Writing a Simple GUI

Second Year Computing Laboratory
Department of Computing
Imperial College

A Summary

These notes are meant to provide information for people working on a simple Java program with a GUI and cover writing of applications and applets, layout of a screen, handing of events, and do not cover drawing graphics, printing, menus and dialogs, or structured screen objects such as trees, lists and tables.

Sun recommends that the `Swing` library, part of JFC (`Java Foundation Classes`) should be used for writing graphical user interfaces rather than the original graphical interface library AWT. `Swing` classes usually have names starting with 'J' to distinguish them from AWT classes. Swing does use constants and methods from AWT but mixing swing and AWT screen components will not work properly. Look at Sun's online Swing tutorial: [http://java.sun.com/docs/books/tutorial/uiswing/] for a complete tutorial on using `Swing` with examples.

Information about all standard Java classes, interfaces and methods are found in the local copy of Sun's api page [http://www.doc.ic.ac.uk/csg-old/java/jdk6docs/api/].

SWT the “Standard Widget Toolkit” is the library used for the Eclipse interface. Although eclipse is implemented with SWT it can be used for developing applications using `Swing`.

Applications and Applets

Applications

An application is a self-contained program which is usually run by typing a command, a Java application must have a public method called `main` in one of the public classes, the application starts when `main` is called. An application needs to create a new window on the screen, usually using the `JFrame` class, first configuring the window and placing other swing items on it and then making it visible. When the user closes the main window the window is hidden, use the method `setDefaultCloseOperation` to specify what is to happen when the main window is closed.

Applets

An applet is a java program interacting with a web page, the applet runs on the client's machine on a browser. Running an applet in the client's machine reduces network traffic and delay. Applets are restricted so that users need not worry about running code with an unidentified author originating from an unknown web site on their browser.

An applet is loaded by a browser which controls it. All applets have to be subclasses of the `Applet` class. To make use of the Swing library you should use a subclass of the `JApplet` class (which is itself a subclass of `Applet`). Instead of defining the method `main`, `JApplet` defines several methods which are called by the browser during the life of an applet, including:

init()	is called when the applet is first loaded.
stop()	is called when the applet becomes invisible, for example when it is behind another window.
start()	is called after `init()` and each time the applet becomes visible.

These methods are implement in `JApplet` as do-nothing methods, Most applets override `init()` with code to initialise the applet and `start()` to start it running. Don't call `System.exit` to halt an applet, it ought not do anything, but if it does the browser running it would probably quit.

An applet uses the same Swing classes as an application, but the applet can access part of the browser's screen. A applet which extends the `JApplet` class uses the method `getContentPane()` to access the `Container` which is displayed by the browser. An application usually creates a window by creating `JFrame`. The application uses `getContentPane()` to access the top level container. One of the first methods that must be used is `setSize` to set the window size. While it is created the window is not visible, items can be added to it and when the layout is complete `setVisible(true)` makes it seen.

The GUI

Constructing the Interface

A Java graphical interface is built using objects from subclasses of the class `JComponent`. Simple components have no subcomponents, for example a `JLabel` shows text or an icon on an area of the screen. The text and the icon are usually specified with the constructor and can be changed by the program as it runs.

Components which are subclasses of the abstract class `Container` can contain other components. When a container is added to the screen the components it contains are drawn inside the frame. The first example shows a `JLabel` placed inside the program's container:

```
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingConstants;


class Example1Gui {
    void example() {
        JFrame examp = new JFrame("A very simple gui");
        JLabel onlyComponent = new JLabel("0",SwingConstants.CENTER);❶

        examp.getContentPane().add(onlyComponent);                    ❷
        examp.setSize(100,100);
        examp.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);         ❸
        examp.setVisible(true);
    }
}

public class Example1  {

    public static void main(String [] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {       ❹
            public void run() {
                Example1Gui gui = new Example1Gui();
                gui.example();
            }
        });
    }
}
```

❶ A label with "0" in the centre of the label is created when the applet is created.

❷
- The method `getContentPane()` returns the place where items are added to the frame.
- The `add` method puts the label inside the container and the label is displayed when the container is drawn.

❸ The program exits when the window is closed.

❹ Starts the correct thread.

Swing and Threads

A gui needs to run at the same as the main program. For example the stop button must be able to stop the main program at any time. Swing runs on its own Java thread to run in parallel with the program that started it. To start Swing properly, use the `invokeLater` method shown in the first example.

Layout

Each container has a layout manager which controls the layout of the components inside it. When components are added or removed from a container or when it is resized, the container is redrawn according to rules of the layout manager. There are several different layout manager implementing the `LayoutManager interface`, and a container's default layout manager can be changed using the `setLayout` method.

The available layout managers can be split into simple ones such as `BorderLayout`, `FlowLayout` and `GridLayout` which provide commonly used layouts, more flexible layout managers like `GridBagLayout` require more work to use and special purpose layout managers such as `ScrollPaneLayout`. As a last resort it is possible to implement your own layout manager or set the layout manager of a container to `null` and manually control the container's appearance.

A simple layout involves picking the right layout manager and setting any parameters it requires. For example the default layout manager for an applet or `JFrame`is the `BorderLayout` layout manager. This arranges up to four components around the border of the container and one in the middle as shown here:

```
import java.awt.Container;
import java.awt.BorderLayout;
import javax.swing.*;

class Example2Gui {

    public void example2() {

        JLabel  aLabel   = new JLabel("0",SwingConstants.CENTER);
        JButton increase = new JButton("+1");                          ❶

        JFrame frame = new JFrame();
        frame.setSize(200,200);
        Container contentPane =  frame.getContentPane();
        contentPane.add(aLabel,BorderLayout.NORTH);                    ❷
        contentPane.add(increase,BorderLayout.CENTER);
        frame.setVisible(true);
    }

}
```

❶ Here we add a button, but leave the handling of pressing of the button for later.
❷ The second argument to `add` tells the layout manager that the label goes at the top (north) of the container.

The code to start the program are only shown in the first example.

More complicated layout

More complicated layout can be done by overriding the default layout manager with a more powerful layout manager like `GridBagLayout` which is very flexible but quite complex to use. A simpler alternative is to split the interface into areas, implement them as separate containers each with their own layout managers. For example:

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

class Example3Gui {

    public void example3() {
        JLabel  result = new JLabel("0",SwingConstants.CENTER);
        JPanel  square = new JPanel();                                  ❶
        JButton increase = new JButton("+1");
        JButton decrease = new JButton("-1");
        JButton red = new JButton("red");
        JButton clear = new JButton("blank");
        JFrame frame = new JFrame();

        frame.setSize(200,200);
        Container contentPane =  frame.getContentPane();
        square.setLayout(new GridLayout(2,2));                          ❷
        square.add(increase);                                           ❸
        square.add(decrease);
        square.add(red);
        square.add(clear);
        red.setBackground(Color.RED);

        contentPane.add(result,BorderLayout.NORTH);
        contentPane.add(square,BorderLayout.CENTER);                    ❹
        frame.setVisible(true);
    }
}
```

❶ To replace the button with a square array of related buttons, create a `JPanel` to contain them.
❷ The default layout manager for the `JPanel` square is replaced with `GridLayout` for a 2 by 2 grid.
❸ Four buttons are created and put inside the square.
❹ The button in the previous example is replaced by the `JPanel`

A layout manager definition does not define what will happen when it is not used correctly. For example when an extra item is added to grid it is not drawn on the square.

Changing the screen

The screen may be changed while the program runs, using `set` methods of a component (such as a `JButton`) to change its like color (background or foreground), border and icons.

A component like a `JPanel` which can also be removed from the container it was in and then replaced by another component. After changing an competent in a container, the container should redrawn using `validate` so the screen is updated.

The Interface to the program

A program interfacing with a keyboard and terminal usually has a main program loop which reads input and produces output, but a program with a gui needs a different control structure because the interface can detect input from more than one source. The example above has four buttons which can be pressed in any order. Libraries that support gui like `Swing` or `gtk` are event-driven. An input action such as a mouse-press is handled by the library which looks to see if the action is over an active area of the screen like a button or menu. Then the library looks to see if any methods (listeners)

are registered as handling the action and if there are it generates an `Event` and calls each listener with the Event as an argument. A listener can be registered for different events and more than one listener can be listening for the same event.

Java Events

Java events generated are usually subclasses of `java.awt.AWTEvent`, and contain information: the 'source' (the object where the event is generated). Different events classes are defined for different input objects for example `mouse event` is generated when the mouse is used and has information about which mouse button was pressed or released.

Java Event listeners

A piece of code that needs to know if a particular event happens should implement an extension of `event listener` such as `mouse listener`. Active components like buttons have methods to add listeners, for example:

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

class Example4Gui {
    int value = 0;
    JLabel result = new JLabel("0",SwingConstants.CENTER);

    public   void example4() {
        JButton increase = new JButton("+1");
        JButton decrease = new JButton("-1");
        JButton red = new JButton("red");
        JButton blank = new JButton("clear");
        JPanel square = new JPanel();
        JFrame frame = new JFrame();
        frame.setSize(300,300);
        Container contentPane =  frame.getContentPane();
        ExampleListener listen = new ExampleListener();                    ❶

        red.setBackground(Color.RED);
        square.setLayout(new GridLayout(2,2));
        square.add(increase);
        square.add(decrease);
        square.add(red);
        square.add(blank);

        contentPane.add(result,BorderLayout.NORTH);
        contentPane.add(square,BorderLayout.CENTER);

        increase.addActionListener(listen);                                ❷
        decrease.addActionListener(listen);
        red.addActionListener(listen);
        blank.addActionListener(listen);
        frame.setVisible(true);
    }
    class ExampleListener implements ActionListener {                      ❸

        public  void actionPerformed(ActionEvent event) {                  ❹
           // the button pressed
            JButton sent = (JButton)event.getSource();
           // the button's label
            String label = sent.getText();
```

```
            // Use the button's text to select the action
            /*
               Write separate listeners if the buttons are not
               related or the actions are not short
            */
            if (label.equals("+1")) {
                value++;
            } else if (label.equals("-1")) {
                value--;
            } else if (label.equals("red")) {
                result.setForeground(Color.RED);
            } else {
                value=0;
                result.setForeground(Color.BLACK);
            }
            // update the value shown in the label
            result.setText(Integer.toString(value));
        }
    }
}
```

❶ One listener is created for all the buttons, an ActionEventListener is used because the program doesn't need to know which mouse button was pressed.

❷ The listener is added to the buttons.

❸ The class implementing the action listener is defined inside the Example4Gui class to be able to access its fields (*Inner classes*).

❹ `actionPerformed` is called when any of the components with this listener are clicked. The event generated is passed as the argument `event`. An Event contains a reference to the component that generated it. event.getSource() is the button that was clicked.

It if often the case that at some time while a program is running that pressing some of the buttons should not have any effect. The recommended method is to call `setEnabled` with argument `false` to disable the button, and with `true` to readable it.

Example 5

Extends the previous example to use icons and test which mouse button is used.

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

class Example5Gui  {
    int value = 0;
    int rows =2 ;
    Class myClass = Example5Gui.class;                                    ❶
    JLabel result = new JLabel("0",SwingConstants.CENTER);
    ImageIcon downButtonIcon,upButtonIcon ;

    public void  example5() {
        upButtonIcon =   new ImageIcon(myClass.getResource("Up.png"))❷;
        downButtonIcon = new ImageIcon(myClass.getResource("Down.png"));
        JButton up = new JButton();
        JFrame frame = new JFrame();
```

```
    Container contentPane =  frame.getContentPane();
    rows = 5;
    frame.setSize(300,300);
    JButton plusMinus = new JButton("+/-");
    JButton red = new JButton("red");
    JButton blank = new JButton("clear");
    ExampleListener listen = new ExampleListener();
    ExampleMouseAdapter mouseListen = new ExampleMouseAdapter();
    red.setBackground(Color.red);
    JPanel square = new JPanel(new GridLayout(rows,rows));
    square.add(plusMinus);
    square.add(red);
    square.add(up);
    square.add(blank);

    contentPane.add(result,BorderLayout.NORTH);
    contentPane.add(square,BorderLayout.CENTER);

    plusMinus.addMouseListener(mouseListen);
    red.addActionListener(listen);
    up.addActionListener(listen);
    blank.addActionListener(listen);
    up.setIcon(upButtonIcon);                                    ❸
    frame.setVisible(true);
}

class ExampleListener implements ActionListener {

    public void actionPerformed(ActionEvent event) {
         // the button pressed
        JButton sent = (JButton)event.getSource();
        // the button's label
        String label = sent.getText();

        if (label.equals("red")) {
            result.setForeground(Color.red);
        } else if (label.equals("clear")) {
            value=0;
            result.setForeground(Color.black);
        } else if ( sent.getIcon() == upButtonIcon) {            ❹
            sent.setIcon(downButtonIcon);
        } else {
            sent.setIcon(upButtonIcon);
        }
        result.setText(Integer.toString(value));
    }
}


class ExampleMouseAdapter extends MouseAdapter {                 ❺

    public void mouseClicked(MouseEvent event) {
        // this overrides the empty method in Mouse Adapter
        if (SwingUtilities.isLeftMouseButton(event)) {           ❻
            value++;
```

```
            } else if (SwingUtilities.isRightMouseButton(event)) {
                value--;
            }
            result.setText(Integer.toString(value));
        }
    }
}
```

❶ To find the location of the program at runtime.
❷ Assumes the png files are in the same location as the program and create the icons.
❸ Put the icon in the button.
❹ The icon in a `JButton` can be accessed or changed.
❺ To implement a `MouseListener` you have to provide code for 5 methods. In this example only one of the possible events are to be used, but the other methods have to be written. It is more convenient to extend a `MouseAdapter` and override the method that is required..
❻ The most convenient way of finding which mouse button was clicked from the event is to use `SwingUtilites.isLeftButton` and `SwingUtilites.isRightButton`.

The Last Example

This example uses a more complicated component, a Slider, to control a value in a range.

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

class Example7Gui {

    static JLabel result = new JLabel("0",SwingConstants.CENTER);
    static  int value = 0;

    public  void example7() {

        JFrame frame = new JFrame();
        frame.setSize(300,300);
        Container contentPane =  frame.getContentPane();
        JSlider slide = new JSlider(JSlider.VERTICAL,-10,10,0); ❶
        slide.setMajorTickSpacing(5);                           ❷
        slide.setMinorTickSpacing(1);
        slide.setPaintTicks(true);
        slide.setPaintLabels(true);
        slide.setBorder(                                        ❸
            BorderFactory.createEmptyBorder(0,0,10,0));

        contentPane.add(result,BorderLayout.CENTER);
        contentPane.add(slide,BorderLayout.EAST);
        slide.addChangeListener(new ChangeValue());
        frame.setVisible(true);
    }

    class ChangeValue implements ChangeListener {

        public void stateChanged(ChangeEvent expn) {
            JSlider source = (JSlider)expn.getSource();
            if (!source.getValueIsAdjusting()) {                ❹
```

```
                value = (int)source.getValue();
                result.setText(Integer.toString(value));
            }
        }
    }
}
```

❶ Create a slider specifying the range and starting value of the knob.
❷ Define the scale to be shown on the slide.
❸ Create a border round the slider and its scale.
❹ The listener is called when the slider moves, here it only changes the label at the end of a movement.