CS2110, Learning to Swing

Goals for this lab:

- Get introduced to Swing and event-driven programming.

Pre-lab Activities to Complete:

- Have attended the lecture introducing Swing. If possible, look over MSD textbook, Chapter 12, first 8 pages (or so).

Last updated: October 23 2011

Introduction:

In this lab, we'll begin to explore how to create GUIs for Java programs using the Swing package. Today we'll see the very basics of how this works, and you'll get introduced the these basic concepts:

- Swing components
- Swing containers
- Event-based programming
    - including "listener" classes
- Intro to WindowBuilder GUI development tool that's in Eclipse Indigo

---

Part 1: What you Need for a Swing Program

Here is a very basic program that has a Swing GUI. Look at this code here and read the explanations. But you should also create a new Java project called GuiApp1 and paste this code into a class in that project and run it (so you can run and modify it).

```
import javax.swing.*;

public class GuiApp1 {

    // we'll put Swing components here as fields
    private JFrame frame;
```

```
    // Launch the application in main()
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                GuiApp1 window = new GuiApp1();
                window.frame.setVisible(true);
            }
        });
        System.out.println("main() method exiting!");
    }

    // Initialize the app and call method to setup the GUI
    public GuiApp1() {
        // we could do other non-GUI initialization here if needed
        initialize(); // our method to setup the GUI
    }

    // Create the GUI (the JFrame)
    private void initialize() {
        frame = new JFrame();
        frame.setSize(400, 300);
        frame.setTitle("Demo App#1");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //frame.pack();
    }
}
```

This example contains a lot of things that look like magic right now, but let's explain the basics here:

1. Note we need the import statement. Swing libraries are in "javax", since they were originally "experimental" but afterwards Sun didn't want to require everyone to change the import statements in their code.
2. What's all that nonsense in the main() method? The **important part** right now is to understand that an instance of own class is being created with *new*, which of course calls the class constructor, which calls initialize(). What about that call to the library static method invokeLater() and the creation of the anonymous Runnable class inside run()? There are some subtle bugs you can hit regarding multi-threading that we don't want to explain at this time. **But if you do this for all your Swing programs, i.e. put your code in a method that gets called from main() this way, you'll never have a problem.** (Good news if you use the WindowBuilder tool later on: it generates this code for us.)
3. Note that your class here (GuiApp1) has a member that is a JFrame. This will be the main window of your GUI application. Therefore we can call JFrame methods on the "frame" field object.
4. In initialize() we will modify the JFrame object that is our main "window". All the methods here have "frame." in front of them. We set its title (this could have been done with the constructor). We can set its size (in terms of pixels) by calling the setSize() method on the JFrame object.

5. After we set up the GUI in the JFrame object, we need to make it visible using the setVisible() call. Not only does this make the JFrame appear, but it also starts an **event-handling thread** that runs and listens for GUI events (e.g. button-presses, etc.) that occur in your GUI. More on this later. We could do this at the very end of initialize(), but in this code we do it in main() after creating our "window" app object.
6. One more bit of magic for now. Normally windows go away and program stops when you click on the red-X in top-right of the window or frame. This doesn't happen automatically in Swing unless you tell it to. This call:
   `frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`
   is the simplest way to make this happen. You always do want it to happen, so add this to your code.

OK, so you run that program and what do you see?

- First, notice that the "main() method exiting" message appears in the Console right away -- it's clearly not true that the program is done, because your window is still there. But main() really is done, and the GUI is running in its own thread of execution, and so your program isn't done until all its threads complete. The method in #5 above makes sure this happens.
- Also, notice the program does end completely when the red-X is used to close the window. If you comment out the line of code mentioned in #6 above, your window may go away but the program is still running. You can see that the red "Stop" button in your Eclipse console and know that something is still running. (If you do this, to stop the entire program you must be sure to hit that red "Stop" button.)
- What about the call to pack() that's commented out? Later we'll put things inside our frame, and pack() resets the size so that the enclosing window shrinks down to fit the enclosed items. What do you think happens if you un-comment that call and re-run this program? Make a guess before you try.

Check-off #1: Show the TAs that you've been able to create the program above and run it.

Part 2: Adding Components to a JFrame

We have good news and bad news:

- The good news: It's very simple to add GUI "controls" or "widgets" like buttons and text-fields into your JFrame!
- The bad news: It's much harder to get them organized so they look right all the time (i.e. when the window is re-sized).

So we're not going to worry about getting the **layout** looking good today. Java provides what are called **layout managers**, discussed in the MSD textbook in Section 12.2.3 which we don't expect you to have read for today. Using these is tricky and requires time and trial-and-error at first, so for now we'll not worry if your GUI looks bad. But we have to tell each JFrame *something* about how to do this, so for now we'll add this line of code in initialize():

```
frame.setLayout( new FlowLayout() ); // use the flow-layout
```

Section 12.3 in the MSD textbook lists some common Swing components that can go into your GUI. (In some languages or operating systems, these are referred to as widgets or controls. And in Java, most everything is really a subclass of Component, including frames and windows; see page 810. But let's just focus on these components here.

- JButton
- JLabel
- JTextField and JTextArea
- JMenuBar

If you want one of these in your JFrame, the basic way to do this is very simple:

1. Create the component with new, and pass any appropriate arguments to the constructor.
    - We're going to declare each Swing component object reference as a field of our class, and then call new in initialize(). This is not the only way to organize your code, but that's how we'll do it here. (This is how WindowBuilder will do it when we use it to build our GUIs.)
2. Call any methods on this that set properties (e.g. its color, size, etc.)
3. "Add" it to the container you want your button etc. to be contained in, e.g. the JFrame object, the field called "frame".

See Secton 12.3 of your MSD textbook. Look at the constructors for the first three components listed above and see what you can do when you create them. Also, look at the methods on those two pages to see what methods you can call to change a component's properties. Note: some of these methods use things that are defined in the older AWT GUI library, upon which Swing is based. You may need to add this import at the top of your program:

```
import java.awt.*;
```

What about Step 3 above? How do you add, say, a JButton object to our JFrame object?
(Note: Since Java 5.0, this is easier than before. We'll explain that first and then tell you about the older way that needed the "content pane".) The class JFrame has a method **add(Component c)** that you call to add Component c to that frame. (Where does it put it in the JFrame? Remember we're not worried about that yet.)

So for our example application, since the current object is a JFrame, we just need to put a line of code like this in initialize():

```
frame.add(okButton);
```

Note for Java 1.4 and earlier: Before Java 5.0, you had to get the "content pane" from the JFrame and add to that. The code in the first edition of our book (and most older books) is written this way. WindowBuilder seems to generate code that looks like this too. That way still works, and if you do Swing programming you should be prepared to see it, but for us let's take advantage of the new simpler way.

Let's Do It!

We'll add three GUI components to our application. To do this, add a variable as a private field of our class, and then use new to create each one in initialize(). Do this to get the following (maybe try this one at a time):

- A JLabel object that "Enter password:"
- A JTextField object (that might be used to hold the password)
- A JButton that says "Submit"

After you instantiate each one in initialize(), modify a few properties on these components. Also, don't forget to add them to the JFrame (the frame object)! Also, experiment with the JFrame's size and whether or not pack() is called to see what pack() really does. Also, don't forget what we told you before: add this line of code after we create the JFrame object named jframe:

```
frame.setLayout( new FlowLayout() ); // use the flow-layout
```

Check-off #2: Show the TAs that you've been able to add these components to your GUI application.

Part 3: Responding to Events

Right now you have some GUI components on the screen, but it isn't "live" -- they don't do anything when you click buttons or fill in text fields. You want **an action** to happen when **something** in your GUI **is affected** by the user's actions. If we re-phrased this in programming terms, you want:

- "an action": Some method you've written that will get called to do what you want, when...
- "is affected": something is created known as **an event**, which is a Java object that's generated by...
- "something in your GUI": one of the components you've created and placed in your GUI.

Every component defined in Swing (such as a JButton or a JTextField) has a set of events defined for it that it may generate as something happens to it. (Think of it like this: a JButton suddenly says "I've been clicked!" Or a JTextField announces "Someone's updated the text in me!")

Events are encapsulated as objects in Java, so there is a superclass named Event and many subclasses derived from it. The subclasses are defined for use by particular components. See page Table 12.5 of the MSD textbook.
You write a method sometimes called "an event handler" that will wait for an event to occur, and then process it. Obviously your event-handler needs to be given the Event object that was created and know which GUI component is associated with that event. Then your event-handler method can do what's needed: "OK, so someone pressed the Submit button. I know how to take care of that!"

Before this works, you have to "make the connection" between your method and the GUI component and the particular event it will generate. This is called **registering the event-handler**. So I have to write code that in effect says, "when the JButton object submitButton gets pressed, call my method named handleSubmit(), please". In Java we put these handlers inside classes that are called Listeners.

All this works in the context of a system of programming called **event-driven programming**. In the programs you've seen so far your code runs from start to end (perhaps making some method calls along the way). You have full control of *when* things are done. But in programs like GUIs, your program has to set things up and then wait and see what the user does. There is a thread of execution called the **event-loop thread** which is a separately running executable process-like something. Someone has written some infrastructure code that is part of the Swing library framework (remember that term?) that implements the event-loop.

In event-driven programming, things work like this:

- In your code, you create some component, say a JButton
- Create a object of a "listener" type that will handle events generated by that button.
- Register your listener object with the JButton. ("Hey button! If you're ever pressed, will you please call the event-handler method I've written that's inside this object I'm registering? Thanks!")
- Your code hangs around, waiting for something to happen.
- The user presses the button. The event-loop creates an Event object, and the JButton calls your event-handler method and passes the Event object as a parameter to your method.
- Your method does whatever's needed for this event on this button.

You can see why event-handlers are examples of what are called **callback methods** (or callback routines). You write them, but you don't call them explicitly yourself. Instead, you pass them (somehow) to some other component in the system (e.g. a JButton) and say "please call this sometime". Callback methods always use this idea of registering your method with some other object in the system. (A real-world example of "call back": you might tell your travel-agent to let you know when a flight to Madrid becomes available that costs less than $700. You tell the agent to call you back on the phone when he or she learns that the desired event occurs, i.e. ticket prices drop.)

Part 4: Creating an Event Listener Object

Creating event-listeners in Java can seem complicated. In part because you create a new object (and perhaps a new class) for each pair of GUI-component and Event that it might generate. And components generate different subtypes of the Java Event superclass. See the Table 12.5 in the MSD textbook. Right now let's just think about the JButton class.

The JButton class can be pressed, which generates an Event object of subclass ActionEvent. Java defines an interface for each kind of subclass of Event, and that interface just defines the right method that will be called when that type of event happens. So when a JButton is pressed:

- Some object in a class that implements the ActionListener interface is called. (Which object? We'll explain soon.)
- That interface ActionListener has one method, actionPerformed(), and this is the method that will handle the event.
- The ActionEvent object that got generated when the button was pressed is passed as an argument to the actionPerformed() method.

So what do you have to do?

1. Write some code that defines an object for a class that implements ActionListener.
2. Make sure your code for this new class includes an actionPerformed() method, and...
3. In that method, do what needs to be done, using the Event object if you need it.
4. And, finally, register your object that implements ActionListener with the GUI component.

Let's use an example that illustrates these steps. Do each of the following steps on your example program.

Step 1: Create a new class called SubmitButtonListener and make it implement the ActionListener interface.

Step 2: In that class, have a method with the required signature (see the table in the MSD book):

```
public void actionPerformed(Event e)
```

In this method, do what you want to do when Submit is pressed. Let's just use System.out.println() to write "Submit pressed" to the Console. Note: Eclipse will help you get the right imports needed here, but if you want to add them by hand, put this at the top of your file: import java.awt.event.*;

Step 3: Back where you created the JButton, add this call to register your event handler object:

```
button1.addActionListener( new SubmitButtonListener() );
```

(Note this assumes your JButton object is called button1 -- you may have named it something different.

Note in Step 3 how we call `new` to create an un-named object of the listener type that is passed as a parameter to the addActionListener() method. *Comment for more advanced students.* Remember the slides on Comparators when we used Java's *anonymous class* technique to define a new class "on the fly"? Many people use that same technique to create and register event-handlers in the same place in the code. If you've forgotten what an anonymous class looks like, that's what's being done in main() in this program to create a new anonymous class that implements the Runnable interface and its method run().

Design issue: If you want that SubmitButtonListener object to have access to anything back in the main class (the JFrame class) such as the other components, it needs to have a way to refer to the JFrame class. You could modify the constructor of SubmitButtonListener to take a parameter of the type of your class, and then pass "this" to it when you create a the listener object. The SubmitButtonListener's constructor could store that object-reference value in a field. But we'd still have to make all things available in the JFrame class (GuiApp1) either by changing private things to be public or providing accessors. Yuck. The simplest solution is to use anonymous classes like we did for the demo in lecture. Your TA may explain this technique more.

Check-off #3: Show the TAs your program, and show that the program recognizes the button was pressed. It can do this by simply print a message to the Console.

Check-off #4: If you have time, write a new listener class that recognizes when a user has typed text into the JTextField and then hit "return". Then it should print the string from the text-field to the Console. You'll need to read things in your book or on the Java APIs on-line to find out what's needed here. Here are some hints:

- Inside your event-handler method, you may need to know which Swing component generated the Event object. How could you do that? Maybe there's a method in the Event class that helps you? Check the Java API docs for Event.
- *Cool Eclipse trick to try this when it comes time to add code:* Make the first code you type be the one line of code that tries to register the event-handler, e.g something that looks like this:
  textfield1.addActionListener(new textfield1Listener());
  (Your JTextField object may have a different name.) If you type this first, Eclipse tells you there's an error since you have not yet created the Listener

class. But use the Eclipse "quick-fix" feature and choose the option to create this class. Make sure you check "create inherited abstract methods." Look at how much code it creates for you.

Part 5: Try Out a GUI Development Tool for Eclipse: WindowBuilder

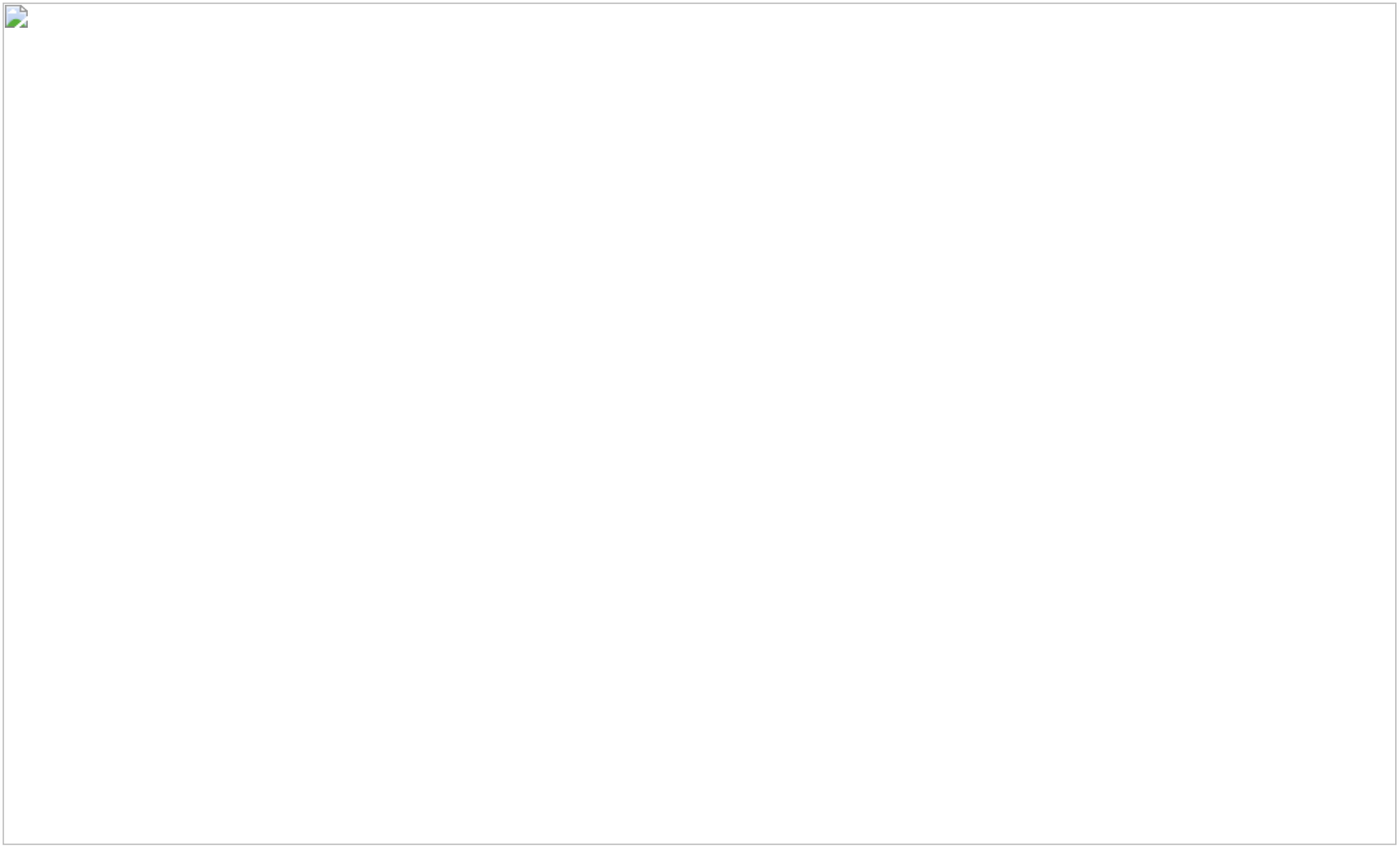
Many languages have tools to help developers create GUI applications, and the latest version of Eclipse has one called WindowBuilder. (It is part of the Indigo release that came out in June 2011.) This tool lets you drag-and-drop GUI components in a WYSIWYG style environment, and you can modify components' properties easily. They also make it much simpler to create event-handling code.

Let's start to build our application again but using WindowBuilder this time. You'll just get a glimpse of how it works today, but if you are working on the GUI for programs in CS2110 later, we suggest you spend some time getting familiar with WindowBuilder.

Close the current project, and start a new one in the usual way ("New Java Project"). Once you have it, do the following to create a new Swing JFrame as your main GUI class.

- From the menu by choose New->Other...->WindowBuilder
- Then choose "Swing Designer" and then "Application Window".
- In the next window, give your main class a name, perhaps GuiApp2.

You will now see code in a window very much like what you saw at the start of this. But at the bottom of that window, you'll see a tab that says "Design". Click that, and you'll now see a more complex set of views within Eclipse appear. One will have a drag-and-drop area where you can place and arrange GUI components. It will help you a lot if you double-click the tab at the top with the name of the code file (e.g. GuiApp2.java) so that you just see the gui-builder WindowBuilder provides you. See the image below:



We'll tell you more about how to use WindowBuilder later. But some important things to keep in mind. Let WindowBuilder have complete control of the initialize() method. This is where it will add code automatically as you drag-and-drop and rearrange GUI compoments. If you change this directly, WindowBuilder may not be able to make sense of your changes and the "link" between drag-and-drop and code-generation may break.

If you want to change properties of components (size, color, etc.), click on a component in the GUI design window, and then use the Properties window (bottom-left window) to see what's there and what can be changed.

To make your new application just like that last one, try doing the following (outside of lab if you're running out of time).

1. When using WindowBuilder, the simplest layout manager to use is called "Absolute". Using this, components stay where you put them and don't adjust when you resize the window. Set the layout manager to be Absolute by clicking on the getContentPane() item in the top-left window (assuming you double-clicked on the GuiApp2.java tab). Use below in Properties click on "Layout" and choose Absolute.
2. In the "Palette" window in the middle, look for the section labeled Components. Here you'll find icons for the three components we added earlier. Drag one of each into your window. You can resize them and reposition them. Use the Properties tab to change their properties (color, size, etc.) if you want.
3. **Important:** Use the properties tab to always rename your components (buttons etc) with meaningful variable names. **Do not** just accept the default names that WindowBuilder gives you.
4. Add event-handlers is *very* nice in WindowBuilder. You don't have to write as much code! Try the following and study the code that's generated:
    A. Right-click the button in the GUI builder, choose "Add Event Handler", then choose "action", then choose "actionPerformed()".
    B. This does two things in your code:
        - It creates a new private class that implements ActionListener with a method actionPerformed() where you can put your event-handler code. (What's a private class? It's a nested class defined inside your current class that's convenient for things like this.)
        - It attaches an ActionListener object to the button.
    C. In the actionPerformed() method, you can add code to do things.

You may not understand all you've just done, but keep these files and study them later. Again, at the very least, you'll see how useful a tool like WindowBuilder can be. But you'll also see that there are many views and windows and it's easy to get lost and confused. But powerful tools are all like that when you first start to use them. If you're going to do a lot of GUI programming, it's worth learning the tool. But we don't expect every student in our class to become GUI experts.

That's it for today's lab! But, on your own, you may want to try some of the following things if you'd like to get better at Swing and GUIs. We would like you to spend some time sometime modifying this project to explore some other things. Here are things you could try:

- See how to create a menu-bar, add menus (like File) to the menu-bar, and add menu-items (like Open, Save, Exit). You can see how to do this in the book. Again, using WindowBuilder makes this easier -- study the code it generated for you.
- Find info in your book(s) or the Java on-line APIs on JDialog, and use this to pop up a simple dialog box when a button is pressed.
- Make an event on one component update another component. For example, when you press a button, text in the JTextField is cleared. To do this, each Listener object needs to have access to the other needed Components. The private classes that WindowBuilder sets up for you makes this pretty easy.

Be sure to turn in your lab check-off sheet before you leave lab!