Module 6

Java continued ...

We continue with our study of Java/Swing by looking at various semi-related topics, but be reminded that the notes given here are very brief, and should be supplemented by studying the examples and explanations found in the Java Tutorial on the web site.

Visible and invisible Java/Swing elements are found in a containment heirarchy. At the top level we have containers for the different types of application (i.e. an applet, or an application, or a dialog). In a middle level we have the panes, and at a lower level the individual components.

Level	**Container**
Top-level	JFrame JApplet JDialog
Mid-level	JPanel JScrollBar JTabbedPane
Component-level	JButton JLabel ...

Every GUI component must be part of a containment hierarchy[1]. Each top-level container has a content pane, and an optional menu bar, and Java/Swing components are added to either the content pane or the menu bar. Every component must be placed somewhere in this containment heirarchy, or it will not be visible.

[1]To view the containment hierarchy for any frame or dialog, click its border to select it, and then press Control-Shift-F1. A list of the containment hierarchy will be written to the standard output stream.

6.1 Layout management

Every container has a default layout manager, which may be over-ridden with your own if for some reason the existing one is unsatisfactory. The Java platform supplies a range of layout managers, but here we will just look briefly at three. Note that these are AWT components, not Swing .

6.1.1 BorderLayout

BorderLayout is the default layout manager for every content pane, and assists in placing components in the north, south, east, west, and center of the content pane.

```
contentPane.add(new JButton("B1"), BorderLayout.NORTH);
```

6.1.2 BoxLayout

BoxLayout puts components in a single row or column. Here is code to create a centered column of components:

```
pane.setLayout(new BoxLayout(pane, BoxLayout.Y_AXIS));
pane.add(label);
pane.add(Box.createRigidArea(new Dimension(0,5)));
pane.add(...);
```

6.1.3 CardLayout

CardLayout is for when a pane has different components at different times. You may think of it as a stack of same-sized cards.

```
cards = new JPanel();
cards.setLayout(new CardLayout());
cards.add(p1, BUTTONPANEL);
cards.add(p2, TEXTPANEL);
```

You can choose the top card to show:

```
CardLayout cl = (CardLayout)(cards.getLayout());
cl.show(cards, (String)evt.getItem());
```

6.2 Creating menus

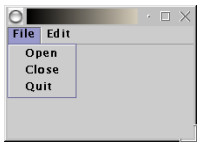
The menu classes are descendants of **JComponent**, and may be used in any higher-level container class (**JApplet** and so on). Here is a small example of a simple menu application, given in the *netbeans* program style:

CODE LISTING **menutest.java**

```
public class menutest extends javax.swing.JFrame {
    public menutest() {
        initComponents();
    }
    private void initComponents() {
        jMenuBar1  = new javax.swing.JMenuBar();
        jMenu1     = new javax.swing.JMenu();
        jMenuItem1 = new javax.swing.JMenuItem();
        jMenuItem2 = new javax.swing.JMenuItem();
        jMenuItem3 = new javax.swing.JMenuItem();
        jMenu2     = new javax.swing.JMenu();
        jMenuItem4 = new javax.swing.JMenuItem();
        jMenu1.setText("File");
        jMenuItem1.setText("Open");
        jMenu1.add(jMenuItem1);
        jMenuItem2.setText("Close");
        jMenu1.add(jMenuItem2);
        jMenuItem3.setText("Quit");
        jMenuItem3.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jMenuItem3ActionPerformed(evt);
            }
        });
        jMenu1.add(jMenuItem3);
        jMenuBar1.add(jMenu1);
        jMenu2.setText("Edit");
        jMenuItem4.setText("Cut");
        jMenu2.add(jMenuItem4);
        jMenuBar1.add(jMenu2);
        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent evt) {
                exitForm(evt);
            }
        });
        setJMenuBar(jMenuBar1);
        pack();
    }

    private void jMenuItem3ActionPerformed(java.awt.event.ActionEvent evt) {
        System.exit(0);
    }
    private void exitForm(java.awt.event.WindowEvent evt) {
        System.exit(0);
    }
    public static void main(String args[]) {
        new menutest().show();
    }
    private javax.swing.JMenuBar jMenuBar1;
    private javax.swing.JMenu jMenu1;
    private javax.swing.JMenuItem jMenuItem1;
    private javax.swing.JMenuItem jMenuItem2;
    private javax.swing.JMenuItem jMenuItem3;
    private javax.swing.JMenu jMenu2;
    private javax.swing.JMenuItem jMenuItem4;
}
```

The end result is:



6.3 Threads in Swing

Java supports a multi-threading mechanism that is sometimes useful. User programs and applets may create several threads to manage different parts of the application. As in any multi-threaded system, we may have critical sections if two threads attempt to access the same variables at the same time. To create threads there are some helpful classes such as **SwingWorker** or **Timer**.

Most Swing components are not thread safe - this means that if two threads call methods on the same Swing component, the results are not guaranteed. The single-thread rule:

Swing components can be accessed by only one thread at a time.

A particular thread, the event-dispatching thread, is the one that normally accesses Swing components. To get access to this thread from another thread we can use **invokeLater()** or **invokeAndWait()**.

6.3.1 Creating threads

Many applications do not require threading, but if you do have threads, then you may have problems debugging your programs. However, you might consider using threads if:

- Your application has to do some long task, or wait for an external event, without freezing the display.
- Your application has to do someting at fixed time intervals.

The following two classes are used to implement threads:

1. **SwingWorker**[2]: To create a thread
2. **Timer**: Creates a timed thread

To use **SwingWorker**, create a subclass of it, and in the subclass, implement your own **construct()** method. When you instantiate the **SwingWorker** subclass, the runtime environment creates a thread but does not start it. The thread starts when you invoke **start()** on the object.

Here's an example of using **SwingWorker** from the tutorial - an image is to be loaded over a network (given a URL). This may of course take quite a while, so we don't block our main thread - (if we did this, the GUI may freeze).

[2] If you find that your distribution does not include SwingWorker.class, download and compile it.

The following code shows the better way of loading the remote image:

CODE LISTING **ImageLoader.java**

```
private void loadImage(final String imagePath,
                       final int index) {
     final SwingWorker worker = new SwingWorker() {
         ImageIcon icon = null;
         public Object construct() {
             icon = new ImageIcon(getURL(imagePath));
             return icon;
         }
         public void finished() {
             Photo pic = (Photo)pictures.elementAt(index);
             pic.setIcon(icon);
             if (index == current)
                 updatePhotograph(index, pic);
             }
         };
     worker.start();
}
```

The **Timer** class is used to repeatedly perform an operation. When you create a **Timer**, you specify its frequency, and you specify which object is the listener for its events. Once you start the timer, the action listener's **actionPerformed()** method will be called for each event.

6.3.2 Event dispatching thread

The event-dispatching thread is the main event-handling thread. It is normal for all GUI code to be called from this main thread, even if some of the code may take a long time to run. However - we have already mentioned that we should not delay the event-dispatching thread.

Swing provides a solution to this - the **InvokeLater()** method may be used to safely run code in the event-dispatching thread. The method requests that some code be executed in the event-dispatching thread, but returns immediately, without waiting for the code to execute.

```
Runnable doWorkRunnable = new Runnable() {
    public void run() { doWork(); }
};
SwingUtilities.invokeLater(doWorkRunnable);
```

6.4 Handling events

Actions associated with Java/Swing components raise events - moving the mouse or clicking a JButton all cause events to be raised. The application program writes a listener method to process an event, and registers it as an event listener on the event source. There are different kinds of events, and we use different kinds of listener to act on them. For example:

Action	Listener type
Button click	ActionListener
A window closes	WindowListener
Mouse click	MouseListener
Mouse moves	MouseMotionListener
Component becomes visible	ComponentListener
Keyboard focus	FocusListener
List selection changes	ListSelectionListener

The listener methods are passed an event object which gives information about the event and identifies the event source.

6.4.1 Event handlers

When you write an event handler, you must do the following:

- Specify a class that either implements a listener interface or extends a class that implements a listener interface.

```
public class MyClass implements ActionListener { ...
```

- Register an instance of the class as a listener upon the components.

```
Component.addActionListener(instanceOfMyClass);
```

- Implements the methods in the listener interface.

```
public void actionPerformed(ActionEvent e) {
    ...//code that reacts to the action...
}
```

Make sure that your event handler code executes quickly, or your program may seem to be slow. In the sample code given so far, we have used window listeners to react if someone closes a window, but not to capture other sorts of events.

6.4.2 Handling events

CODE LISTING **CheckBoxDemo.java**

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

public class CheckBoxDemo extends JPanel {
    JCheckBox chinButton;
    JCheckBox glassesButton;
    StringBuffer choices;
    JLabel pic;
    public CheckBoxDemo() {
        chinButton = new JCheckBox("Chin");
        glassesButton = new JCheckBox("Glasses");
        CheckBoxListener myListener = new CheckBoxListener();
        chinButton.addItemListener(myListener);
        glassesButton.addItemListener(myListener);
        choices = new StringBuffer("--ht");
        pic = new JLabel(new ImageIcon( "geek-" + choices.toString() + ".gif"));
        pic.setToolTipText(choices.toString());
        JPanel checkPanel = new JPanel();
        checkPanel.setLayout(new GridLayout(0, 1));
        checkPanel.add(chinButton);
        checkPanel.add(glassesButton);
        setLayout(new BorderLayout());
        add(checkPanel, BorderLayout.WEST);
        add(pic, BorderLayout.CENTER);
        setBorder(BorderFactory.createEmptyBorder(20,20,20,20));
    }
    class CheckBoxListener implements ItemListener {
        public void itemStateChanged(ItemEvent e) {
            int index = 0;
            char c = '-';
            Object source = e.getItemSelectable();
            if (source == chinButton) {
                index = 0;
                c = 'c';
            } else if (source == glassesButton) {
                index = 1;
                c = 'g';
            }
            if (e.getStateChange() == ItemEvent.DESELECTED)
                c = '-';
            choices.setCharAt(index, c);
            pic.setIcon(new ImageIcon( "geek-" + choices.toString() + ".gif"));
            pic.setToolTipText(choices.toString());
        }
    }
    public static void main(String s[]) {
         JFrame frame = new JFrame("CheckBoxDemo");
         frame.addWindowListener(new WindowAdapter() {
             public void windowClosing(WindowEvent e) {
                 System.exit(0);
             }
         });
         frame.setContentPane(new CheckBoxDemo());
         frame.pack();
         frame.setVisible(true);
    }
}
```

Here is an example of event handling code, simplified from the tutorial. It displays a small graphic, and has two checkboxes. When you change either checkbox, an **itemListener** responds to the event and changes the graphic.

6.5 Summary of topics

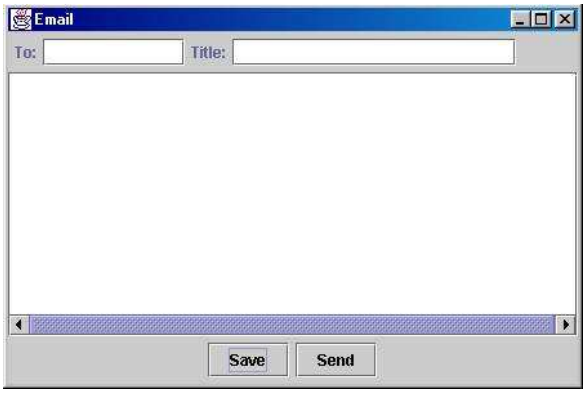
In this module, we introduced the following topics:

- The containment heirarchy
- Layout managers
- Menus
- Threading
- Event handling

Questions for module 5

1. Research the root pane that comes with every highest level container in Java Swing. Briefly describe each of its componenets and state what each could be used for.

2. Give layout management code for the following:



3. Give code for a small menu-style application which makes the console beep whenever a menu item is selected.

Further study

- The **Java tutorial** at http://www.comp.nus.edu.sg/~cs3283/ftp/Java/JavaTutorial/
- **Swing Connect** at http://www.comp.nus.edu.sg/~cs3283/ftp/Java/swingConnect/