


Chapter 8

Java continued


ALERT

- ✔ MCQ test next week
- ✔ This time
- ✔ This place
- ✔ Closed book


ALERT

- ✔ Assignment #2 is for groups of 3
- ✔ Like extended version of tkpaint, but has
  - ✔ menus
  - ✔ persistence
  - ✔ compound objects


Last week

- Tool sets for Java/Swing
- The relationship between JFC, Java and Swing.
- Simple first programs

This week

- Heirarchy
- Layout managers
- Simple first programs

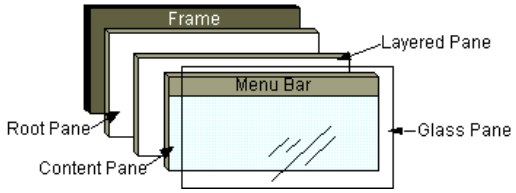
Containment heirarchy

Top level provides panes for descendants to paint themselves

Control-Shift-F1 to view

```
t2[frame0,954,518,126x43,layout=java.awt...
   javax.swing.JRootPane[,4,24,118x15,la...
      javax.swing.JPanel[null.glassPane,...
      javax.swing.JLayeredPane[null.laye...
         javax.swing.JPanel[null.content...
            javax.swing.JLabel[,0,0,118x...
```


Containment heirarchy



Containment heirarchy

The glass pane: Intercepts input events for the root pane.

The layered pane: Serves to position its contents, which consist of the content pane and the optional menu bar.

The content pane: The container of the root pane's visible components, excluding the menu bar.

The menu bar: The home for the root pane's container's menus.

Containment heirarchy

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

Containment heirarchy

Every GUI component must be part of a containment hierarchy[4].

Each top-level container has

- a content pane, and an
- optional menu bar

[4] 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.

Containment heirarchy

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.

Layout management

- ✔ Every container has a default layout manager
- ✔ It may be over-ridden
- ✔ A range of layout managers supplied
- ✔ These are AWT components, not Swing

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);
```


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(...);
```


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);
```


CardLayout

You can choose the top card to show:

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

Creating menus

The menu classes are descendants of **JComponent**, and may be used in any higher-level container class (**JApplet** and so on).

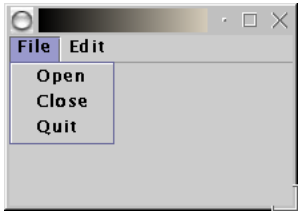
Creating menus

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");
        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;
}
```


Menus

The end result is:



Threads in Swing

- ✔ Java supports multi-threading
- ✔ We may have critical sections
- ✔ To create threads use **SwingWorker** or **Timer**.

Threads

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 accessed by only one thread at a time.

Threads

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()**.

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.

Implementing threads

The following two classes are used to implement threads:

1. **SwingWorker**[5]: To create a thread
2. **Timer**: Creates a timed thread

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

SwingWorker

To use **SwingWorker**, create a subclass of it, and in the sub-class, 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.

Example

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).

SwingWorker example

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();
}
```


Timer

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.

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.

InvokeLater

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);
```


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.

Listener types

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

Listeners

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

Event handlers

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

- Specify a class
- Register an instance of the class as a listener
- Implement the methods

Specify class

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

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


Register it

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

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

Implement method

Implements the methods in the listener interface.

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


Event handling

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.

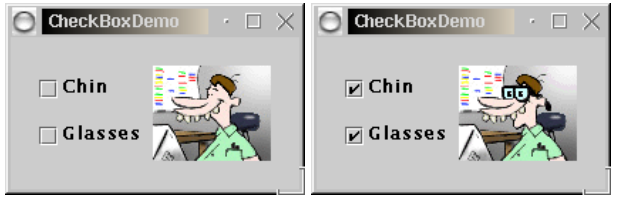
Handling events

CODE LISTING **CheckBoxDemo.java**

```
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 ("--h");
        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;
            } 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);}
                                 }
        );
```


Example code

When you change either checkbox, an **itemListener** responds to the event and changes the graphic.



Summary of topics

In this module, we introduced the following topics:

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