```

a. What must be done in the classes that implement the interface?
   public boolean hasAuthor();

b. Make changes to the **Novel** class as appropriate.

c. Make changes to the **Magazine** class as appropriate. (See ArrayList contains method in the API).

```
public boolean hasAuthor(Author a){

    if (authors.contains(a))

       return true;
```

```
            return false;
          }
        }
```

17. Verify that your code results in the following for all WrittenWork objects:
    a. Malcolm Gladwell authored a Novel specifically.

```
int maxMag = 0;
        String maxMagName = " ";
        boolean foundSpencer = false;;
        for (WrittenWork w: products){
                if (w instanceof Novel){
                    if(w.getAuthorName().equals("Malcom Gladwell"))
                        System.out.println("Malcom authored a novel");
                }
```

    b. Steven Johnson wrote the largest number of Books in the book store.

```
if (w instanceof Magazine){
                    if(w.getNumPages() > maxMag){
                        maxMag = w.getNumPages();
                        maxMagName = w.getTitle();
                    }
                if (w.getAuthorName().equals("Spencer Johnson")){
                    foundSpencer = true;
                }
            }
        }
        if (foundSpencer == false){
            System.out.println("No WrittenWork objects found for author");
        }
```

    c. IEEE Spectrum has the largest number of Authors in the book store.

```
if (maxMagName.equals("IEE Spectrum")){
            System.out.println("IEEE has the most pages");
        }
```

d. No WrittenWork objects were authored by Spencer Johnson.

```
int max = 0;
        String maxName = " ";

        for (Author a: authors){
            if(a.getNumBooksAuthored() > max){
                max = a.getNumBooksAuthored();
                maxName = a.getName();
            }

        }
        if (maxName.equals("Steven Johnson")) {
                System.out.println("Steven has the most books in the store");
        }

    }
```

Week 7 Lab

Part 1: GUI Elements

1. Modify the labels First Name and Last Name to be Student First Name and Student Last Name.
   private javax.swing.JLabel firstNameLabel;
   private javax.swing.JLabel lastNameLabel;
   firstNameLabel.setText("Student First Name");
   lastNameLabel.setText("Student Last Name");

2. Modify the existing programmes in the combo box list to be: BSc Computer Science (Special) and BSc Information Technology (Special)
   programmeComboBox = new javax.swing.JComboBox<>();
   programmeComboBox.setModel(new javax.swing.DefaultComboBoxModel<>(new String[] {
   "BSc Computer Science(Special)",
   "BSc Information Technology(Special)",
   "BSc Computer Science(Major)",
   "BSc Information Technology(Major)"
   }));

   programmeComboBox.setSelectedItem(2);
3. Add two new programmes to the combo box list: BSc Computer Science (Major), BSc Information Technology (Major).
   SEE ABOVE
4. Set the default display value in the combo box to BSc Computer Science Major.
   programmeComboBox.setSelectedItem(2);

5. Use the ButtonGroup object (buttonGroup1) to group the radio buttons so that either the Full-time or the Part-time option is selected, but not both at once. Neither is selected by default.
   fullTimeRadioButton = new javax.swing.JRadioButton();
   partTimeRadioButton = new javax.swing.JRadioButton();
   buttonGroup1 = new javax.swing.ButtonGroup();
   buttonGroup1.add(fullTimeRadioButton);
   buttonGroup1.add(partTimeRadioButton);
   fullTimeRadioButton.setText("Full-time");
   fullTimeRadioButton.setToolTipText("");
   partTimeRadioButton.setText("Part-time");

Part 2: Simple Event Handling

Some GUI components are used to collect and display data (textfields, text area, combo box, radio button) and others respond to user initiated actions (buttons).

1. In the StudentWindow class, write a method called **clearData( )** that essentially clears all data in the textfields and text area, and resets the values in the combo box and radio buttons to their defaults

```
public void clearData(){
        firstNameField.setText(" ");
        lastNameField.setText(" ");
        buttonGroup1.clearSelection();
        programmeComboBox.setSelectedIndex(2);
        outputTextArea.setText(" ");

    }
```

2. Create a new class **ClearButtonListener** that implements the **ActionListener** interface. An object reference to the **StudentWindow** must be passed into the ClearButtonListener class (e.g via the constructor). The **actionPerformed(..)** method must be overridden and it should invoke the **clearData( )** method.

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class ClearButtonListener implements ActionListener{
    private StudentWindow gui;

    public ClearButtonListener(StudentWindow sw){
        gui = sw;
    }

  public void actionPerformed(ActionEvent e){
        gui.clearData();
    }
}
```

3. In the StudentWindow class, associate the **clearButton** with an instance of the **ClearButtonListener** so that the button's functionality is implemented.

```
clearButton.addActionListener(new ClearButtonListener(this));
```

Part 3: Data Collection via the GUI

Four pieces of data should be collected when the user presses the Register button: first name, last name, programme and status.

1. In the StudentWindow class, write 5 methods that wrap the GUI component accessor/mutator methods:
   - **public String getFirstName(){**
     return firstNameField.getText();
     }
   - **public String getLastName(){**
     return lastNameField.getText();
     }
   - **public String getProgramme(){**
     return programmeComboBox.getSelectedItem().toString();
     }
   - **public String getStatus(){**
     if (partTimeRadioButton.isSelected())
     return partTimeRadioButton.getText();
   - **if (fullTimeRadioButton.isSelected())**
     return fullTimeRadioButton.getText();
     return "none";
     }
   - **public void setOutputText(String output){**
     outputTextArea.setText(output);
     }
2. Create a new class **RegisterButtonListener** that implements the **ActionListener** interface. An object reference to the **StudentWindow** must be passed into the RegisterButtonListener class (e.g via the constructor). The **actionPerformed(..)** method must be overridden and it should invoke the **four** methods created in Step 1 to collect data, clear the existing data (use the clearData( ) method) and then display the data collected (fifth method from Step 1).

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class RegisterButtonListener implements ActionListener{
   private StudentWindow gui;

   public RegisterButtonListener(StudentWindow sw){
      gui = sw;
   }

   public void actionPerformed(ActionEvent e){
      String fn = gui.getFirstName();
```

```
            String ln = gui.getLastName();
            String degree = gui.getProgramme();
            String status = gui.getStatus();

            gui.clearData();

            gui.setOutputText("Registered" +
            fn + " " +
            ln + " " +
            degree + " " +
            status + " " );
        }
    }
```

3. In the StudentWindow class, associate the **registerButton** with an instance of the **RegisterButtonListener** so that the button’s functionality is implemented.
   registerButton.addActionListener(new RegisterButtonListener(this));

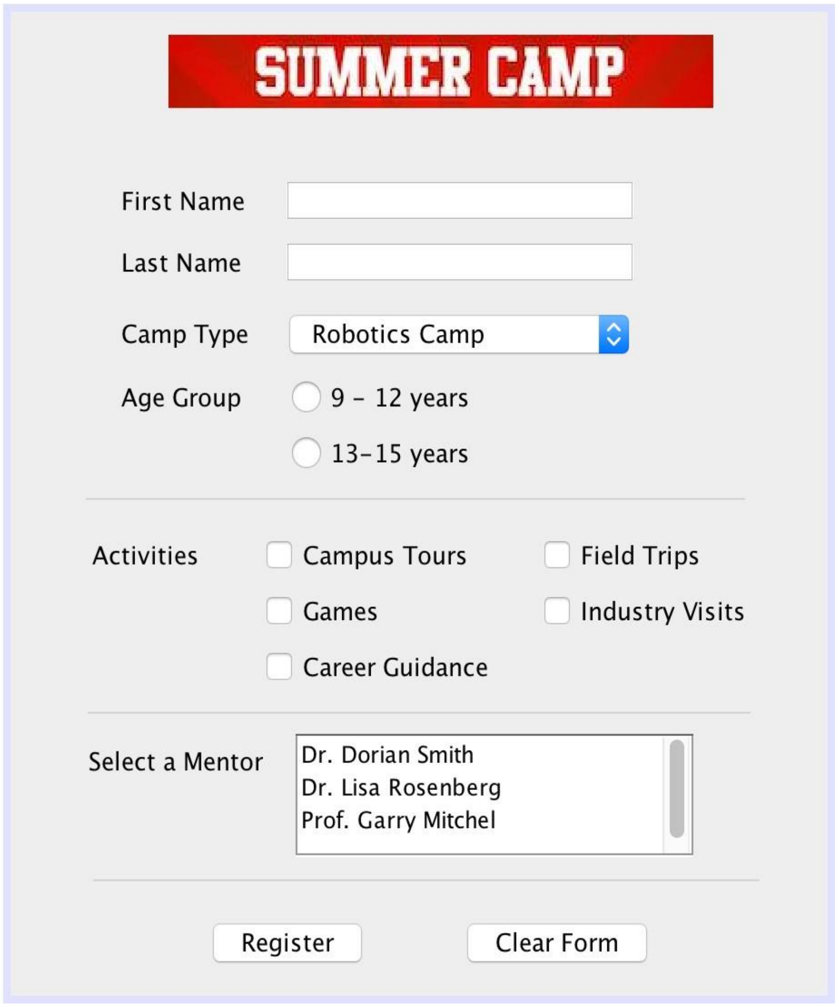


Figure 1

Part 3: Adding Simple Components

1. Switch to the Source View of the Lab8.java class. Type the following code in the main method of the class to generate a GUI instance:

```
SummerCampForm gui = new SummerCampForm();
gui.setVisible(true);
```

Part 5: Adding Mouse Listeners

A MouseListener allows a MouseEvent, initiated by the user's manipulation of the mouse, to be tracked. Examples include: mouseClicked, mouseEntered, mouseExited, mousePressed, mouseReleased.

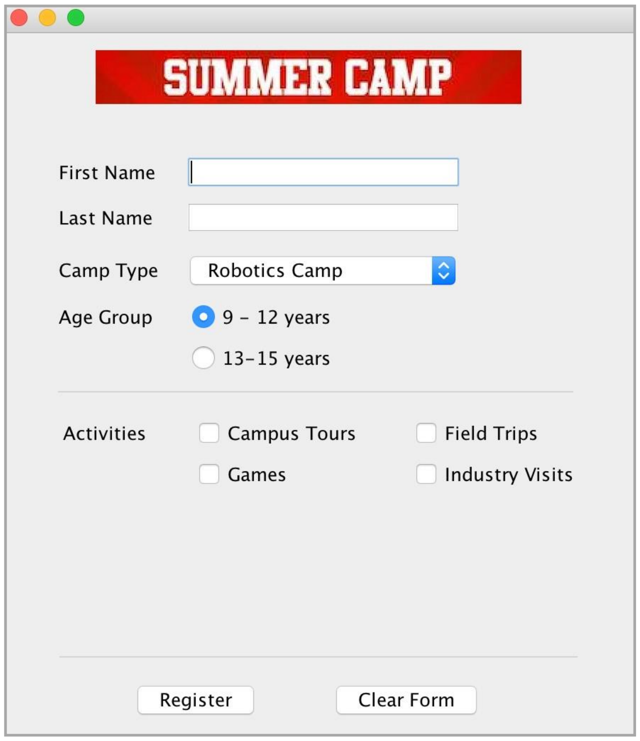


Figure 2: Default Screen

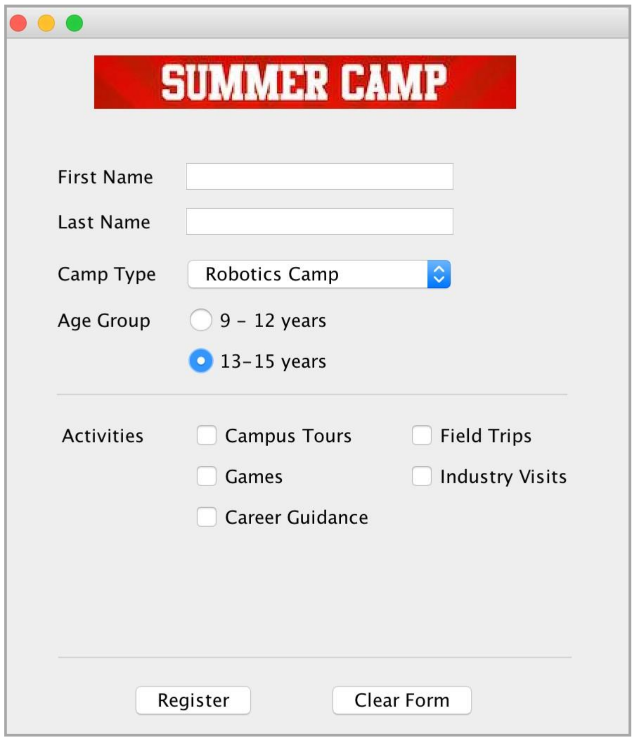


Figure 3: Additional Check-box (Career Guidance) is displayed when 13-15 years Age Group option is selected.

1. Which GUI component would have to be monitored in order to implement the transition from Figure 2 to Figure 3? Which MouseEvent, associated with the GUI component you identified, would trigger the screen transition?

   Answer: **MouseReleased**

2. Switch to Source View of the SummerCampForm class. Write code to make the the Default Screen look as presented in Figure 2 when the GUI is launched.
   - The Career Guidance check box hidden.
     careerGuidance.setVisible(false);
   - The entire mentorPanel (JPanel object) is hidden.
     mentorPanel.setVisible(false);
   - The 9-12 age group is selected by default.
     youngerGroupButton.setSelected(true);
   - The Robotics Camp option is selected by default.

```
 CampTypeCB.setSelected("Robotics Camp");
```

- Write the code to implement the transition from Figure 2 to 3.

```
Private void olderGroupButtonMouseReleased
(java.awt.event.MouseEvent evt) {
If (olderGroupButton.isSelected())
          careerGuidance.setVisible(true);
else
          careerGuidance.setVisible(false);


}
```

3. Write the code to implement the transition shown in Figure 3 to that of Figure 4.

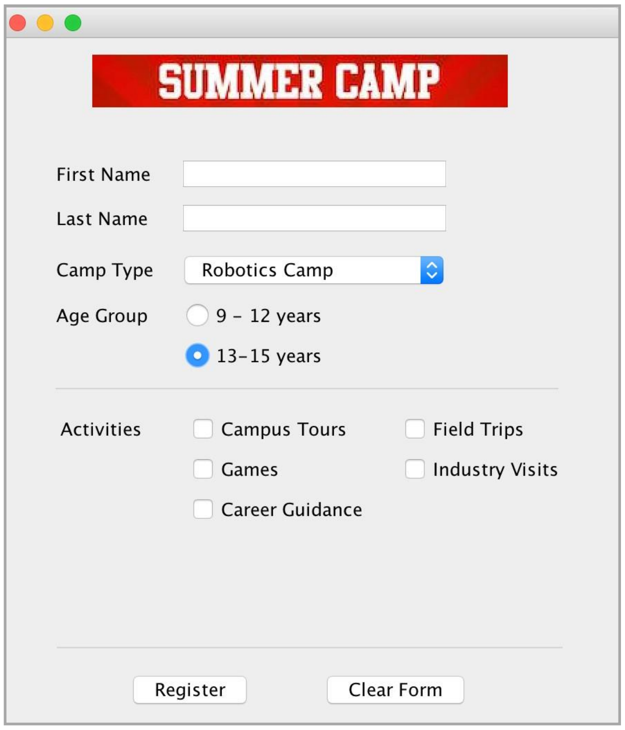


Figure 3: Screen when 13-15 years is selected
(Repeated for convenience from Page 11)

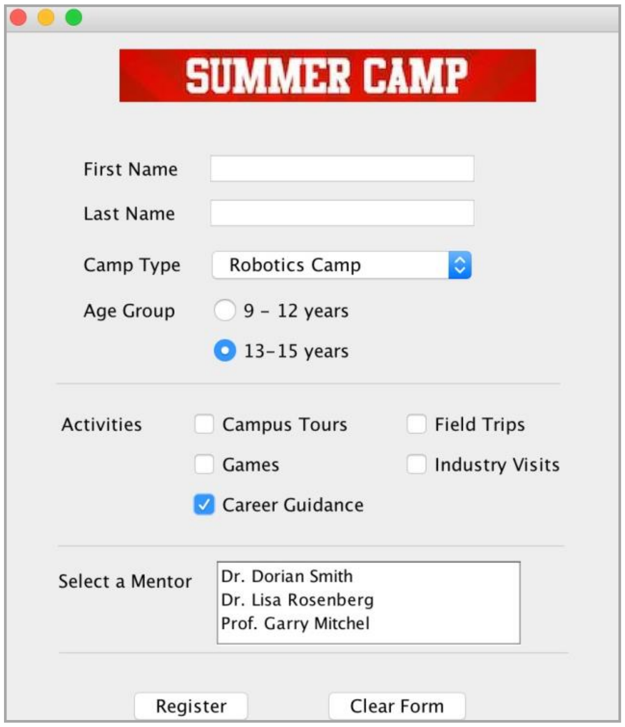


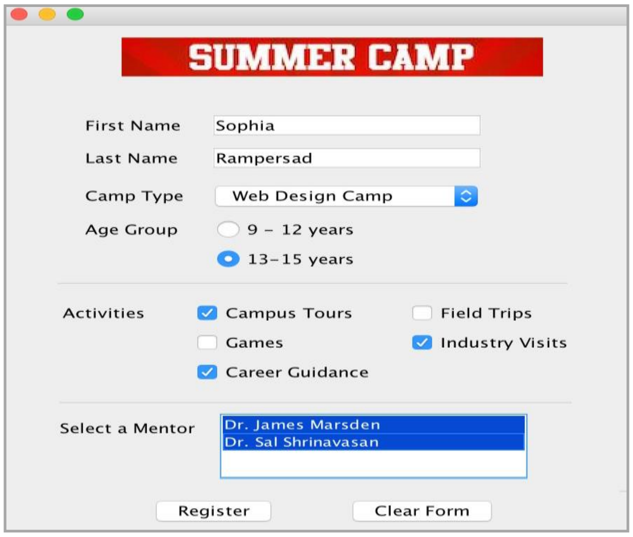
Figure 4: The entire mentorPanel (JPanel) is made visible when the Career Guidance option is selected

Work out the steps to get this done.

Answer: **private void careerGuidanceMouseClicked (java.awt.event.MouseEvent evt) {**

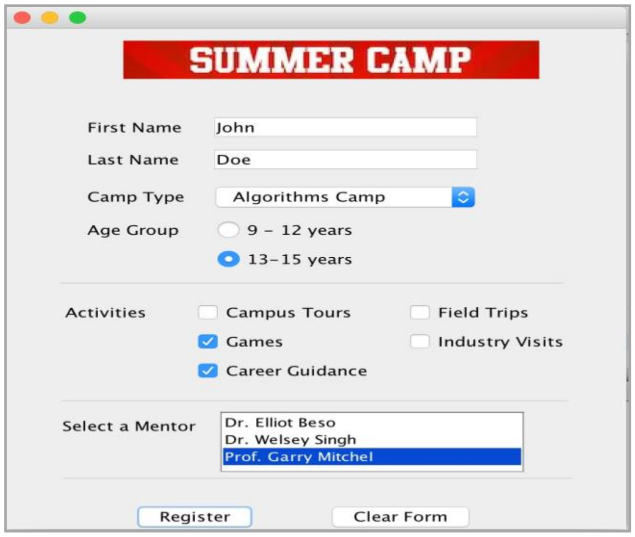
```
if (careerGuidance.isSelected()){
 mentorPanel.setVisible(true);
}
```

5. Write the code to implement the transition shown in Figure 5 and Figure 6 when a different Camp Type is selected.



First Name: Sophia
Last Name: Rampersad
Camp Type: Web Design Camp
Age Group: 9 – 12 years [✓ 13–15 years]
Activities: [✓ Campus Tours] Field Trips Games [✓ Industry Visits] [✓ Career Guidance]
Select a Mentor: Dr. James Marsden, Dr. Sal Shrinavasan

Figure 5: The Mentors for the Web Design Camp are Dr. James Marsden, and Dr. Sal Shrinavasan



First Name: John
Last Name: Doe
Camp Type: Algorithms Camp
Age Group: 9 – 12 years [✓ 13–15 years]
Activities: Campus Tours Field Trips [✓ Games] Industry Visits [✓ Career Guidance]
Select a Mentor: Dr. Elliot Beso, Dr. Welsey Singh, [✓ Prof. Garry Mitchel]

Figure 6: The Mentors for the Algorithms Camp are Dr. Elliot Best, Dr. Welsey Singh and Prof. Garry Mitchel

Which GUI component would have to be monitored in order to implement the transition from Figure 5 to Figure 6? Which ActionEvent, associated with the GUI component you identified, would trigger the screen transition?

Answer: **ActionPerformed on the CampType JComboBox**

Work out the steps to get this done.

Answer:

```
private void jComboBox1ActionPerformed(java.awt.event.ActionEvent evt) {
   String campType = jComboBox1.getSelectedItem().toString();
If (campType.equals(“Web Design Camp”)) {
   mentorBox.setModel (new javax.swing.AbstractListModel<String> () {
   String [ ] strings = {“Dr. James Marsden, Dr. Sal Shrinavasan”);
   public int getSize() {
         return strings.length;}
   public String getElementAt (int i) {return strings[i];});

}

}

If (campType.equals(“Algorithms Camp”)) {
   mentorBox.setModel (new javax.swing.AbstractListModel<String> () {
   String [ ] strings = {“Dr. Elliot Beso”, “Dr.Welsey Singh”, “Prof Garry Mitchel”);
   public int getSize() {
         return strings.length;}
   public String getElementAt (int i) {return strings[i];});

}

If (campType.equals(“Robotics Camp”)) {
   mentorBox.setModel (new javax.swing.AbstractListModel<String> () {
   String [ ] strings = {“Dr. Dorian Smith”, “Dr. Lisa Rosenberg” , “Prof Garry
Mitchel”);
   public int getSize() {
         return strings.length;}
   public String getElementAt (int i) {return strings[i];});

}
```

TIP: Google the Java API of any class to find out more about the methods that

can be used

Part 6: Adding Action Listeners

The GUI should have some functionality when the buttons are clicked, namely, data collection for the Register button and clearing the components and resetting the form for the Clear button.

1. Attach an ActionListener object to the Register Button.

   registerButton.addActionListener(new RegisterButtonListener(this));

2. Write code for the actionPerformed( ) method of the Register Button ActionListener that:

- Collects the data from all of the visible components on the screen.
- Stores the data in variables

```
private void jButton1ActionPerfromed (java.awt.event.ActionEvent evt) {
              String a = jTextField1.getText();
              String b = jTextField2.getText();
              String c = jComboBox1.getSelectedItem().toString;
```

```
            String d = " ";
        if (youngerGroupButton.isSelected())
            d = "9-12";
        if (olderGroupButton.isSelected())
            d = "13-15";
        java.util.ArrayList<String> acts =  new java.util.ArrayList<String< ();
        for (javax.swing.JCheckBox var : activities) {
            if (var.isSelected())
            acts.add(var.getText());
    }
        java.util.ArrayList<String> mentors =  mentorBox.getSelectedValuesList();

    }
```

3. Attach an ActionListener object to the Clear Button.

   clearButton.addActionListener(new ClearButtonListener(this));

4. Write code for the actionPerformed( ) method of the Clear Button ActionListener that:
   - Resets the form to the default presentation of elements (Figure 2)
   - Clears the data from the visible components.

```
java.util.ArrayList<javax.swing.JCheckBox> activities =  new
java.util.ArrayList<javax.swing.JCheckBox>();
public summerCampForm() {
initComponents();
activities.add(jCheckBox1);
activities.add(jCheckBox2);
activities.add(jCheckBox3);
activities.add(jCheckBox4);
activities.add(jCheckBox5);
careerGuidance.setVisible(false);
mentorPanel.setVisible(false);
youngerGroupButton.setSelected(true);
}

private void jButton2ActionPerfromed (java.awt.event.ActionEvent evt) {
jTextField1.setText("");
jTextField2.setText("");
mentorPanel.setVisible(false);
careerGuidance.setVisible(false);
jComboBox1.setSelectedIndex(0);
youngerGroupButton.setSelected(true);
for (javax.swing.JCheckBox a : activities)
    a.setSelected(false);
}
```

Part 7: Connecting the GUI to the Domain Object

1. Create a new Java class called Camper in your Project. This will be used to create objects that store data collected from the GUI.
2. Paste the following code in the class body. Ensure the package declaration (`package lab8;`) to the top of the class is not altered.

```
public class Camper {
private String firstName;
private String lastName;
private String campType;
private String ageGroup;
    private java.util.ArrayList<String> activities;
private java.util.ArrayList<String> mentors;

    public Camper(String fName, String lName, String cType){
        firstName = fName;
lastName = lName;         campType
= cType;

        activities = new java.util.ArrayList<String>();
mentors = new java.util.ArrayList<String>();
    }
    public void setAgeGroup(String ageGp){
        ageGroup = ageGp;
    }

    public void setActivities(java.util.List<String> acts){
     if(acts != null)
         activities = new java.util.ArrayList<String>(acts);

    }
    public void setMentors(java.util.List<String> ments){
    if(ments != null)
         mentors = new java.util.ArrayList<String>(ments);
    }

    public String toString(){
     String s= "";
     s+= firstName + " " + lastName + " " + campType + " " + ageGroup + "\n";
     s+= "Activities: " + activities.toString() + "\n";
s+= "Mentors: " + mentors.toString() + "\n";
     return s;
    } }
```

3. Modify the actionPerformed( ) method for the Register button so that:
   a. All of the data collected from the GUI is used to create a Camper object
   b. The Camper object is inserted into an ArrayList of campers who have registered
   c. The details of all the campers registered so far are printed out to the console

```
        java.util.ArrayList<javax.swing.JCheckBox> activities =  new
        java.util.ArrayList<javax.swing.JCheckBox>();


        java.util.ArrayList<Camper> campers=  new java.util.ArrayList<Camper>();


private void jButton1ActionPerfromed (java.awt.event.ActionEvent evt) {
                String a = jTextField1.getText();
                String b = jTextField2.getText();
                String c = jComboBox1.getSelectedItem().toString;
                String d = " ";
        if (youngerGroupButton.isSelected())
                d = "9-12";
        if (olderGroupButton.isSelected())
                d = "13-15";
        Camper x = new Camper(a,b,c);
        java.util.ArrayList<String> acts =  new java.util.ArrayList<String< ();
        for (javax.swing.JCheckBox var : activities) {
                if (var.isSelected())
                acts.add(var.getText());
}
        x.setActivities(acts);
        java.util.ArrayList<String> mentors =  mentorBox.getSelectedValuesList();

        x.setMentors(mentors);

        campers.add(x);

        for (Camper z : campers);
                System.out.println(z);
}
```

The University of the West Indies, St. Augustine
COMP 2603 Object Oriented Programming 1
2020/2021 Semester 2
Lab 9

In this lab, we will create various Collection objects and examine the Java APIs. Previous topics such as Polymorphism, Inheritance, and Interfaces still apply to this lab.

Part 1: LinkedList

TIP: Use a

about a class

[redacted]

1. Create a new project in BlueJ called **Lab9**. Bring up the **Collection** interface in Google.
2. Create a **Plant.java** class in your project with the following code:

```
public class Plant{     private String name;

    public Plant(String name){
       name = name;
    }

    public String toString(){
return "Plant Name: " + name;
    }
}
```

3. Create a main class called **Greenhouse** in your project. Include the statement **import java.util.*;** at the top of your main class. You will be creating collection objects which require use of this package.

4. In the **Greenhouse** main method:
   a. Create a new **LinkedList** object called **vegetables** which holds **Plant** objects: **Collection <Plant> vegetables = new LinkedList<Plant>( );** What would happen if we omitted <Plant> from this step? Why do we use it then?

      Answer: we would have a linked list that could hold all types of objects, not just Plant objects.

      Is the **vegetables** object polymorphic? How do you know?

      Answer: Yes, it is polymorphic as the static (Collection) is not the same as the dynamic (LinkedList).
   b. Write code to print out the details of the plants in the **vegetables** collection if the collection is not empty. If the collection is empty, then print out “No plants were found in the vegetables collection”.

```
if(vegetables.isEmpty()){
        System.out.println("No plants were found in the vegetable
collection");

    }

    else {

        for(Plant p: vegetables)

            System.out.println(p);

    }

 }
```

c. Create the following **Plant** objects and add them to the **vegetables** collection.

Plant Object	Plant Name
p1	Large Tomato
p2	Small Tomato
p3	Potato

```
//Creating Plants
    Plant p1= new Plant("Large Tomato");
     Plant p2= new Plant("Small Tomato");
     Plant p3= new Plant("Potato");

    //Adding them to the linkedlist
     vegetables.add(p1);
     vegetables.add(p2);
     vegetables.add(p3);
```

d. Print out the details of the **vegetables** collection to verify the objects were added in step (c).
e. Type the following code:

```
if(vegetables.contains(p1) )   // line1
      System.out.println("Large Tomato Plant found");
else
      System.out.println("No Large Tomato Plant found");
```

What was printed? Why?

Answer: It printed "Large Tomato Plant found" and reason for it being because the "vegetables" collection contains the Plant object p1, which represents a large tomato plant.

f. Add **p1** to the **vegetables** collection a second time. Print out the **vegetables** collection. Was **p1** added? What type of **Collection** is **vegetables**? Does it allow duplicate objects?

   Answer: Yes p1 was added to the vegetables collection and the vegetables collection is a Linked list collection and yes it allows duplicate objects.

g. Replace line 1 in step (e) with the following:
   `if(vegetables.contains("Large Tomato Plant") )`
   What happens to the output now? Explain why this happens.

   Answer: the contains method utilizes the .equals method which tests sameness by memory. So therefore, the string "Large Tomato Plant" is not equal to the object p1.

h. Create a new **Plant** object **p4** with its type as "Small Tomato". Add it to the **vegetables** collection and print out the **vegetables** collection. Was **p4** added? Suppose we do not want duplicates in the **vegetables** collection. How can we make this work for a **LinkedList**? What does the **Plant** class need to have?

   Answer: Yes it was added overload the equals method.

5. In the **Plant** class, write an **equals( )** method that checks equality based on the **Plant name**. In other words, two **Plant** objects are equal if they both have the same **name**. The **equals( )** method has the following signature: **public boolean equals(Object obj)**

```
public boolean equals(Object obj){

  if(obj instanceof Plant){

    Plant p = (Plant)obj;

    if(p.name.equals(name)){

      return true;

    }

  }
```

return false;

}

6. Create a new **Plant** object **p5** with its type as "Small Potato". Add it to the **vegetables** collection and print out the **vegetables** collection. Are there still duplicates even though we supplied an equals( ) method to discriminate between objects? Why?

   Answer: yes, because LinkedList accepts duplicates, and we have no method preventing that utilizes our overridden equals.

   A way to avoid is doing the following.

   ```
   If(!vegetables.contains(p1)){

           Vegetables.add(p1);

   }
   ```

7. Create a new **Plant** object **p6** with its type as "Small Potato". Don't add it to the **vegetables** collection though.
   a. Type the following code and run the program:

      ```
      if(vegetables.contains(p6) )   // line1
            System.out.println("Small Potato Plant found");
      else
            System.out.println("No Small Potato Plant found");
      ```

      `Small potato plant found` is printed. Why did this happen?

      Answer: Yes, it is printed. Why? our equals method is comparing the attribute name of the plant class.

   b. Comment off the **equals( )** method in the **Plant** class. Repeat step (a) above. No `Small Potato Plant found` is printed. Why did this happen now?

      Answer: Yes, it is printed because now we are checking equality based on objects.

8. Add p6 to the vegetables collection and print out the collection. You should have:

   ```
   Plant Name: Large Tomato
   Plant Name: Small Tomato
   Plant Name: Potato
   Plant Name: Small Tomato
   Plant Name: Small Potato
   Plant Name: Small Potato
   ```

9. Try to retrieve the 4th element in the **vegetables** collection directly. Is there a method in the Collection interface to make this work?

Answer:

```
LinkedList<Plant> pvegetables= (LinkedList) vegetables;

System.out.println(pvegetables.get(3));
```

10. Cast the **vegetables** collection to its dynamic type. Examine the List interface API.
    a. Identify least three of the methods native to the List interface:

    - **Get**
    - **Add**
    - **Remove**

    b. Try these out on the **vegetables** collection.

```
System.out.println(pvegetables.get(3));

pvegetables.add(4, p1);
pvegetables.remove(4);
System.out.println(pvegetables.size());
```

    c. Create a new **Plant** object **p7** with its type as "Lettuce" . Write code to insert p7 at position 2 but add the element that is originally at position 2 to the end of the **vegetables** collection. Examine the API method set(int index, E element) for this task.

Answer:

```
Plant p7 = new Plant("Lettuce");
Plant pos2 = pvegetables.get(2);
pvegetables.set(2, p7);
pvegetables.add(pos2);
```

The third line of code replaces the element at index 2 of the pvegetables ArrayList with the p7 object. This means that the element previously at index 2 will be replaced with the p7 object.

Finally, the fourth line of code adds the pos2 object (which was retrieved in the second line of code) to the end of the pvegetables ArrayList.

    d. Add three more random Plant objects of your choice.
    e. Remove the first element in the **vegetables** collection and insert it at the middle of the **vegetables** collection.

Plant first= pvegetables.removeFirst();

int size = pvegetables.size();

int middleIndex =size/2;

pvegetables.add(middleIndex, first);

11. Create a new ArrayList that only contains half of elements in the **vegetables** collection. Use the Collection API method for this.

    Answer:

    This is one way of doing since the ArraList has an overloaded constructor:
    `ArrayList(Collection<? extends E> c)`
    Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

    The sublist is entered in the brackets() for parameter.

    ArrayList<Plant> halfVegetales = new ArrayList<Plant>(pvegetables. subList(0, size/2));

    Another way is

    ArrayList<Plant> halfVegetales = new ArrayList<Plant>();
    halfVegetables.addAll(pvegetables. subList(0, size/2));

    ArrayList (Java Platform SE 8) (oracle.com)
    List (Java Platform SE 8) (oracle.com)
    Collection (Java Platform SE 8) (oracle.com)

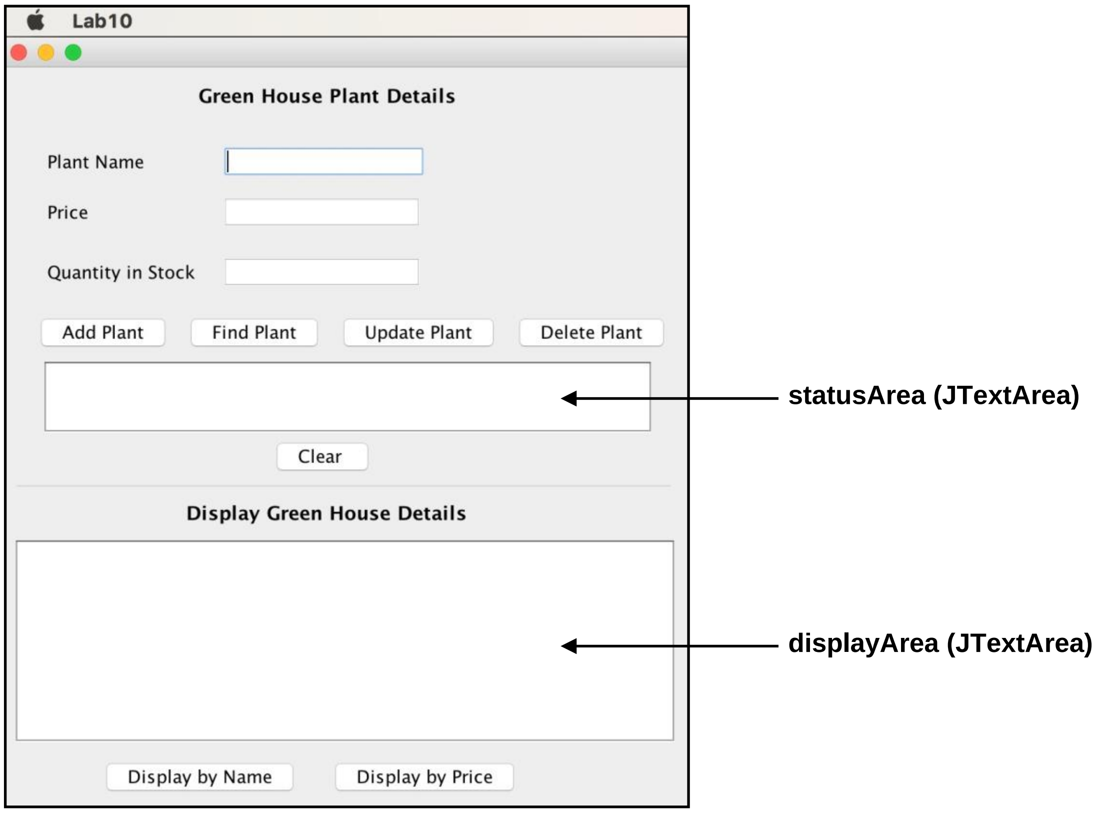
The University of the West Indies, St. Augustine
COMP 2603 Object Oriented Programming 1
2020/2021 Semester 2
Lab 10

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.

   ```
   private Nursery nursery;

   public GreenhouseGUI(Nursery n){
       nursery = n;
       initComponents();
   }
   ```
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.

   Runner class is Lab 10

   BEFORE
   ```
   public static void main(String[] args) {
       GreenhouseGUI gui = new GreenhouseGUI();
       gui.setVisible(true);
   }
   ```
   AFTER
   ```
   public static void main(String[] args) {
       Nursery n = new Nursery();
       GreenhouseGUI gui = new GreenhouseGUI(n);
       gui.setVisible(true);
   }
   ```
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**.

```
public String addPlant(String name, String price, String quantity){
    if(name!=null && price != null && quantity != null){
        int plantQuantity = Integer.parseInt(quantity);
        double plantPrice = Double.parseDouble(price);
        Plant p = new Plant(name, plantPrice, plantQuantity);
        if(plants.add(p))
            return "Plant succesfully added";
    }
    return "Plant "+ name +" not added";
  }
```

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”.

2. Test whether the **Add Plant** button works properly now. Add error checking code as necessary.

```
public String addPlant(String name, String price, String quantity){
    if(name!=null && price != null && quantity != null){
        int plantQuantity = Integer.parseInt(quantity);
```

```
            double plantPrice = Double.parseDouble(price);
            Plant p = new Plant(name, plantPrice, plantQuantity);
            if(plants.add(p))
                return "Plant succesfully added";
        }
        return "Plant "+ name +" not added";
    }
```

Parse is changing the values from a string to an Integer or Double

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: **It returns a to string of the ArrayList basically printing what is in the ArrayList.**

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: **Yes, because ArrayLists allow duplicates.**

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

No, The ArrayList does not automatically sort with that being said we also did no write any functionality to sort it.

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. **Plant p = new Plant(plantName);**
- It then uses the **contains(..)** method of the **Collection** interface to test whether the **Plant** object is in the **Collection**. **if(plants.contains(p)){**
- If the **Plant** object is not in the **Collection**, null is returned. **return null**;
- 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.

```
if(pt.equals(p))
 return pt;
```

FULL CODE

```
private Plant getPlantObject(String plantName){
   Plant p = new Plant(plantName);
   if(plants.contains(p)){
       for(Plant pt: plants){
           if(pt.equals(p))
               return pt;
       }
   }
   return null;
}
```

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

In arraylist get returns element specified positon in the list as of right now we do not know that.

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.

```
/*  This method retrieves a plant object from the collection using a plant
```

```
*   name as the search criteria. If found, the details of the plant are
*   returned as an ArrayList of Strings by invoking the getPlantDetails()
*   method from the Plant class.
*/
public ArrayList<String> getPlant(String plantName){
   if(plantName!=null){
      Plant tempPlant = new Plant(plantName);
      Iterator<Plant> iter = plants.iterator();
      while(iter.hasNext()){
         Plant p = iter.next();
         if(tempPlant.equals(p))
            return p.getPlantDetails();
      }
   }
   return null;
}
```

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.

```
private void findButtonActionPerformed(java.awt.event.ActionEvent evt) {
      // TODO add your handling code here:
      String plantName = plantNameField.getText();
      java.util.ArrayList<String> result = nursery.getPlant(plantName);
      if(result!=null)
      statusArea.setText("Plant details are shown: " + result.toString());

      else
      statusArea.setText("No" + plantName + "found");
  }
```

The purpose of this method is to search for a plant name that the user enters a text field named plantNameField. The method then calls a method named getPlant() on an object named nursery and passes the entered plant name as an argument.

The getPlant() method returns an ArrayList of Strings containing information about the plant. If the getPlant() method returns a non-null result, the method sets the text of a component named statusArea to the string representation of the returned ArrayList.

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?

This approach wasn't used because we wanted to avoid coupling the view should not directly process or know how processing is being taken placed. In favour of this we used the association object which is nursery.

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).**

```
private void findButtonActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    String plantName = plantNameField.getText();
    java.util.ArrayList<String> result = nursery.getPlant(plantName);
    if(result!=null)
    statusArea.setText("Plant details are shown: " + result.toString());

    else
    statusArea.setText("No " + plantName + " plant found");
}
```

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?

The contains would be used to check if a specified element is in the collection and it uses equal to determine if its present. the equals method compares two objects based on their space in memory rather than the name itself.

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

```
public String getName(){
    return this.name;
}

public boolean equals(Object o){
    if(o instanceof Plant){
        Plant p = (Plant)o;
        return this.name.equals(p.getName());
    }
    throw new IllegalArgumentException("Cannot check ueqality with non-Plant
  objects");
}
```

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

It is found.

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

https://www.youtube.com/watch?v=M4f6EqZRqD4

Lab 11

In this lab, we will continue with the GUI and a domain class from Lab 10 using TreeSets and HashSets, and the Comparable interface. You may use either Netbeans or the BlueJ IDE for this lab

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?

   plants = new HashSet <Plant> ();

   //code did not need to be changed in nursery class because hashsets is lower in the hierarchy than collection, it's a subclass of collection

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

Yes, it would work but duplicates would be allowed due the the hashset using the parent hashcode method

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?

   ```
   public int hashCode(){
   return toString().hashCode();
   }
   ```

//observation: no duplicates are allowed

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?

   ```
   public int hashCode(){
   ```

return this.name.hashCode();
}
//observation: it's the same as #3
//these methods must use the same criteria on which to base their functionality for consistency because in general cases hash objects uses hashcode and everything else uses equals

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?

//No the list is not sorted by name because hashsets do not automatically sort and no code was written to sort it

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?

   plants = new TreeSet <Plant> ();
   //can be done because treeSet is a subclass of collection

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.

//An error will occur because in order to use treeset the comparable interface must be implemented therefore plat needs to implement the comparable interface

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

//compareTo() method

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.

```
public int compareTo (Object obj){
if (obj instanceof Plant){
        Plant p = (Plant) obj;
        int result = p.name.compareTo(name);
return result;
}
Throw new IllegalArguementException("Not a plant"); }
```

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?

//the plants were added and displayed in ascending order

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?

```
private class PriceComparator implements Comparator{
        }
//a compareTo method must be provided
```

Write the code for the **compare(..)** method in the **PriceComparator**. 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.

```
public int compare(Object o1, Object o2){
if ((o1 instanceof Plant) && (o2 instanceof Plant)){
Plant p1 = (Plant) o1;
Plant p2 = (Plant) o2;

if (p1.getPrice() < p2.getPrice()) return -1;
if (p1.getPrice() == p2.getPrice()) return -0;
if (p1.getPrice() > p2.getPrice()) return 1;
}
throw new IllegalArguementException ("Objects are not Plants");
}
```

2. 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?

```
public String getPlantsByPrice (){
    if (plants.isEmpty())
    return "No Plants";

    PriceComparator pc = new PriceComparator();
    TreeSet<Plant> plantsByPrice = new TreeSet <Plant> (pc);

    for (Plant p: plants){
        if (plantsByPrice.add(p)){
        System.out.println ("Added");
        }
    return plantsByPrice.toString();
}}
```

3. 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?

//everything still works but its not sorting by price because it was not added to the gui button

4. 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.

   displayArea.setText(this.nursery.getPlantsByPrice);

5. 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 <u>Collections</u> interface for a method that will sort the **ArrayList** using the **PriceComparator**. What is the name of this method and how is it invoked? Add code to the **getPlantsByPrice( )** method based on your answer above.

```
public String getPlantsByPrice (){
    if (plants.isEmpty())
    return "No Plants";

    PriceComparator pc = new PriceComparator();
    ArrayList<Plant> plantsByPrice = new ArrayList <Plant> (plants);
    Collections.sort(plantsByPrice, pc);
    return plantsByPrice.toString();
}}
```

Lab 12

Learning Objectives

- Create and use a HashMap to store, retrieve, remove and update objects
- Create and use a TreeMap to store, retrieve, remove and update objects

Create a new BlueJ Project called Lab12. Add the Java files provided on myElearning to your project.

Part 1: Create Passenger objects

(a) Create a main class, Lab12, containing 5 Passenger objects with the following names:

```
Passenger p1 = new Passenger("Joe");
Passenger p2 = new Passenger("Sid");
Passenger p3 = new Passenger("Lou");
Passenger p4 = new Passenger("Gil");
Passenger p5 = new Passenger("Moe");
```

(b) Add the passengers to a **TreeMap** called **passengers** using the passenger name as the key and the passenger object as the value.

```
TreeMap <String, Passenger> passengers;
  passengers = new TreeMap<String, Passenger>();

  passengers.put(p1.getName(), p1);
  passengers.put(p2.getName(), p2);
  passengers.put(p3.getName(), p3);
  passengers.put(p4.getName(), p4);
  passengers.put(p5.getName(), p5);
```

(c) Print out the <key,value> pairs in the passengers map

```
System.out.println(passengers);
```

(d) Write code to print Gil's the ticket number.

```
Passenger gilObj = passengers.get("Gil");
  Ticket gilTick = gilObj.getTicket();
  String ticketValue = "\n Gil" + gilTick.toString();
  System.out.println(ticketValue);
```

(e) Add the following code to update Sid's name to Syd and retrieve Syd's ticket number.

```
p2.setName("Syd"); //did not change the value in the treeMap
  passengers.remove("Sid");
  passengers.put(p2.getName(),p2);
  Passenger syd = passengers.get("Syd");
  Ticket sydsTicket = syd.getTicket();
  System.out.println(sydsTicket);
```

(i) Why is the error thrown?

(ii) What must be done in order to make the code work? Modify accordingly.

Part 2: Create Vehicle objects

In the Lab12 main class:

(a) Create 5 **Vehicle** objects using the following code

```
   for(int i = 0; i<5; i++){
 Vehicle v = new Vehicle(getRandomNumber(1,20),
getRandomNumber(1,5),
getRandomNumber(1,5),
getRandomNumber(1,5)));
     System.out.println(v); }
```

(b) Edit the code from 2(a) so that 5 **Vehicle** objects are created with the following plateIDs: RLM01, CTJ02, DSC03, MYA04, BTN05. These correspond to objects v1, v2, v3, v4, and v5.

```
Vehicle v1 = new Vehicle ("RLM01", getRandomNumber(1,5),
getRandomNumber(1,5),getRandomNumber(1,5));
Vehicle v2 = new Vehicle ("CTJ02", getRandomNumber(1,5),
getRandomNumber(1,5),getRandomNumber(1,5));
Vehicle v3 = new Vehicle ("DSC03", getRandomNumber(1,5),
getRandomNumber(1,5),getRandomNumber(1,5));
Vehicle v4 = new Vehicle ("MYA04", getRandomNumber(1,5),
getRandomNumber(1,5),getRandomNumber(1,5));
Vehicle v5 = new Vehicle ("BTN05", getRandomNumber(1,5),
getRandomNumber(1,5),getRandomNumber(1,5));
```

(c) Create a **TreeMap** called **vehicles** that stores <Vehicle, Passenger> key-value pairs. Add the following mapping and print out the map.

Vehicle	Passenger
RLM01	P1

```
vehicles.put(v1,p1);
System.out.println(vehicles);
```

(d) Add the following mappings and print out the map.

Vehicle	Passenger
MYA04	P3
CTJ02	P5

```
//adding mappings
   vehicles.put(v4, p3);
   vehicles.put(v2, p5);
   System.out.println(vehicles);
```

What do you notice about the order of the mappings? How is this order achieved?

//It's ordered alphabetically by PlateID

(e) Add the following code after part (d) above.

```
//It's ordered alphabetically by PlateID
  Vehicle v6 = new Vehicle
  ("CTJ02",getRandomNumber(1,5),getRandomNumber(1,5),getRandomNumber(1,5));
   vehicles.put(v6, p2);
   System.out.println("Part 2(e)\n" +vehicles);
```

(i) What do you notice about the contents of the map? Why did this happen?

//Treemaps don't allow duplicate keys
//Plate Id is the key so the old one is replaced.

(ii) Is it possible to have Vehicle object v6 and v2 used as keys in the map?Why or why not?

//No, because it is identical but they can have dup values

Part 3: Store Ticket and Passenger objects in a HashMap

In the Lab12 main class:

(a) Create a **HashMap** called **ticketList** that stores <Ticket, Passenger> key-value pairs

```
HashMap <Ticket, Passenger> tickets = new HashMap <>();
```

(b) Add the five passengers from 1(a) to the HashMap and printout the ticketList contents.

```
tickets.put(p1.getTicket(),p1);
   tickets.put(p2.getTicket(),p2);
   tickets.put(p3.getTicket(),p3);
   tickets.put(p4.getTicket(),p4);
   tickets.put(p5.getTicket(),p5);

   System.out.println (tickets);
```

(c) Create a new Ticket object tx with the ID 100.
(i) Try to insert it into the ticketList for passenger p3 (Lou). Did this work? Why?

```
Ticket tx = new Ticket();
tx.setTicketNumber(100);
tickets.put(tx, p3);
System.out.println("Part 3(c)(i)\n" +tickets);
```

(ii) Try to remove Ticket object with ID 100 as a key from the ticketList using the code below.Did this work? Why or why not?

```
 Ticket ty = new Ticket();
     ty.setTicketNumber(100);
     tickets.remove(ty);
     System.out.println("Part 3(c)(ii)\n" +tickets);
     //failed because ty considered distinct from the others.
```

(iii)Try to determine whether the ticketList contains a key object with ID = 100 using the code below:

```
 Ticket tz = new Ticket();
tz.setTicketNumber(100);
System.out.println("Part 3(c)(iii)\n"+ tickets.containsKey(tx ));
System.out.println("Part 3(c)(iii)\n"+ tickets.containsKey(ty ));
System.out.println("Part 3(c)(iii)\n"+ tickets.containsKey(tz ));

// Did this work? Why not?
//Tx true cuz physically added, Ty recognizes as something distinct
```

(d) Add a hashCode( ) and equals( ) method to the Ticket class where both work using the Ticket ID.

```
 public boolean equals (Object obj) {
     if (obj instanceof Ticket) {
        Ticket p = (Ticket) obj;
        return (p.ticketNumber == this.ticketNumber);
     }

     throw new ClassCastException();

  }

  public int hashCode () {
     String s = ticketNumber + "";
     return s.hashCode();
  }
//
```