A Brief Introduction to Graphical User Interfaces

1. Introduction

A very attractive feature of Java is that the system has a comprehensive library of ready made types of object. To impose some order they're organised in packages. The classes that we use for making Graphical User Interfaces (GUI) are found in the "`java.awt`" package and the "`javax.swing`" packages. To be pedantic about the way in which things are organised, in fact the "`awt`" package is inside the "`java`" package and the "`swing`" package is inside the "`javax`" package.

A graphical user interface usually consists of a window that contains components such as buttons, text boxes, menu bars and so on. In Java all these things are types of objects (no surprise there). It's relatively easy to learn how to build an interface from these objects but it's a bit harder to make it do things. The trick is in understanding how you can tie events like a button being pressed to the execution of a particular piece of java code.

We'll provide a very brief guide to some basic components. The example classes that we refer to may be found in a ZIP file on the Vula assignment page called "`GUITut.zip`".

2. Windows

The first thing we need to be able to do is create a window. There are actually different kinds but for simplicity we'll just go for what's probably the most commonly used. This class of window is a "JFrame". Here's a sample class that demonstrates the use of a JFrame object (Java reserved words are in bold):

```
import javax.swing.JFrame;
//
public class GUICreatorA {

    private GUICreatorA() {}

    public static void main(final String[] args) {
        final JFrame window;

        window = new JFrame("GUICreatorA");
        window.setSize(300, 200);
        window.setLocation(50, 100);

        window.setVisible(true);
    }
}
```

The first line is an import statement that serves to say when the name "JFrame" appears in the text it refers to the JFrame class in the "javax.swing" package.

(The class has an empty-bodied private constructor to prevent GUICreatorA class objects from being created. It's not that sort of class. It simply contains a main method that may be run.)

The important information is in the main method:

1. It declares a variable called 'window'.
2. It creates a JFrame object and assigns it to the field.
3. It says that the window represented by the JFrame object should have a size of 300 pixels by 200 pixels.
4. It says that the window should be positioned on the screen 50 pixels from the left hand side and 100 pixels down from the top.
5. And finally, it says that the window represented by the JFrame object must be made visible – this does not happen automatically.

3. Adding Components

There are a variety of graphical interface components that we might wish to add to our window. Let's say that we'd like a text field below which appears a button. This requires the use of JTextfield and JButton objects. These are also found in the "`javax.swing`" package of components. Here's a new version of our class:

```
import java.awt.BorderLayout;
import java.awt.Container;
//
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;
//
public class GUICreatorB {

    private GUICreatorB() {}

    public static void main(final String[] args) {

        final JFrame window;
        final JButton button;
        final JTextField textField;

        window = new JFrame("GUICreator B");
        window.setLocation(50, 100);

        button = new JButton("Press Here");
        textField = new JTextField(20);

        final Container contentPane = window.getContentPane();
        contentPane.add(button, BorderLayout.SOUTH);
        contentPane.add(textField, BorderLayout.NORTH);
        window.pack();

        window.setVisible(true);
    }
}
```

3.1 Overview

As you can see the class declaration has got significantly larger. The list of import statements has increased. You'll see ones for JButton and JTextField. The rest are needed so that we can state where we want to put these objects in our window.

We'll get the less complicated changes out of the way first. There are declarations for two new local variables, one for the button object and the other for the textfield object, and there are assignment statements for each of these.

- When we create a button, we can specify the legend that will appear on it. We've chosen for our button to display "Press Here".
- When we create a text field, we can specify how wide it should be. We've chosen that it should be 20 columns wide.

3.2 The Content Pane

A JFrame is quite a complex thing. To support the various interface features that programmers might want, the window it represents is understood to be composed of a sandwich of layers known as "panes". We add graphical components to the layer known as the "content pane".

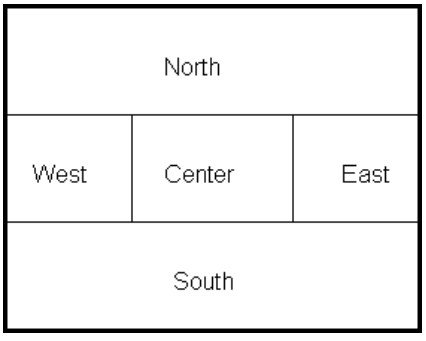
The content pane is thought of as a kind of container for graphical components. In fact it is an object of the class "Container" from the "`java.awt`" package - so that explains another import statement. It also explains the new statement in the constructor that declares a variable "contentPane" and assigns to it the result of performing the "getContentPane" method on the JFrame object. (We think you can guess from the name what the method does.)

Immediately following this statement are the instructions for adding the button and text field to the pane. Each is an invocation of the "add" method. The method accepts two parameters. The first identifies the graphical component to be added, the second states where it should be placed.

3.3 The Layout Manager

Rather than state that a graphical component should be placed 3 pixels right and 10 pixels down from the left hand top corner of the window or whatever, the arrangement of items in a Java graphical interface is controlled by a 'layout manager'.

In the case of a JFrame content pane the arrangement is controlled by a 'BorderLayout' manager. You can request that the manager place your component in one of five sections as illustrated by the following picture.



In the invocations of the add method we indicate that the button should be placed in the south section by using the value `BorderLayout.SOUTH`, and the text field in the north section by using the value `BorderLayout.NORTH`. The use of these BorderLayout values explains the last import statement.

4. Giving the button an effect

Usually when a graphical user interface has a button, pressing that button has some effect. Ours does not do anything much. To change that situation involves introducing more new concepts.

4.1 Overview

The basic requirement is that when the button is pressed it causes some block of program statements to be executed i.e. a block of statements that bring about the desired effect. The means by which this is achieved is in fact quite elegant.

1. We define a special type of object called an ActionListener. This type of object has a method called "`actionPerformed`" that defines the block of statements we want executed.
2. We create one of these objects and give the button a reference to it.

When the button is pressed it invokes the object's "`actionPerformed`" method and this brings about the desired effect.

Whilst we're talking about graphical user interfaces, a software object can also be understood to have an interface. It's the set of methods that can be used to interact with it. An ActionListener is simply an object that has a particular interface: one that includes an "actionPerformed" method.

4.2 Defining a type of ActionListener

Let's say that we want to define a class of object called ButtonListener that is an ActionListener then we need to use the following template.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
//
public class ButtonListener implements ActionListener {

    // Field declarations here

    public ButtonListener(){
          // Constructor statements
    }

    public void actionPerformed(ActionEvent e){
          // Put the statements that we want here.
    }
}
```

This looks like the class declarations that we're familiar with except for the import statements and the additional clause in the first line of the declaration. The "implements ActionListener" class simply states that our class implements or provides an ActionListener interface i.e. it has an `actionPerformed` method.

The actionPerformed method accepts a parameter of type ActionEvent. This convention allows information to be passed about the button press that has occurred. We don't need to consider this parameter stuff at this stage. It suffices that it's part of the actionPerformed method declaration.

4.3 Passing a reference to a button

Step two of the scheme concerns giving the button object a listener reference. This is quite simple. Assuming that we have in fact completed our ButtonListener class, we'd use the following statements in our main method:

```
ButtonListener listener = new ButtonListener()
button.addActionListener(listener);
```

We just create an instance of our ButtonListener class and then pass it as a reference in a call of the button `addActionListener` method. Once we've done that, whenever the button is pressed it calls the `actionPerformed` method of the object that we've given it.

5. A Complete Example

Let us say that we want to adapt our GUICreatorB so that a button press causes the date and time to be displayed in the text field.

5.1 Obtaining the date and time

The current date and time can be obtained by using the Date class from the "`java.util`" package (an import statement will be required). A instance of this class represents the date and time at the point of its creation. We can perform `toString` on a Date object and the effect is to return a string representation of the date and time. For example,

```
//...
Date dateAndTime = new Date();
String stringRep = dateAndTime.toString();
System.out.println(stringRep);
//...
```

Executing this (as part of a suitable class declaration) produce output like:

```
Wed Nov 09 14:29:44 GMT 2005
```

5.2 Building a ButtonListener class

Now we define an ActionListener that can use the Date code to place the current date and time in the GUI text field:

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
//
import javax.swing.JTextField;
//
import java.util.Date;
//
public class ButtonListener implements ActionListener {

    private JTextField textField;

    public ButtonListener(final JTextField field) {
        this.textField = field;
    }

    public void actionPerformed(ActionEvent e) {
        final Date dateAndTime = new Date();
        final String text = dateAndTime.toString();
        textField.setText(text);
    }
}
```

The class is *slightly* more complicated than section 4.2 might suggest. The problem is that the ButtonListener needs a reference to the JTextField that it is supposed to manipulate. To this end, a ButtonListener has a 'textField' instance variable and a constructor that accepts a JTextField reference as a parameter.

The `actionPerformed` method uses the `setText` method to set the content of the JTextfield to the current date and time.

5.3 Putting it all together

Combining all our new bits (ButtonListener, adding listener to button, obtaining date and time) we can construct a complete class declaration. We've called it GUICreatorC.

```
import java.awt.BorderLayout;
import java.awt.Container;
//
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;
//
import javax.swing.WindowConstants;
//
public class GUICreatorC {

    private GUICreatorC() {}

    public static void main(final String[] args) {

        final JFrame window;
        final JButton button;
        final JTextField textField;

        window = new JFrame("GUICreator C");
        window.setLocation(50, 100);

        textField = new JTextField(20);
        textField.setEditable(false);

        button = new JButton("Press Here");
        button.addActionListener(new ButtonListener(textField));

        final Container contentPane = window.getContentPane();
        contentPane.add(button, BorderLayout.SOUTH);
        contentPane.add(textField, BorderLayout.NORTH);
        window.pack();
        window.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);

        window.setVisible(true);
    }
}
```

Since we're only using the text field to display the date and time, we don't want it to be possible for the user to click on it and type in something. Therefore we've added a line to the constructor to say it is not editable (by the user).

Finally, what should happen when the program user closes our applications window? We've decided that this should cause the program to terminate. To do this we've added the following statement to the constructor.

```
window.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
```

There are a number of constants defined in WindowConstants (imported from `javax.swing`) that may be used to specify what should happen when a window's close button is pressed. We've used "`WindowsConstants.EXIT_ON_CLOSE`" to indicate we want the program to terminate.

END