

# *Notes from Well House Consultants*

*These notes are written by Well House Consultants and distributed under their Open Training Notes License. If a copy of this license is not supplied at the end of these notes, please visit*

*<http://www.wellho.net/net/whcotnl.html>  
for details.*

## 1.1 Well House Consultants

Well House Consultants provides niche training, primarily but not exclusively in Open Source programming languages. We offer public courses at our training centre and private courses at your offices. We also make some of our training notes available under our "Open Training Notes" license, such as we're doing in this document here.

## 1.2 Open Training Notes License

With an "Open Training Notes License", for which we make no charge, you're allowed to print, use and distribute these notes provided that you retain the complete and unaltered license agreement with them, including our copyright statement. This means that you can learn from the notes, and have others learn from them too.

You are NOT allowed to charge (directly or indirectly) for the copying or distribution of these notes, nor are you allowed to charge for presentations making any use of them.

## 1.3 Courses presented by the author

If you would like us to attend a course (Java, Perl, Python, PHP, Tcl/Tk, MySQL or Linux) presented by the author of these notes, please see our public course schedule at

<http://www.wellho.net/course/index.html>

If you have a group of 4 or more trainees who require the same course at the same time, it will cost you less to have us run a private course for you. Please visit our onsite training page at

<http://www.wellho.net/course/otc.html>

which will give you details and costing information

## 1.4 Contact Details

Well House Consultants may be found online at

<http://www.wellho.net>

[graham@wellho.net](mailto:graham@wellho.net)

technical contact

[lisa@wellho.net](mailto:lisa@wellho.net)

administration contact

Our full postal address is

404 The Spa  
Melksham  
Wiltshire  
UK SN12 6QL

Phone +44 (0) 1225 708225

Fax +44 (0) 1225 707126

# *Extending Graphics in Java*

The original graphics classes of Java – the AWT – provide functionality at a very low level. A lot of coding effort is required to provide even a simple GUI. The Swing classes are built on top of the AWT and largely supplant it; they provide much higher level components so that you can rapidly bolt together the look, feel and functionality for a graphic-driven application.

<i>Simple Swing</i> .....	4
<i>More Complex Swing</i> .....	14

## Background

The initial releases of Java (1.0 and 1.1) included the applet package and the Applet class therein, which can be extended by Java programmers to provide their own applets. An applet is a small application intended to run in the user's browser, actually on their own client system. JVMs are built into browsers such as Internet Explorer and Netscape to support applets.

Browsers are graphic in nature, and there are security issues involved in letting remote code access keyboard, screen, etc., of a user's machine. So alongside the applet package, Sun provide the awt package - the Abstract Windowing Toolkit.

The lowest levels of graphics programming, where you decide which dots to light to draw a line, are in the specialist realm of the operating system / window system / hardware designer these days. The next level up, where you're placing vectors based on their dot (pixel) positions on the display is also a highly specialist business. All these low-level graphics are taken care of for you by the AWT, which provides drawing in a window. It knows where the window is, whether it's visible, how to control the video memory to make a line or polygon or character or button, etc.

There are many classes in the AWT; basically, they comprise:

- classes for drawing and manipulating components
- classes for drawing graphics primitives
- classes for managing the layout of components
- classes for handling events

The `java.awt.image` package provides additional facilities for image handling, including image manipulation and filtering.

The facilities provided by the awt tend to be somewhat low level (meaning that there's a lot of work to build a simple GUI), and the look and feel varies between platforms as it heavily relies on the local system's look and feel.

In Java 1.2, the swing classes were introduced in `javax.swing`.

Swing classes use the awt facilities, but are at a higher level, meaning that you need to write less code for more results.

You should bear in mind that although there's a set of Swing classes available for Java 1.1, you're unlikely to find them on many Internet user's browsers, so stick with the awt classes if this is your target market. Companies, such as Microsoft, that included Java in their browsers, aren't rushing to upgrade their browsers, and even when they do update, there's a long history of users running with old software.

A plugin is available to support Java and Swing on browsers, and if your application is Intranet-based (or stand-alone), then Swing may be the route for you to go.

### 2.1 Simple Swing

#### *Hello Swing World*

Here's the simplest stand-alone Swing application:

```
import javax.swing.*;

public class Swtiny extends JPanel {

    // Swing main program

    public static void main (String [] args) {

        JFrame frame = new JFrame ("Tiny, Stand alone");
        JLabel jl = new JLabel("A Tiny Example");

        frame.getContentPane().add(jl);
```

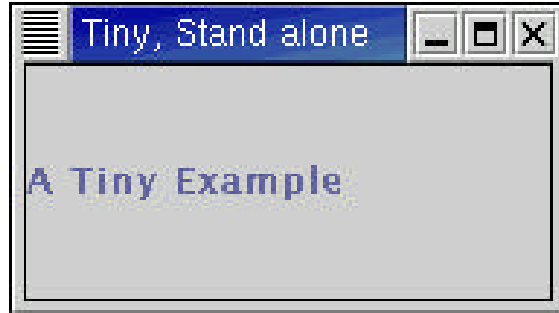
```

        frame.setSize(180,80);
        frame.setVisible(true);
    }
}

```

Our application creates a top-level Frame (a window) into which we're going to place our component. The Frame is labelled with the text "Tiny, Stand alone".

Figure 1 Running our swing application



The frame is set to a size of 180 x 80 pixels, and to be visible - without settings like these, you won't see it, or it will be 0 x 0.

JFrames have many properties, such as their content pane. Do you see how much this is modelled on a real window? Rather than being filled with transparent glass, we fill our window pane with components, in this case a Swing Label (or JLabel), with the text "A Tiny Example" on it.

Components are created through constructors,<sup>1</sup> but they aren't placed in any position in any pane until instructed to do so by the program. This is rather like creating a piece for a board game, but not actually placing it onto the board until the time comes (and until you know where you want to place it). Think of it being like a house or hotel in Monopoly.

### Using multiple components

You'll want to have more than one component on a Pane; just look at any typical GUI window and there's a whole lot of components there. The layout managers from the AWT provide the ability to lay out components in many ways. You can even write your own custom layout manager. We'll use the most basic – a flow layout – for our first example:

```

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

public class S1 extends JFrame {

    public static void main (String [] args) {
        new S1().setVisible(true);
    }

    public S1 () {

        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());

        cp.add(new JLabel("This is a test"));
    }
}

```

<sup>1</sup> just as you would expect ... after all, that's the whole OO paradigm of Java

```

cp.add(new JButton("OK!"));
pack();

}
}

```



Figure 2 Using multiple components

With a flow layout, the components are placed one after another along the top of the pane. When the right-hand side is reached (which hasn't happened in this case), a second row is started, and so on.

Rather than specify a window size as we did in the earlier example that wasn't using a layout manager, we've now left the `pack` method to arrange the components, which in turn allows the size of the window to be calculated.

**Advantage:** The window isn't going to be too small, nor have spurious white space.

**Disadvantage:** Run the same application GUI twice with different data, and it will come up looking different and in different sizes.

Following on from our Monopoly hotels, we now create the components off the board using their constructors. The banker hands them to the player who's purchased them through the `add` method, and he chooses which properties to place them on using `pack`.

You'll note that `getContentPane` returns a `Container` object. This is an underlying AWT object; the rule is that if an object starts with a capital J, then it's a Swing object, and if it starts with any other letter it's in the awt. You will need to know which are which:

- To know where to look in the reference material
- To know what you need to import
- To know whether your class is useable in old browsers without the Java plugin

Let's use another layout manager (a grid layout) in order to set up a series of buttons like a telephone keypad:

```

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

public class S2 extends JFrame {

public static void main (String [] args) {
    new S2().setVisible(true);
}

public S2 () {

    Container cp = getContentPane();
    cp.setLayout(new GridLayout(4,3));

    String [] Phone = {"7","8","9","4","5","6",
                      "1","2","3","*","0","#"};

    for (int i=0;i<Phone.length;i++) {
        cp.add(new JButton(Phone[i]));
    }
}
}

```

*Important to note: If an object starts with a capital J, then it's a Swing object, and if it starts with any other letter it's in the awt.*

```

    }
    pack();
}
}

```

Figure 3 Using another layout manager



Our GridLayout is set to four rows by three columns, and as we add components in they fill up the rows one-by-one.

There's a huge number of properties you can change, and additional methods available on layout managers and on individual components, but we're going to move on now to look at events.

### Event handling

Our Tiny "Hello Swing World" applications were really just too small. Although they provide a frame and add graphics to the frame, they're just for display and you can't actually do anything with them. There's not even any way of getting rid of the window and closing the application properly.

Let's add event handling to our tiniest program:

```

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

public class Ssmall extends JPanel {

    // Swing main program

    public static void main (String [] args) {

        JFrame frame = new JFrame ("Stand alone");
        JLabel jl = new JLabel ("Exits properly");

        frame.getContentPane().add(jl);
        frame.setSize(180,80);
        frame.setVisible(true);

        frame.addWindowListener(new winEvent());
    }
}

```

Which also involves us in adding an event handler class as well:

```
import java.awt.event.*;
class winEvent extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}
```

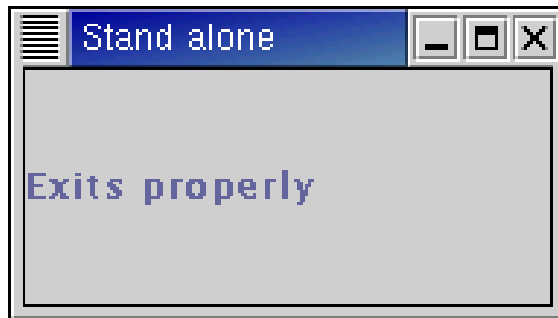


Figure 4 Adding an event handler class

Notice that our main program defines our GUI, then defines a number (one!) of things to be done when an event happens, then appears to "fade out".

Because we've extended a Swing class, the application is left in a listener loop. It's waiting for an event to happen, and acting on that event when it gets it.

As well as **windowClosing**, we can provide methods such as **windowActivated** and **windowDeactivated** if we want to track when focus comes in and out of our window, and several others. The **WindowAdapter** class is a trivial implementation of the **WindowListener** interface. If you provide all seven methods defined by that interface, you can switch to an "implements" and save your inheritance.

### Feedback from events on individual components

As well as handling events on our window, we can (and usually will) handle them on individual components. Let's use the **ActionListener** interface to note which buttons are pressed on our telephone pad. In this example, we'll echo the key strokes to **System.out**, but then we'll move on to providing the feedback within the GUI.

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

public class Dial extends JFrame {

    public static void main (String [] args) {
        new Dial().setVisible(true);
    }

    public Dial () {

        Container cp = getContentPane();
        cp.setLayout(new GridLayout(4,3));

        String [] Phone = {"7","8","9","4","5","6",
                          "1","2","3","*","0","#"};

        for (int i=0;i<Phone.length;i++) {
            JButton current;
            cp.add(current = new JButton(Phone[i]));
        }
    }
}
```

```

        current.addActionListener(
            new ActionListener()

// Definition through an inner class =====
        {
            public void actionPerformed(ActionEvent e) {
                System.out.print(e.getActionCommand());
            }
        }
// End of inner class definition =====
    );
    }
    pack();

}
}

```

We've chosen to use an inner class here to define the action to be performed which saves us getting too many separate source files all over the place. You will find that a file called *Dial\$1.class* has turned up so it does not save you a class file, but then you'll be using a jar for distribution, won't you?

The keypad just looks the same as the earlier one did, but when you run it and select buttons, you'll get ...

```

$
01225708225
$

```

Figure 5 Providing feedback and controls



## Providing feedback in the GUI

We'll now complete our telephone dialling GUI by providing feedback of the number being dialled within the panel, and also providing controls such as a "done" button. Here's the source code:

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

public class Dialer extends JFrame {

    static String PhoneNumber = "";
    JLabel Result;

    public static void main (String [] args) {
        new Dialer().setVisible(true);
    }

    public Dialer () {

        Container cp = getContentPane();
        cp.setLayout(new GridLayout(2,1));

        // Upper panel - responses

        JPanel upper = new JPanel();
        upper.setLayout(new GridLayout(3,1));

        upper.add(new JLabel("Dialer Example"));
        upper.add(Result = new JLabel(""));
        JButton OK;
        upper.add(OK = new JButton("OK"));
        OK.addActionListener(
            new ActionListener()
        // Definition through an inner class =====
        {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Called "+PhoneNumber);
                System.out.println("Dialling Completed");
                System.exit(0);
            }
        }
        // End of inner class definition =====
        );

        // Lower panel - keypad

        JPanel lower = new JPanel();
        lower.setLayout(new GridLayout(4,3));

        String [] Phone = {"7","8","9","4","5","6",
            "1","2","3","*","0","#"};

        for (int i=0;i<Phone.length;i++) {
            JButton current;
            lower.add(current = new JButton(Phone[i]));
            current.addActionListener(
```

```

        new ActionListener()

// Definition through an inner class =====
    {
        public void actionPerformed(ActionEvent e) {
            PhoneNumber = PhoneNumber + e.getActionCommand();
            Result.setText(PhoneNumber);
        }
    }
// End of inner class definition =====
    );
    }
    cp.add(upper);
    cp.add(lower);
    this.pack();

}
}

```

Things to notice here:

```

    static String PhoneNumber = "";
    JLabel Result;

```

are both declared as class rather than local variables, so that they can be used within the inner class. Local variables can only be used in the inner class if they're declared as final (i.e. unchanging), which is not appropriate in this case.

We've now got not one but three grid layouts, as we've provided a more complex composite layout. We could have used any of the other layout managers (Border, Grid, GridBag, Flow, Box) or our own for any of the layouts had we wished. Indeed, a Border might have been more appropriate for the outer layout as it would allow the upper and lower areas to differ in size.

The inner panels had to be added to the outer panel. This does not happen automatically, and failure to do so means that the inner panels are not displayed. The whole still had to be packed.

The main action listener (the one that we've used for all the buttons on the telephone pad) actually goes back and modifies the text in the feedback box:

```

    public void actionPerformed(ActionEvent e) {
        PhoneNumber = PhoneNumber + e.getActionCommand();
        Result.setText(PhoneNumber);
    }

```

which is really at the heart of the changes in this example.

Note that Swing uses the window manager and windows look and feel from the client on which it is running. Our Dialer application running on an Apple system will have the "Aqua" look.

### *Hello Swing World as an applet*

The minimal Swing applet doesn't need to allow for window closing (after all, it's the browser that provides such management) so a "Hello Swing World" applet is much simpler:

```

import javax.swing.*;
public class Swapp extends JApplet {
// Swing applet
public void init() {

```

```

JLabel jl = new JLabel("Swing based Applet");
this.getContentPane().add(jl);
}
}

```

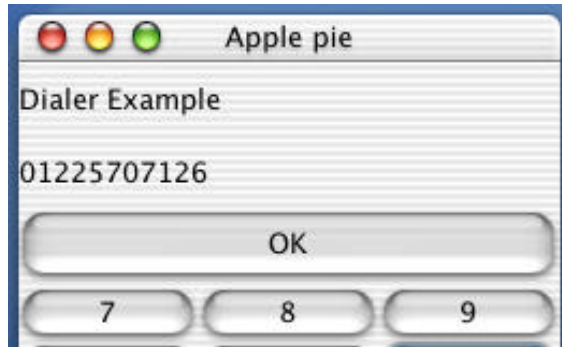


Figure 6 Apple's Aqua look

In order to run this, though, you do need a web page in HTML which includes an object tag (or an applet tag, although that's now deprecated):

```

<html>
<head>
<title>Swing Applet - Hello World</title>
</head>
<body bgcolor=white>
Test applet:<P>
<object code=Swapp.class width=220 height=110>
</object><P>
Uses Java Swing
</body>
</html>

```

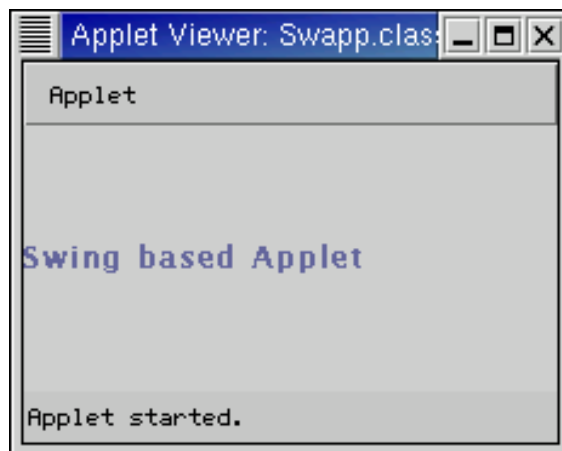


Figure 7 Running the Swing applet

## A Complete GUI on an applet

Here's our Dialler application as an Applet:

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

public class DialApplet extends JApplet {
    static String PhoneNumber = "";
    JLabel Result;

    public void init () {
        new DialApplet();
    }
    public DialApplet () {
        Container cp = getContentPane();
        cp.setLayout(new BorderLayout());
        // Upper panel - responses

        JPanel upper = new JPanel();
        upper.setLayout(new GridLayout(3,1));

        upper.add(new JLabel("Dialer Example"));
        upper.add(Result = new JLabel(""));
        JButton OK;
        upper.add(OK = new JButton("OK"));
        OK.addActionListener(
            new ActionListener()
// Definition through an inner class =====
        {
            public void actionPerformed(ActionEvent e) {
                // Following inappropriate for an Applet!
                // System.out.println("Called "+PhoneNumber);
                // System.out.println("Dialling Completed");
                // System.exit(0);
            }
        }
// End of inner class definition =====
    );

    // Lower panel - keypad

    JPanel lower = new JPanel();
    lower.setLayout(new GridLayout(4,3));

    String [] Phone = {"7","8","9","4","5","6",
                      "1","2","3","*","0","#" };

    for (int i=0;i<Phone.length;i++) {
        JButton current;
        lower.add(current = new JButton(Phone[i]));
        current.addActionListener(
            new ActionListener()
// Definition through an inner class =====
        {
            public void actionPerformed(ActionEvent e) {
```

```

        PhoneNumber = PhoneNumber + e.getActionCommand();
        Result.setText(PhoneNumber);
    }
}
// End of inner class definition =====
);
}
cp.add(upper, BorderLayout.NORTH);
cp.add(lower, BorderLayout.SOUTH);
// cp.pack();
}
}
}

```



Figure 8 Our Dialler application as an applet

You might like to note that we had to make the following changes to our stand-alone GUI application to change it into an Applet:

1. **main** was removed, and **init** added
2. It now extends **JApplet** rather than **JFrame**
3. **pack** was removed
4. The size of the window is now taken from the HTML rather than from the size that's needed to hold the components (beware, you may find unwanted white space, or compressed or overlapping parts of your GUI if you don't think this one through properly).
5. **System.out.println** and **System.exit** were removed. You can't print to the user's display, nor can you exit from a browser. Chances are that results would be passed back via your network to the host from which the applet was loaded.

What if you want to use the same class as both a stand-alone application and within a browser? That's quite feasible; you provide both **main** and **init** methods. Note whether you're running an applet or not (a good use for a boolean). You can then put in small adjustments as necessary to cover the points that we listed as changes above.

Note: We have modified this Dialler example to use a **BorderLayout** for the outer frame; this was not a necessary change as we moved across to an applet. It's done to provide you with an example of a different layout to the **GridLayout**.

## 2.2 More Complex Swing

Swing provides you with a huge selection of components that you can use in your GUI, each with a very wide range of properties and methods available. The **JComponent** abstract class is subclassed (directly or indirectly) to ...

JButton

```

JMenuItem
JCheckBoxMenuItem
JMenu
JRadioButtonMenuItem
JToggleButton
JCheckBox
JRadioButton
JColorChooser
JComboBox
JFileChooser
JInternalFrame
JLabel
JLayeredPane
JDesktopPane
JToolTip
JMenuBar
JOptionPane
JPanel
JPopupMenu
JProgressBar
JRootPane
JScrollBar
JScrollPane
JSeparator
JSlider
JSplitPane
JTabbedPane
JTable
JToolBar
JList
JTree
JViewport

```

We'll choose just two of these for a further look in this module; you'll very much get the idea from these components, and then be able to make good use of the others from documentation and books available elsewhere (have a look at our library listed in the back of this manual; we have complete books on Swing).

### *The JTree component*

JTrees are used to display hierarchical trees and to navigate around them; you could use them for a directory structure to navigate a set of object inheritances, or (at just two levels) to navigate around school classes, and the students who are in each class.

This is Java, so using a JTree is all about using Objects of different types. Most text-books give sophisticated examples that seem better suited to showing how clever the programmer who wrote them was than to helping the newcomer. We'll take a different approach here.

Let's start by looking at the structure. Here's a piece of code that sets up the simplest possible JTree:

```

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

public class TreeDemoSetup extends JPanel {
public static void main (String [] args) {

```

```

JFrame frame = new JFrame("First JTree");
Container cp = frame.getContentPane();
frame.setSize(180,90);
frame.setVisible(true);

DefaultMutableTreeNode Node = new DefaultMutableTreeNode("Top Level");
DefaultTreeModel Behind = new DefaultTreeModel(Node);
JTree simple = new JTree(Behind);
cp.add(simple);

}
}

```



Figure 9 Setting up a simple JTree

As we would expect, we've created an empty tree, and when we display it, it contains nothing. But at least we have the initial structure in place.

What are the elements?

We create NODES that are going to be placed into a TREE, which is then placed into a JTREE Swing component. You can create your own classes for nodes and trees, or implement interfaces that extend abstract classes, or, as we've done here, use the default ones that are provided by Swing.

Let's add some Nodes to our tree:

```

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

public class TreeDemo2 extends JPanel {

public static void main (String [] args) {

String [][] DataSample = { {"numbers","one","two","three"},
    {"fruit","apple","orange","banana","cherry","tomato"},
    {"desserts","pie","pudding"} };

JFrame frame = new JFrame("First JTree");
Container cp = frame.getContentPane();

DefaultMutableTreeNode Node = new DefaultMutableTreeNode("Top Level");
DefaultTreeModel Behind = new DefaultTreeModel(Node);

for (int i=0; i<DataSample.length; i++) {
    DefaultMutableTreeNode Kid = new DefaultMutableTreeNode(DataSample[i][0]);
    Behind.insertNodeInto(Kid,Node,i);
    for (int j=1; j<DataSample[i].length; j++) {

```

```

        DefaultMutableTreeNode GKid = new DefaultMutableTreeNode(DataSample[i][j]);
        Behind.insertNodeInto(GKid,Kid,j-1);
    }

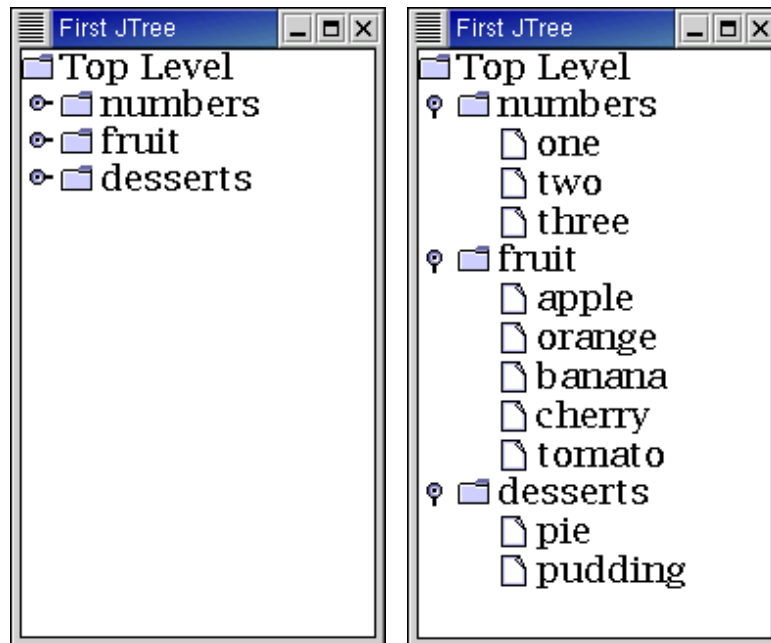
}

JTree simple = new JTree(Behind);
cp.add(simple);
frame.setSize(180,290);
frame.setVisible(true);

}
}

```

Figure 10 Adding nodes to the tree



You'll note that the default classes are doing a lot of work for you here. If you add Children to a node, then the Icon changes from a file Icon to a directory icon. The opening and closing of the directories is all taken care of for you too.

Further methods are available to allow you to change the icons in use, to select whether or not multiple branches can be open at the same time, and so forth.

By default, the text displayed alongside each Object is the result of running the `toString` method on that Object, so that information can be changed as the application is run and the JTree is refreshed. If you write a **TreeCellRenderer**, you can change this default behaviour.

Since the JTree can shrink and grow in height, it's likely that you'll want to include it in a **JScrollPane** component so that you don't start losing vital branches and leaves off the bottom of your window.

And you'll also want to be adding event handling so that you can make use of your JTree for making selections.

To finish off this introduction to the JTree, here's an example that selects all the image files (.gif and .jpgs) in subdirectories of a directory named on the command line, and offers them in a tree. To keep the code simpler, all our application does with the selections is display them in a Text area, but this application could be extended to give you details of each of the images you select, a "thumbnail", and much more.

How does it run? ...

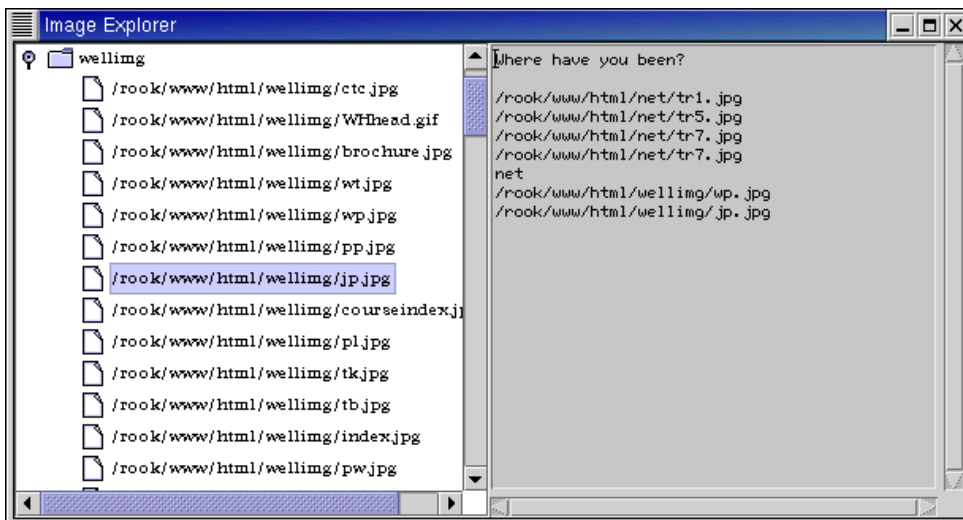


Figure 11 Selecting all the image files and offering them in a tree.

```

import java.io.*;
import java.awt.*;
import java.util.*;
import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;

// Complete example of JTree to explore images held in subdirectories
// of a given base directory

public class ImageExplorer extends JPanel {

    static Vector Selected;
    static TextArea Display;

    public static void main (String [] args) {

        Selected = new Vector();

        JFrame frame = new JFrame("Image Explorer");
        Container cp = frame.getContentPane();
        cp.setLayout(new GridLayout(1,2));

        DefaultMutableTreeNode Node = new DefaultMutableTreeNode(args[0]);
        DefaultTreeModel Behind = new DefaultTreeModel(Node);

        String [] Directories = (new File(args[0])).list();
        int level1 = 0;

        for (int i=0; i<Directories.length; i++) {
            File CurrentDir = new File(args[0] + "/" + Directories[i]);
            if (CurrentDir.isDirectory()) {
                DefaultMutableTreeNode Kid = new DefaultMutableTreeNode(Directories[i]);
                Behind.insertNodeInto(Kid,Node,level1++);
                int level2 = 0;
                String [] Contents = (CurrentDir.list());
                for (int j=0; j<Contents.length; j++) {
                    if (Contents[j].endsWith(".jpg") || Contents[j].endsWith(".gif")) {
                        String CurrentObj = CurrentDir + "/" + Contents[j];
                        DefaultMutableTreeNode GKid = new DefaultMutableTreeNode(CurrentObj);
                        Behind.insertNodeInto(GKid,Kid,level2++);
                    }
                }
            }
        }
    }
}

```

```

        }
    }
}

JTree simple = new JTree(Behind);
simple.setFont(new Font("TimesRoman", Font.PLAIN, 10));

JScrollPane Scroll = new JScrollPane();
(Scroll.getViewPort()).add(simple);

TreeSelectionListener tsl = new TreeSelectionListener() {

// Inner class - tree selection listener
public void valueChanged(TreeSelectionEvent evt) {
    TreePath Follow = evt.getPath();
    Selected.add(Follow);
    String thistime = Follow.getLastPathComponent().toString();
    Display.append(thistime+"\n");
}
}

// End of inner class definition
};

simple.addTreeSelectionListener(tsl);

Display = new TextArea("Where have you been?\n\n");

cp.add(Scroll);
cp.add(Display);

frame.setSize(600,300);
frame.setVisible(true);

}

}

```

### *The JTable Component*

We'll finish this introduction to Swing with a little on the JTable Component. The JTable Swing Component is used for displaying and editing Tabular Data.

Here's a table and the code that generated it:

```

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

public class TableDemo extends JPanel {

public static void main (String [] args) {

    String [][] DataSample = { {"one","two","three","four"},
                                {"apple","orange","banana","cherry"},
                                {"potato","carrot","pea","broccoli"},
                                {"dessert","pie","pudding","sweet"} };
}
}

```

```

String [] tops = {"001","002","003","004"};

JFrame frame = new JFrame("First JTree");
Container cp = frame.getContentPane();

Vector overall = new Vector();
Vector titles = new Vector();

for (int i=0; i<DataSample.length; i++) {
    Vector row = new Vector();
    for (int j=0; j<DataSample[i].length; j++) {
        row.add(DataSample[i][j]);
    }
    overall.add(row);
    titles.add(tops[i]);
}

JTable SampleTable = new JTable(overall,titles);

SampleTable.setFont(new Font("TimesRoman", Font.PLAIN, 18));
cp.add(SampleTable);
frame.setSize(380,200);
frame.setVisible(true);
}
}

```



Figure 12 A JTable component

In our first example, we've used a Vector of Vectors, since there's a convenience constructor it's taken as its parameter, just one of eight constructors that are available. Just as there were **TreeModel** Objects used in JTrees, you'll use **TableModel** objects in JTables, and you'll very often place your table in a **JScrollPane**. This should all start to look hideously familiar.

## Exercise

1. Create a JTree component to the left of a text area, as you saw in the ImageExplorer example in this module. Read in the `/etc/hosts` file line by line, stripping out comments (from `#` to line end). Use the first item on each line as a top-level node, and all subsequent items as second-level nodes in your tree.

2. Extend the exercise to report on selected items in the text area. Same data .... `/etc/hosts` starts:

```
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting. Do not change this entry.
#
#
127.0.0.1      localhost
255.255.255.255 broadcasthost
#
# Part 1 - Subnet 192.168.200.xxx;
#
#
192.168.200.1  left right
192.168.200.2  up down sideways
192.168.200.3  carlisle
192.168.200.4  ash mansell arthur
192.168.200.5  hunt expo92
192.168.200.6  gatwick
192.168.200.7  cod
```

And your running example should look something like this:



# *License*

*These notes are distributed under the **Well House Consultants Open Training Notes License**. Basically, if you distribute it and use it for free, we'll let you have it for free. If you charge for its distribution of use, we'll charge.*

### 3.1 Open Training Notes License

Training notes distributed under the **Well House Consultants Open Training Notes License** (WHCOTNL) may be reproduced for any purpose PROVIDE THAT:

- This License statement is retained, unaltered (save for additions to the change log) and complete.
- No charge is made for the distribution, nor for the use or application thereof. This means that you can use them to run training sessions or as support material for those sessions, but you cannot then make a charge for those training sessions.
- Alterations to the content of the document are clearly marked as being such, and a log of amendments is added below this notice.
- These notes are provided "as is" with no warranty of fitness for purpose. Whilst every attempt has been made to ensure their accuracy, no liability can be accepted for any errors of the consequences thereof.

Copyright is retained by Well House Consultants Ltd, of 404, The Spa, Melksham, Wiltshire, UK, SN12 6QL - phone number +44 (1) 1225 708225. Email contact - Graham Ellis (graham@wellho.net).

Please send any amendments and corrections to these notes to the Copyright holder - under the spirit of the Open Distribution license, we will incorporate suitable changes into future releases for the use of the community.

If you are charged for this material, or for presentation of a course (Other than by Well House Consultants) using this material, please let us know. It is a violation of the license under which this notes are distributed for such a charge to be made, except by the Copyright Holder.

If you would like Well House Consultants to use this material to present a training course for your organisation, or if you wish to attend a public course is one is available, please contact us or see our web site - <http://www.wellho.net> - for further details.

Change log  
Original Version, Well House Consultants, 2004

Updated by: \_\_\_\_\_ on \_\_\_\_\_

Updated by: \_\_\_\_\_ on \_\_\_\_\_

Updated by: \_\_\_\_\_ on \_\_\_\_\_

Updated by: \_\_\_\_\_ on \_\_\_\_\_

Updated by: \_\_\_\_\_ on \_\_\_\_\_

Updated by: \_\_\_\_\_ on \_\_\_\_\_

Updated by: \_\_\_\_\_ on \_\_\_\_\_

*License Ends.*