Writing a Simple Graphical User Interface

Second Year Computing Laboratory
Department of Computing
Imperial College

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, and handling of events. They 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.

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 its 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. The method `setDefaultCloseOperation` can be used to specify what is to happen when the main window is closed.

Applets

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

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

These methods are implemented in `JApplet` as do-nothing methods. Most applets override `init()` with code to initialise the applet and `start()` to start it running. You should not 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. An applet which extends the `JApplet` class can use the method `getContentPane()` to access the `Container` which is displayed by the browser. An application usually creates windows by creating one (or more) `JFrame` objects. The application uses `getContentPane()` to access the top level container of the window. One of the first methods that must be used is `setSize` to set the window size. When a window is created it is not visible. Items can be added to it and when the layout is complete `setVisible(true)` makes it appear on the screen.

The Graphical User Interface

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` simply shows text or an icon on an area of the screen. The text and the icon are usually specified with the constructor, but 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:

```
package example1;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingConstants;

class Example1Gui {
protected  void example() {
        final JFrame examp = new JFrame("A very simple gui");
// A label with "0" in the centre of the label
// is created when the applet is created.
        final JLabel onlyComponent = new JLabel("0",SwingConstants.CENTER);

// The add method puts the label inside the container
// and the label is displayed when the container is drawn.
        examp.add(onlyComponent);
        examp.setSize(100,100);
// The program exits when the window is closed.
        examp.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// The screen is not visible until it is complete.
        examp.setVisible(true);
    }
}

public class Example1  {

    public static void main(final String [] args) {
// Starts the GUI on a separate thread.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                final Example1Gui gui = new Example1Gui();
                gui.example();
            }
        });
    }
}
```

Swing and Threads

A GUI needs to run at the same time as the main program. For example a 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 managers implementing the `LayoutManager interface`, and a container's default layout manager can be changed using the `setLayout` method.

There are three main types of layout managers. Simple ones, such as `BorderLayout`, `FlowLayout` and `GridLayout`, provide commonly used layouts. Flexible layout managers like `GridBagLayout` require more work to use. There are also special purpose layout managers, such as `ScrollPaneLayout` that are tailored to certain tasks. 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 (north, south east and west) and one in the middle (center) as shown here:

```
package example2 ;

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

class Example2Gui {

    public void example2() {

        final JLabel  aLabel   = new JLabel("0",SwingConstants.CENTER);
// Here we add a button, but leave the handling
// of pressing the button for later.
        final JButton increase = new JButton("+1");

        final JFrame frame = new JFrame();
        frame.setSize(200,200);
// The second argument to add tells the layout manager
// that the label goes at the top (north) of the container.
        frame.add(aLabel,BorderLayout.NORTH);
// The contents of the CENTER region will expand to
// fill the rest of the frame.
        frame.add(increase,BorderLayout.CENTER);
        frame.setVisible(true);
    }
}

public class Example2 {
    public static void main(final String [] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                final Example2Gui gui = new Example2Gui();
                gui.example2();
            }
        });
    }

}
```

More complicated layout

A more complicated layout can be achieved by overriding the default layout manager with a more powerful layout manager, such as `GridBagLayout` which is very flexible but quite complex to use. A simpler alternative is to split the interface into areas, and implement each area as a separate container each with its own layout manager. For example:

```
package example3;

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

class Example3Gui {

    public  void example3() {
        final JLabel  result = new JLabel("0",SwingConstants.CENTER);
// To replace the button with a square array of related
// buttons, create a JPanel to contain them.
        final JPanel  square = new JPanel();
        final JButton increase = new JButton("+1");
        final JButton decrease = new JButton("-1");
        final JButton red = new JButton("red");
        final JButton clear = new JButton("blank");
        final JFrame frame = new JFrame();

        frame.setSize(200,200);
// The default layout manager for the JPanel square is
// replaced with GridLayout for a 2 by 2 grid.
        square.setLayout(new GridLayout(2,2));
// Four buttons are created and put inside the square.
        square.add(increase);
        square.add(decrease);
        square.add(red);
        square.add(clear);
        red.setBackground(Color.RED);

        frame.add(result,BorderLayout.NORTH);
// The button in the previous example is replaced by the JPanel.
        frame.add(square,BorderLayout.CENTER);
        frame.setVisible(true);
    }
}

public class Example3 {
    public static void main(String [] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                Example3Gui gui = new Example3Gui();
                gui.example3();
            }
        });
    }
}
```

A layout manager definition does not specify what will happen if it is not used correctly. For example if a fifth item is added to the 2 by 2 grid used above, 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 color, border and/or icons. A component, such as a`JPanel`, can also be removed from the container it was in and then replaced by another component. After changing a competent in a container, the container should be redrawn using the `validate` method so that 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. A program with a Graphical User Interface (GUI) needs a different control structure since the interface can detect input from more than one source at the same time. The example above has four buttons which can be pressed in any order. Libraries that support GUIs such as `Swing` and `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. The library then looks to see if any objects (called listeners) are registered as handling the action. If there are it generates an `Event` and calls the appropriate method of each listener with that 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 are usually subclasses of `java.awt.AWTEvent` and contain information about the 'source' (the object where the event was generated). Different event classes are defined for different input objects. For example a `MouseEvent` is generated when the mouse is used and contains 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 the `EventListener` interface, such as `MouseListener`. Active components, such as buttons, have methods to add listeners, for example:

```
package example4;

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

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

    public  void example4() {

        final JButton increase = new JButton("+1");
        final JButton decrease = new JButton("-1");
        final JButton red = new JButton("red");
        final JButton blank = new JButton("clear");
        final JPanel square = new JPanel();

        final JFrame frame = new JFrame();
        frame.setSize(300,300);
// One listener , an ActionListener , is created for all the buttons
// since the program doesn't need to know which mouse button was pressed.
        final ExampleListener listen = new ExampleListener();

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

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

// The listener is added to the buttons.
        increase.addActionListener(listen);
        decrease.addActionListener(listen);
        red.addActionListener(listen);
        blank.addActionListener(listen);
        frame.setVisible(true);
    }

// The class implementing ActionListener is defined inside the code
// class to be able to access its fileds.
    class ExampleListener implements ActionListener {

// 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() returns the button that was clicked.
        public  void actionPerformed(final ActionEvent event) {
            /* only do this if the buttons are related and
               the action is short */
            // the button pressed
            final JButton sent = (JButton)event.getSource();
            // the button's label
            final String label = sent.getText();

            // Use the button's text to select the action
            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));
        }
    }
}

public class Example4 {
    public static void main(String [] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                Example4Gui gui = new Example4Gui();
                gui.example4();
            }
        });
    }

}
```

Sometimes, when a program is running pressing a button should not have any effect. In this case you can use `setEnabled` with argument `false` to disable a button and with `true` to enable it.

Simplifying

The previous example can be simplified a bit:

```
package example4a;

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

// The class extends JFrame instead of creating a new JFrame object.
class Example4aGui extends JFrame implements ActionListener {
    private int value = 0;
    private final JLabel result = new JLabel("0",SwingConstants.CENTER);

    public  void example4a() {

        final JButton increase = new JButton("+1");
        final JButton decrease = new JButton("-1");
        final JButton red = new JButton("red");
        final JButton blank = new JButton("clear");
        final JPanel square = new JPanel();

        setSize(300,300);

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

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

// this refers to the JFrame.
        increase.addActionListener(this);
        decrease.addActionListener(this);
        red.addActionListener(this);
        blank.addActionListener(this);
        setVisible(true);
    }
// An actionPerformed method must be defined in this class.
// No inner class is needed.
        public  void actionPerformed(final ActionEvent event) {
            /* only do this if the buttons are related and
               the action is short */
            // the button pressed
            final JButton sent = (JButton)event.getSource();
            // the button's label
            final String label = sent.getText();

            // Use the button's text to select the action
            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));
        }
    }


public class Example4a {

    public static void main(String [] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                Example4aGui gui = new Example4aGui();
                gui.example4a();
            }
        });
    }

}
```

Since each interface implemented by a class must have a different name, only one `ActionListener` can be implemented in in the example class.

Example 5

We now extend the previous example to use icons and test which mouse button is being used.

```
package example5;

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

class Example5Gui  {
    private int value = 0;
    private int rows = 2 ;
// To find the location of the program at runtime.
    private final Class myClass = Example5.class;
    private final JLabel result = new JLabel("0",SwingConstants.CENTER);
    private ImageIcon downButtonIcon ;
    private ImageIcon upButtonIcon ;

    protected void  example5(final int theRows) {
// Assumes the .png files are in the same location as the program.
// Creates the icons.
        upButtonIcon =   new ImageIcon(myClass.getResource("Up.png"));
        downButtonIcon = new ImageIcon(myClass.getResource("Down.png"));
        final JButton upButton = new JButton();
        final JFrame frame = new JFrame();
        rows = theRows;
        frame.setSize(300,300);
        final JButton plusMinus = new JButton("+/-");
        final JButton red = new JButton("red");
        final JButton blank = new JButton("clear");
        final ExampleListener listen = new ExampleListener();
        final ExampleMouseAdapter mouseListen = new ExampleMouseAdapter();
        red.setBackground(Color.red);
        final JPanel square = new JPanel(new GridLayout(rows,rows));
        square.add(plusMinus);
        square.add(red);
        square.add(upButton);
        square.add(blank);

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

        plusMinus.addMouseListener(mouseListen);
        red.addActionListener(listen);
        upButton.addActionListener(listen);
        blank.addActionListener(listen);
// Put the icon in the button.
        up.setIcon(upButtonIcon);
        frame.setVisible(true);
    }

    class ExampleListener implements ActionListener {

        public void actionPerformed(final ActionEvent event) {

             // the button pressed
            final JButton sent = (JButton)event.getSource();
            // the button's label
            final 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) {
// The icon in a JButton can be accessed or changed.
```

```
                sent.setIcon(downButtonIcon);
            } else {
                sent.setIcon(upButtonIcon);
            }
            result.setText(Integer.toString(value));
        }
    }


// To implement a MouseListener you have to provide code for 5 methods.
// In this example only one of the possible events is used, but the other
// methods would still have to be written (as empty methods).
// In this case it is more convenient to extend MouseAdapter and
// override the method that is required.
    class ExampleMouseAdapter extends MouseAdapter {

        public void mouseClicked(MouseEvent event) {
            // this overrides the empty method in Mouse Adapter

// The most convenient way of finding out which mouse button was clicked
// from the event is to use SwingUtilities.isLeftButton and
// SwingUtilities.isRightButton.
            if (SwingUtilities.isLeftMouseButton(event)) {
                value++;
            } else if (SwingUtilities.isRightMouseButton(event)) {
                value--;
            }
            result.setText(Integer.toString(value));
        }
    }
}

public class Example5 {
    public static void main(String[] args) {
        //  get the program parameter
        final int rows;
        if (args.length > 0) {
            try {
                rows = Integer.parseInt(args[0]);
                javax.swing.SwingUtilities.invokeLater(new Runnable() {
                    public void run() {
                        Example5Gui gui = new Example5Gui();
                        gui.example5(rows);
                    }
                });
            } catch (NumberFormatException except) {
                System.err.println("The first parameter should be a number");
                System.exit(1);
            }
        } else {
            System.err.println("The number of rows should be given as an argument");
        }
    }
}
```

Final Example

Our last example uses a more complicated component, a slider, to control a value in a range.

```
package example7 ;

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

class Example7Gui {

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

    public  void example7() {

        final JFrame frame = new JFrame();
        frame.setSize(300,300);
// Create a slider specifying the range and starting value of the knob.
        final JSlider slide = new JSlider(JSlider.VERTICAL, -10,10,0);
// Define the scale to be shown on the slide.
        slide.setMajorTickSpacing(5);
        slide.setMinorTickSpacing(1);
        slide.setPaintTicks(true);
        slide.setPaintLabels(true);
// Create a border around the slider and its scale.
        slide.setBorder(
            BorderFactory.createEmptyBorder(0,0,10,0));

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

    class ChangeValue implements ChangeListener {

        public void stateChanged(final ChangeEvent expn) {
            final JSlider source = (JSlider)expn.getSource();
// The listener is called when the slider moves.
// Here it only changes the label at the end of the movement.
            if (!source.getValueIsAdjusting()) {
                value = (int)source.getValue();
                result.setText(Integer.toString(value));
            }
        }
    }
}

public class Example7 {

    public static void main(String []argv) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                Example7Gui gui = new Example7Gui();
                gui.example7();
```

```
                        }
                    });
        }
}
```