ESS 201 Programming in Java
T1 2020-21
Lab 5
9 Sept 2020

Part A (to be worked on during the lab class)
These exercises are on the topics of abstract classes and interfaces. Complete the following exercises to familiarize yourself with additional aspects of inheritance. These need not be submitted, but show your work to the TA before leaving the class.

1. *Abstract classes*
   Redo the exercise in Part 2 of Lab 1 (electric and diesel cars) in Java. Design a base class Car that has an abstract method to find the range. Implement derived classes that would represent electric and diesel cars. Try to design these classes such that the base class has no data members.
   How different is your design here compared to that of Lab 1? You need not implement the complete program, but fill in enough of the classes to convince yourself of the design.

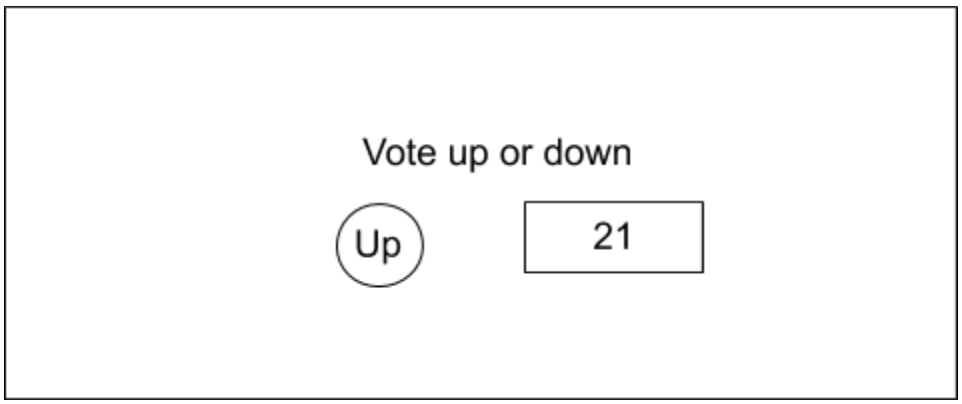
2. *Interfaces*
   An *interface* in Java is effectively an abstract class with no concrete methods, no data members, and all its methods declared as **public abstract**. A derived class of an interface is said to *implement* the interface. Thus, given a class Circle, and an interface Drawable (as in the file Lab5a2.java), create a class DrawableCircle that extends Circle and implements Drawable. Note that while a class can "extend" only one base class, it can "implement" any number of interfaces.

```
class DrawableCircle extends Circle implements Drawable {
 // implement methods of Drawable,
   ...
 // implement/override methods of Circle as needed
  ...
}
```

Part B (to be submitted in one week)

A simple UI consists of widgets like buttons, text boxes, toggle buttons etc. We will implement a very simple UI as an exercise in OOP and also explore what is referred to as “Event driven programming”. This uses inheritance, abstract classes, interfaces etc.

Consider a voting or counting application:



It has a button to show if you are upvoting or downvoting, and another button that you actually press to up or down vote. The second button shows the current vote count
Thus the first button controls the effect of pressing the second button. Above that, we have some fixed text.

While this seems simple, rigging a UI to do this in a flexible way is not straightforward. We will assume a simple UI framework (which is vaguely similar to most UI frameworks) to do this.
The attached zip file Lab5b-v2 has updated skeleton code. You will need to fill these out and modify as needed.

The structure is as follows:
We have a package ***widgets*** that contain the different types of UI elements. We first have some common classes/interfaces:

- **Widget**: parent class of all widgets - buttons, panels, labels etc. Represents a rectangular area of the display
- **Clickable**: interface that can be used to handle mouse clicks. Information about clicks are sent only to classes that implement this interface. Mouse clicks outside the bounds of a widget are ignored by the widget

Then we have some typical UI elements:

- **Panel**: The rectangular area that represents the visual area of the application. It contains a list of child widgets
- **Button**: a rectangular area that can be clicked and displays some text

- **Label**: a rectangular area that displays a string
  For all widgets, we assume that the text is displayed with reference to the lower left corner.

The class **Display** in package ***display*** manages interactions with the underlying system. Its functionality:

- has a list of panels associated with it
- notified Panels when they need to be redrawn (typically after every user action)
- has a list of Clickables registered with it
- receives mouse clicks (which it passes on to the Clickable entities that are registered with it).
- provides methods to draw boxes, circles, text

For our application, called **CountingApp**, we extend buttons to have two kinds

- **Toggle**: which allows a user to toggle between up or down votes
- **Counter**: which keeps track of the number of clicks on the button, but totals based on the state of Toggle at the time the Counter is clicked

The class CountingApp sets up the necessary widgets.

The main creates the instance of Display and CountingApp. After that, to simulate user actions, it reads the display for mouse clicks and sends them to Display.

The input is a series of x y values representing mouse clicks in Display pixel coordinates. For all cases, (0,0) is the lower left corner of the display/widget. An input of 0,0 indicates end of input. You can assume that clicks on the boundary of widgets are ignored.

The sample code has values for the location and size of widgets
So, for this above example:
Sample input

```
270 130
290 140
300 250
170 120
270 140
0 0
```

The final count would be 1, which we would see as the output of the text written out for the Counter.

Expected output (from methods in Display)
Box from 50, 50 to 450, 250
Circle at 175, 135 : radius 25
Text at 150, 110 : Up
Box from 250, 110 to 350, 150
Text at 250, 110 : 0
Text at 200, 200 : Vote up or down
Box from 250, 110 to 350, 150
Text at 250, 110 : 1
Box from 250, 110 to 350, 150
Text at 250, 110 : 2
Circle at 175, 135 : radius 25
Text at 150, 110 : Down
Box from 250, 110 to 350, 150
Text at 250, 110 : 1