CS 141: Introduction to Computer Science
Winter Term, 2013

Lab 5

In this lab, you will learn some of the tools used to build Graphical User Interfaces (GUIs) in Java.

Graphical User Interfaces (GUIs)

Begin by creating a `Lab5` class containing the following:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Lab5 {

    public Lab5() {
        JFrame frame = new JFrame("Lab 5");

        JButton button = new JButton("the button");
        frame.add(button);

        frame.pack();
        frame.setVisible(true);
    }
}
```

This is the general outline for most GUI code. The first 3 lines are outside the class. They tell the compiler that you want to use some libraries that come with Java, but which it doesn't load by default. The code creates a window (in our case a `JFrame`) and then puts a button component in it. In this context, a *component* is an object that either takes input from the user (like the `JButton` in the example above) or displays something to them (we'll see an example of this later). At the bottom of the constructor are a couple of calls that make the window ready to display. The `pack` method arranges all the components and figures out how big the window needs to be in order to fit them. Then `setVisible` makes the window actually display. (You can also call `setVisible(false)` to make a window disappear without ending the program, though we don't do that in this lab.)

Compile and run this program. You should get a little window with a button that doesn't do anything. (You will get a couple of warnings as well, but ignore those.) Not too exciting; kill the program by hitting the "X" button in the top right corner of the Window (on Windows) or typing control-C into the terminal from which you started this program (on Linux or Mac).

To make the button do something, we need to install a *listener*. Listeners are objects that respond to actions such as a button being pressed. They are invoked when the action occurs rather than being called by another part of the program. Programming with listeners is often called *event-driven programming* because actions are also called *events*.

To define our first listener, we implement the `ActionListener` interface, which contains the single method `actionPerformed`. Add the following to your code *inside* the `Lab5` class (but outside its methods):

```
private class QuitListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
       System.exit(0);
    }
}
```

By putting this class inside the `Lab5` class, you are making it an *inner class*. Inner classes cannot be used except from within objects of the class containing them, but have the advantage of being allowed to access the attributes of that class (since they are part of it). When the `actionPerformed` method of a `QuitListener` object is called, it causes the program to exit. Note the argument to `System.exit`; 0 means a "normal exit" and non-zero would indicate that the program exited due to an error.

Once you have the listener, associate it with the button using the following: (in the `Lab5` constructor)

```
button.addActionListener(new QuitListener());
```

This line creates a `QuitListener` object to receive events generated by the button. It is also a good idea to change the button's label (the argument to its constructor) to "Quit". Now, when you compile and run the program, your button actually does something!

What about if we want multiple buttons in our window? Use cut and paste to create a second button object (with a different label) and add it to the frame. When you compile and run this code, the results are disappointing; the new button covered up the old one. That's because we didn't tell Java how to arrange the buttons. To do that, we need to specify a *layout manager*, an object responsible for arranging different components within the frame. Do this by adding the line

```
frame.setLayout(new FlowLayout());
```

immediately after the frame is created (i.e. near the top of the method). When you compile and run the program after this change, the buttons will appear side by side.

Add two more buttons (again, with distinct labels) and see how they are laid out by this layout manager. Then, see how the layout changes as you resize the window.

Next, change the layout manager to `GridLayout` (the constructor for `GridLayout` takes arguments so use `new GridLayout(2,2)`). See how the buttons are laid out and how resizing the window works now. Also, try a 3 by 2 grid (still with only 4 buttons).

The last layout manager we will try is `BorderLayout`. To use this, set the layout manager to a `BorderLayout` object. Then, when you add a button to the frame, include `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, `BorderLayout.WEST`, or `BorderLayout.CENTER` as a second argument to `add`. Assign each button to a different region. Then see what happens, particularly when you resize the window.

There are other layout managers, but more complicated layouts are typically constructed by nesting simpler ones. Use a `JPanel` object to create a nested layout as follows:

```
JPanel panel = new JPanel(new FlowLayout());
panel.add(new JButton("button 5"));
panel.add(new JButton("button 6"));
frame.add(panel, BorderLayout.SOUTH);
```

Note that the `JPanel` itself contains buttons, which are organized by the layout manager established when the panel is constructed, but also serves as a component in the frame. Put this code into the constructor and see how it appears. Try different variations to see how they are drawn.

Once you are ready to move on, it is time to consider output components. We will now look at the `JLabel` component, which is simple but surprisingly useful. Create a `JLabel` by passing a String to its constructor and add it to either `panel` or `frame`. When you compile and run the program, the `JLabel` appears as text in the specified place. Unlike with a button, however, this text cannot be pushed; it is entirely for output. To see how labels can be used this way, make the label an attribute of your class instead of a local variable. Then create a listener class whose `actionPerformed` method calls the label's `setText` method. (Recall that inner classes can touch object attributes). The `setText` method takes a string argument, which becomes the new text displayed by the `JLabel`. Add this listener to one of your buttons and watch what happens when the button is pressed.

For the rest of the lab period, try to create a GUI that looks like a calculator, with a `JLabel` at the top and a grid of numbers and operators below. For this you will need to use multiple layout managers (which ones?) nested within each other. Before writing the code, draw a picture of the GUI you want and then figure out what layout managers you will use.

If you have more time, see if you can make your calculator actually work. A good first step is to have it display the number being typed. What attributes will you need for this? What listeners will you need? (I suggest you use the same type of listener for multiple buttons. For example, all the number keys can use listeners of the same class since they do nearly the same thing. The difference can be accounted for by using a distinct object of that class for each button. By passing an argument to the listener's constructor, you can tell it which button it is listening to.)