

The Frightened Freshers Guide To CO520 - Further Object-Oriented Programming (Java)

Thomas Paul Carlson

February 16, 2006

1 Introduction

This document serves the purpose of being a digitalised form of my notes for Further OOP. You should note that right now this document only covers the small section about GUIs and how to start building a GUI. A lot of the material in CO520 is based on the CO320 course so you will definitely need to have a strong base in CO320 before tackling CO520.

You should also note that in its current form this document does NOT cover inheritance, super and subclasses, scope, interfaces...

2 My First GUI Application

In java you can either use AWT or Swing in your GUIs. Swing is basically just AWT with extra fancy new things added to it and as a consequence most applications you use Swing in you will need to use a little bit of AWT.

First off, you do need to import a couple of classes from the Java API to the class you are building the GUI in.

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

Now that you have the relevant import statements sorted out you can begin to build the GUI. The starting point for GUIs is the JFrame. The JFrame is basically just a window on which you add different components. The code below is taken from the lecture slides as an example of initialising a simple window and the comments explain what is going on

```
private JFrame frame;

private void makeFrame()
{
    // Create a new window
```

```

    frame = new JFrame("ImageViewer");
    // Get the content pane of this window
    // The content pane is where most of the
    // GUI stuff you write will end up in
    Container contentPane = frame.getContentPane();
    // Create a JLabel to add to the pane...
    JLabel label = new JLabel("I am a label!");
    // ...and add it to the content pane
    contentPane.add(label);
    // using .pack() will cause the window to figure
    // out just how much space each control will
    // take up on the screen.
    frame.pack();
    // We do want to be able to see it!
    frame.setVisible(true);
}

```

If you typed this out into BlueJ then you would probably not be too impressed with it, but its useful. Really it is! That function makes your first ever GUI, which consists of a window with a little label on that says I am a label. Using this same sort of code you will be able to add menus (Using JMenuBar, JMenu and JMenuItem) as well as other swing controls like scrollbars, listboxes and so on.

Okay, so now you have got yourself a window with a little label on. Great fun. Now how about something useful like a menu bar.

3 Adding a Menu Bar to your application

More code. Note that this is also taken from the lecture slides.

```

private void makeMenuBar(JFrame frame)
{
    JMenuBar menubar = new JMenuBar();
    frame.setJMenuBar(menubar);

    // create the File menu
    JMenu fileMenu = new JMenu("File");
    menubar.add(fileMenu);

    JMenuItem openItem = new JMenuItem("Open");
    fileMenu.add(openItem);

    JMenuItem quitItem = new JMenuItem("Quit");
    fileMenu.add(quitItem);
}

```

From this code you can see that there are a few things going on here. Firstly, a new JMenuBar is created. This is the root from which all other JMenus will stem and is possibly the most important thing here! Note that the frame in frame.setJMenuBar is the same frame used in the previous section. Once you have your base JMenuBar, you can add in a JMenu which is like the File, Edit, View, Tool, Help and so on that you see on a lot of applications. Finally, a JMenuItem must be added so your menus can have some options. So after all this, your application will have a menubar with a File menu, on which would be Open and Quit.

Okay, so you have menus. But what if you want them to do something? The answer, my friend, is not blowing in the wind (Farking xmms...) it is in fact the ActionListener interface.

4 The ActionListener Interface

So what is this magical thing of doom? Well if you are not sure quite what an interface is, its basically something that says to the class that implements it YOU MUST IMPLEMENT THE METHODS I CONTAIN. And thats about it. If you knew this already then thats okay, if not it may be worth flicking over the lecture slides for the lectures about inheritance.

Anyway, onto the ActionListener Interface. It has a single method, actionPerformed. When something happens to an AWT or Swing object, the Class that the GUI was built in is told 'hey, something happened!'. In practice, this means your Class needs to do something when the Quit button is pressed to stop a dumb user from wondering why your application does not quit when there is in fact no quit code...

Once again, the following code is ripped from the slides.

```
public class ImageViewer implements ActionListener
{
    // window creation stuff goes here

    public void actionPerformed( ActionEvent e)
    {
        String command = e.getActionCommand();
        if(command.equals("Open")) {
            // What to do when open is pressed
        }
        else if (command.equals("Quit")) {
            // What to do when quit is pressed
        }
    }

    private void makeMenuBar(JFrame frame)
```

```

    {
        JMenuItem openItem = new JMenuItem("Open");
        // More menu creation code here
        openItem.addActionListener(this);
    }
}

```

So, what on earth is this chunk of code doing? The bit you should be concerning yourself with is the `actionPerformed` section. What this is doing is that it is triggered whenever the Menu items are clicked on and generates an `ActionEvent`. This method grabs the name of the command and then does some simple String comparison to figure out exactly what generated the `ActionEvent` and do stuff as it sees fit. Possibly the most important part of this code comes afterwards with the `openItem.addActionListener` code, this says to the `openItem` ‘Whenever you have an `ActionEvent`, please tell me!’. Note that you will need to add action listeners to all the Menu items and any other clickable objects you may create.

Events generated will correspond to User Interface interactions, such as clicking a button or on a menu item. Frames will generate `WindowEvents` and Menus will generate `ActionEvents`. Objects that are notified when an `ActionEvent` or `WindowEvent` is generated are called Listeners.

The (really) major issue with this style of catching whatever events are generated is that you need additional things in the `if` statement whenever you add more buttons. Also, if you change the captions on your menu items then the `actioncommand` they generated and you will need to once again edit the `if` statement. Basically this method of catching events really sucks unless you are using it for a very, very small application with almost no buttons or menus.

So how about we have a somewhat better example?

5 Nested Class Definitions and non-centralised event handling

As you saw in the previous section, centralised event handling sucks quite a bit. It scales badly and you can only really identify components by text strings(!). Before I get into the nitty-gritty of event handling without stupid ifs...heres a quick commercial break brought to you by a pair of Classes.

```

Public Class Enclosing
{
    // Do some stuff
    // Need a constructor here
    Private Class Inner
    {

```

```

        // Do something here
        // Look! No constructor!
    }
}

```

Wow. Well thats...different. Basically, you are allowed in Java to embed Classes inside of other Classes. These inner classes have a few special properties of their own that you ought to know about. With the inner classes, private fields from its outer class may be used, completely breaking the lovely model you currently have of public/private/protected. You will also note that the inner class has no constructor. This is because it is designed to be used for single-shot anonymous objects that you do not assign to variables or anything of the sort. In fact, (As far as we know so far, at least) the main reason Inner classes were introduced to the Java language is because the centralised method of event handling sucked horribly.

So what the hell does this all have to do with GUIs?! Well, maybe the following code snippet may help. Once again, this is taken from the lecture slides.

```

openItem.addActionListener(
new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        openFile();
    }
}
);

```

Right. Confused? Good. First of all, remember that openItem is a JMenuItem and that the addActionListener method tells java that it should tell this class whenever something happens to this object. Think of this whole thing like openItem.addActionListener(zomglotsofstuffhere); and things get a little bit clearer.

The inner class here (Called ActionListener) has a single method called actionPerformed, as with the awful style of event management. Inside this method everything that needs to be done when this event is fired off is written. This style of event management is quite nice, although I expect with a lot of GUI controls the GUI code could become pretty long. However, with the amount of controls we will be using I do not see this as being a problem.

6 Thanks To...

Code examples ripped from lecture slides for the CO520 course (Copyright David Barnes and Michael Kolling)