

## 16. Images in Java Applets and Applications

Images are a primary part of building a professional looking user interface. In this chapter we'll looking at loading and displaying images in applets and in applications as well as at using some of the tools for constructing them.

All versions of Java can display images in GIF (.gif) format and JPEG (.jpg) format. Some versions may allow you to display other formats, but these will not be portable to all platforms. Both applets and applications can display images, but they use different methods in different classes.

The image classes and methods in Java can appear very confusing at first. There is an `Image` class as part of the `awt` package, and there is an `awt.image` package which contains ancillary classes which operate with and on images. The `Image` class consists of the following methods:

```
// Constructors
    public Image();

// Methods
public abstract void flush();
public abstract Graphics getGraphics();
public abstract int getHeight(ImageObserver  observer);
public abstract Object
    getProperty(String  name, ImageObserver  observer);
public abstract ImageProducer getSource();
public abstract int getWidth(ImageObserver  observer);
```

and the `awt.image` package consists of the following classes:

```
class ColorModel
class CropImageFilter
class DirectColorModel
class FilteredImageSource
class ImageFilter
class IndexColorModel
class MemoryImageSource
class PixelGrabber
class RGBImageFilter
```

## Displaying an Image in an Applet

Let's first consider how to display an image in an applet. The applet's **getImage** method returns an object of type `Image` which you can display in the paint method using the `Graphics` method **drawImage**. The arguments to the **getImage** method are the URL where the image file is to be found and the image filename. You can find the server address that you loaded the applet from using the applet method **getCodeBase** and the URL that the document (.html) was loaded from using the **getDocumentBase** method. In summary, to load an image in an applet, the statement is

```
Image image1;
image1 = getImage(getCodeBase(), filename);
```

Then, you draw the image on the screen yourself in the paint method:

```
public void paint(Graphics g) {
    g.drawImage(image1, x, y, w, h, this);
}
```

The final **this** argument is a reference to the current applet as an instance of the `ImageObserver` interface. This provides a way for the thread that handles loading and drawing of images to notify the applet that the image is available. The complete applet for loading and drawing an image is:

```
import java.awt.*;
import java.applet.*;
public class Imgapplet extends Applet {
    private Image image;
    //-----
    public void init() {
        setLayout(null);
        image = getImage(getCodeBase(), "books05.gif");
        setVisible(true);
    }
    //-----
    public void paint(Graphics g) {
        g.drawImage(image, 10, 10, image.getWidth(this),
image.getHeight(this), this);
        g.drawImage(image, 100, 10, 64, 64, this);
    }
}
```

The applet displaying these images is shown in Fig 14- 1, and the code is `imgapplet.java` with a simple html file `imgapplet.html` on your example disk in the `\Images` directory. It is important to note that you can use the Java

interpreter bundled with Netscape to display many Java applets, but when an applet requests an image, it is making an http: request that cannot be satisfied on your local machine. For these programs, you can best test them using the `sppletviewer` program provided with Java. We used the `appletviewer` in Figure 14-1.

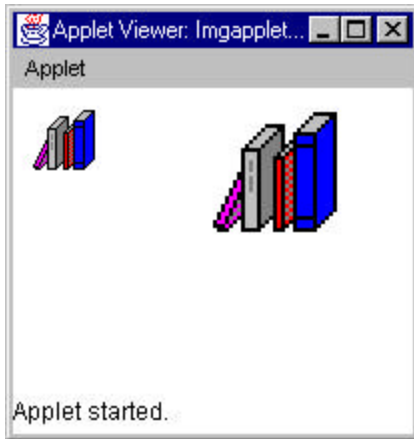


Figure 14-1: The `imgapplet` program displaying the same image at two sizes.

Note that we draw the image twice, once at 32 x 32 pixels and again as 64 x 63 pixels. The Java image drawing methods will stretch or compress the image as needed to fit the scale you specify.

## Displaying an Image in an Application

The logic of loading and displaying an image in an application is similar, but you clearly do not need to reference a parent URL. However, the `Frame` inheritance tree does not have a direct `getImage` method that you can call. Instead, we have to delve a little way under the covers of Java to the `Toolkit` class, which contains the base methods which all implementations of Java must implement in order to produce a graphical user interface. You can obtain the current instance of the toolkit for your platform using the `Component`'s `getToolkit()` method. Thus, to load an image in an application, you call:

```
image = getToolkit().getImage(filename);
```

A complete version of the same program as an application is given on your example disk as `imgapp.java` and is shown below:

```
import java.awt.*;
public class Imgapp extends XFrame {
    private Image image;
    //-----
    public Imgapp() {
        super("Image application");
        setLayout(null);
        image = getToolkit().getImage("books05.gif");
        setBounds(10,10,200,150);
        setVisible(true);
    }
    //-----
    public void paint(Graphics g) {
        g.drawImage(image, 10, 30, 32, 32, this);
        g.drawImage(image, 100, 30, 64, 64, this);
    }
    //-----
    static public void main(String argv[]) {
        new Imgapp();
    }
}
```

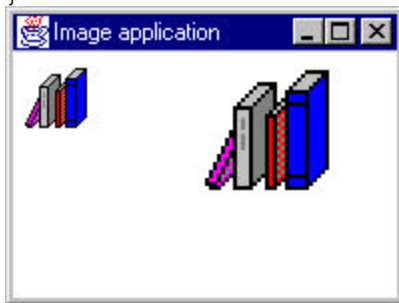


Figure 14-2. The image display in an application.

## Using the MediaTracker Class

When you execute a `getImage` method, the image is loaded in another thread. If you are writing a relatively passive applet, having one image drawn slowly and later than the rest of the controls is not much of a drawback. However, when there several images in a window and they are an

integral part of the program this can be quite disconcerting. Further, in applets, each image is loaded and decompressed separately over the network, and while they may each be loaded in separate threads, the elapsed time for loading ten icon images is still substantial.

If you want to make sure that images are fully loaded before beginning execution of an applet or application, you can add each of the images to a list contained in an instance of the `MediaTracker` class and then ask the class whether each is loaded before proceeding. This class has the following methods:

```
public void addImage(Image image, int id);
public void addImage(Image img, int id, int w, int h);
public boolean checkAll();
public boolean checkAll(boolean load);
public boolean checkID(int id);
public boolean checkID(int id, boolean load);
public Object[] getErrorsAny();
public Object[] getErrorsID(int id);
public boolean isErrorAny();
public boolean isErrorID(int id);
public int statusAll(boolean load);
public int statusID(int id, boolean load);
public void waitAll();
public boolean waitAll(long ms);
public void waitID(int id);
public boolean waitID(int id, long ms);
```

The `addImage` method of the `MediaTracker` object adds images to an internal array or collection along with an integer identifier.

```
track = new MediaTracker(this);
image = getImage(getCodeBase(), "books05.gif");
track.addImage(image, 0);
```

While these identifiers are most commonly simply sequential integers, they can be any values you choose. Then, when you have begun loading of all the images and you want to discover if they have completed, you can use any of the wait methods to check on any or all of the images. The example below follows from one given by Flanagan. It is on your example disk as `TrackImage.java` and the result of this program is show in Figure 14-2.



Figure 14-2: The TrackImage.java program, showing the two images loaded and the final message displayed along the bottom.

```
import java.awt.*;
import java.applet.*;
public class TrackImage extends Applet {
    private Image images[];
    private MediaTracker track;
//-----
    public void init() {
        setLayout(null);
        setFont(new Font("Helvetica", Font.PLAIN, 12));
        images = new Image[2];
        track = new MediaTracker(this);
        //begin loading images
        images[0] = getImage(getCodeBase(), "books05.gif");
        images[1] = getImage(getCodeBase(), "beany.gif");
        //add images into tracker
        for (int i = 0; i < 2; i++)
            track.addImage(images[i], 0);
//check to see if they are done loading
        for (int i = 0; i < 2; i++) {
            showStatus("loading image: " + i);
            try {
                track.waitForID(i);
            }
            catch (InterruptedException e) {
            };
            if (track.isErrorID(i))
                showStatus("Error loading image: " + i);
        }
    }
}
```

```

    }
}
//-----
public void paint(Graphics g) {
    int x, w, h;
    for (int i=0; i <2; i++) {
        x = 100*i + 10;
        w = images[i].getWidth(this);
        h = images[i].getHeight(this);
        g.drawImage(images[i], x, 10, w, h, this);
    }
}
}

```

Now, it isn't always necessary to make sure that all of the images are loaded in the constructor or init routine. You just need to make sure they are loaded before you begin using them. This could be somewhat later, depending on the nature of your application or applet.

## Converting Between Image Types

Java allows you to display images in both JPEG and GIF formats. Both of these are compressed image formats compare to bitmaps where there is at least one byte for each pixel. In general JPEG compression is a little better the GIF and is usually better for pictures. GIF format is usually better for drawings.

If your images are to be downloaded over the internet, it is sometime useful to have them drawn in rough form and filled out later. You can do this by making an *interlaced* GIF file, where alternating lines from the entire image are stored in the first part of the file, and the remaining interlaced lines are stored in the last half of the file. As the image is received and drawn by your browser, it appears to draw the entire image in "rough draft" form and then go back and fill in the remaining lines. This gives the user faster feedback as to the nature of the image you are presenting.

To convert a GIF file into an interlaced GIF, you can use PaintShop Pro ([www.jasc.com](http://www.jasc.com)). Run the program and read in the icon file you wish to convert. Then select File/Save As... as shown in Figure 14-4, and select the GIF89a file format and select Interlaced. You now have a file with a .gif extension containing interlaced data.

## Making Blinking Alternating Images

Once of the simplest forms of animation is the blinking of an image between two or more related images. You can easily do this in Java by reading in the images and then displaying them alternately after a specified delay. We do this in the following program:

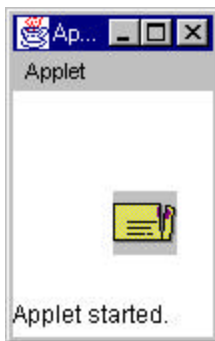
```
import java.awt.*;
import java.applet.*;
//blinks back and forth between 2 images
public class BlinkImage extends Applet implements Runnable {
    Image open, closed;        //two images
    MediaTracker track;        //and tracker
    Thread timer;              //timer for blinking
    boolean openvis;           //flag for which to draw
    public void init() {
        setLayout(null);
        //read in the two images
        open = getImage(getDocumentBase(), "mail02a.gif");
        closed = getImage(getDocumentBase(), "mail02b.gif");
        //make sure they are read in
        track = new MediaTracker(this);
        track.addImage(open, 0);
        track.addImage(closed, 1);
        //check to see if they are done loading
        for (int i = 0; i < 2; i++) {
            showStatus("loading image: " + i);
            try {
                track.waitForID(i);
            }
            catch (InterruptedException e) {
            };
            if (track.isErrorID(i))
                showStatus("Error loading image: " + i);
        }
        //create a thread
        timer = new Thread(this);
        openvis = false;        //first display closed
        timer.start();          //start thread
    }
    //-----
    public void run() {
        //this method is called in new thread
        while (true) {
            openvis = ! openvis; //switch between images
            repaint(50,50,32,32); //and redraw
            try {
                Thread.sleep(100);
            }
        }
    }
}
```



```

        catch (InterruptedException e) {
            };
        }
    }
//-----
    public void update() {
    }
//-----
    public void paint(Graphics g) {
        if (openvis)
            g.drawImage(open, 50,50, Color.lightGray, this);
        else
            g.drawImage(closed, 50,50, Color.lightGray, this);
    }
}

```



Note that to avoid flickering, we repaint only the area of the image itself. In this case, the two images are mail02a.gif and mail02b.gif converted from the Visual Basic Icon files of the same name, and made transparent using PaintShop Pro.

## The ImageIcon Class

The Swing classes (Java Foundation Classes) provide another convenient way to handle images using the ImageIcon class. While the primary purpose of the ImageIcon class is to provide images for buttons, menus and other Swing controls, you can load them and paint them in much the same way. The MediaTracker function is built into the ImageIcon class, and you can use its getImageLoadStatus() method to find out if the image is loaded yet.

Below we show a program similar to ImgApp above, which uses the ImageIcon class.

```

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

public class ImgIcon extends JFrame {
    private ImageIcon image;
    //-----
    public ImgIcon() {
        super("Image application");
        image = new ImageIcon("books05.gif");
        setBounds(100,100,100,100);
        setVisible(true);
    }
    //-----
    public void paint(Graphics g) {
        image.paintIcon (this, g, 20, 20);
    }
    //-----
    static public void main(String argv[]) {
        new ImgIcon();
    }
}

```

The only disadvantage to using the `ImageIcon` class in Applets is that your browser must support Java 2, and not all browsers do.

## Summary

In this chapter, we've looked at how you can load and display images in both applets and applications and discussed how to use the `MediaTracker` class. We've also looked at how to capture and store images and make them part of classes rather than separate loadable files. Finally we looked at image animation using both Java and GIF animation methods.

In the next chapter we'll pick up on building user interfaces again, covering menus and dialog boxes and go on from there to how we can build our own controls.

## 17. Menus and Dialogs

So far we've looked at the main visual controls you use in building applets and applications. Since visual applications are usually derived from the `Frame` class, they also allow you to add drop-down menus to these `Frame` windows. You can, of course, create frame windows from applets as well, but this is done less frequently because of the somewhat inelegant "Unsigned Java Applet" banner that appears along the bottom of such windows.

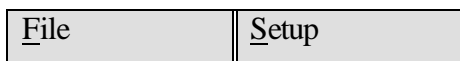
In the following chapter, we'll discuss the Swing classes which have somewhat more elaborate menu features, and these can be added to applets.

### How Menus are Used

A *menu* is a bar that runs across the top of frame windows containing the title of a category of operations. By convention, most menus have "File" as the leftmost entry, followed by Edit. The next to the rightmost menu item will often be "Window" and the rightmost entry is usually "Help." You can insert as many other menu categories between these as are necessary for your application.

Clicking on a menu always causes a list of menu options to drop down under the menu bar. You can then click on any of these options to cause the program to carry out some desired operation. Menus need not be only one level deep, of course; you can create menu items that expand to several more items to the right of the drop-down menu bar.

In the example illustrated in Figure 15-1, the menu bar consists of the two items **File** and **Setup**. Clicking on "File" causes a menu to drop down consisting of "Open," a horizontal separator line, and "Exit." Clicking on "Setup" causes a two-item menu to drop down, consisting of the "Appearance" choice and the "Preferences" choice, where a right-arrow appears to the right of "Preferences." When you click on Preferences, the additional menu items "Colors" and "Filetypes" expand to the right of the arrow.



<u>O</u> pen...	<u>A</u> ppearance...	
-----	<u>P</u> references -->	<u>C</u> olors...
<u>E</u> xit		<u>F</u> iletypes

Figure 15-1: A schematic representation of a simple two item menu with expansion of the “Preferences” choice to two more items.

## Menu Conventions

It is conventional to group items in a menu together using a horizontal separator bar between different logical types of operations. For example, the “File” menu has the usual “Open” menu choice and the “Exit” menu choice. Since exiting from the program is logically rather unlike opening or manipulating files, we separate the Exit choice from the Open choice using a horizontal bar. In more complex menus it is not unusual to see 3 or 4 separator bars on a menu. If you require more than that, you might well reconsider your design, as it becomes confusing to the eye.

Menu items that bring up a new window where the user can select additional choices or enter data into fields always should have an elipsis (...) following them to indicate in advance to your user that they should expect a new window or dialog box to come up when they select this item.

## Creating Menus

To add menus to a frame window, you create an instance of the `MenuBar` class and add it to the frame using the `setMenuBar` method:

```
MenuBar mbar = new MenuBar();    //create menu bar
setMenuBar(mbar);                //and add to Frame
```

Each menu item that runs across the menu bar is an object of type `Menu`, and each item under it is an object of type `MenuItem`.

Thus to create the two entry menu bar we drew in Figure 15-1, we just add two elements to our new menu bar:

```
//Create two top level menu items
File = new Menu("&File",true);
Setup = new Menu("&Setup",false);

//and add them to Menubar
mbar.add(File);
mbar.add(Setup);
```

To add items under the menu, we add *them* to the newly created menu objects:

```
//File menu is File->Open, separator, Exit
Open = new MenuItem("&Open...");
Exit = new MenuItem("E&xit");
File.add(Open);
File.add(new MenuItem("-")); //separator
File.add(Exit);
```

To draw a menu separator, we simply add a new menu item whose caption is a single hyphen. You can also add a separator using the **addSeparator()** method. This gives us the left-hand menu, with the F, O and x underlined, as shown in Fig 15-2. The methods you can use on a Menu object include:

```
public MenuItem add(MenuItem mi);
public void add(String label);
public void addSeparator();
public int countItems();
public MenuItem getItem(int index);
public boolean isTearOff();
public void remove(int index);
public void remove(MenuComponent item);
```

Because Java allows you to add and remove menu elements dynamically during a program, you can change some of the menu elements depending on the nature of the processes your program is carrying out. You need to be careful, however, not to do too much of this as menus that constantly vary can be extremely confusing to navigate.

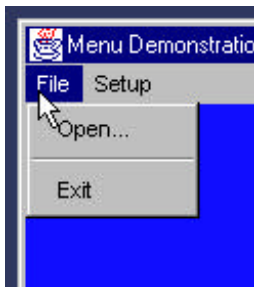


Figure 15-2: The File menu.

## MenuItem Methods

Once you have created menu items under the menu bar, you can use the following methods on them.

```
public void setEnabled(boolean);
public String getLabel();
public boolean isEnabled();
public void setLabel(String label);
```

A menu item which is *disabled* is grayed, but remains visible. You cannot select it. You can also change the name of menu items using the **setLabel()** method.

## Creating Submenus

To add menu items that expand to the right into more menu items, we just add instances of the `Menu` class rather than of the `MenuItem` class, and add the `MenuItems` to them. There is no logical limit to the depth you can extend menus, but practically extending them more than one level becomes hard to use and hard to understand.

So to complete the example we began above, we write the following menu code:

```
//Setup menu is Setup-> Appearance,
//                          Preferences->Colors
//                          Filetypes
//                          Sound
```

```

Appearance = new MenuItem("Appearance");
Setup.add(Appearance);

Preferences = new Menu("Preferences"); //extends
Setup.add(Preferences); //add menu here for sub menus
    Colors = new MenuItem("Colors");
    Preferences.add(Colors);

    Filetypes = new MenuItem("Filetypes");
    Preferences.add(Filetypes);
Sound = new MenuItem("Sound");
Setup.add("Sound");

```

The resulting extended menu is show in Figure 15-3.

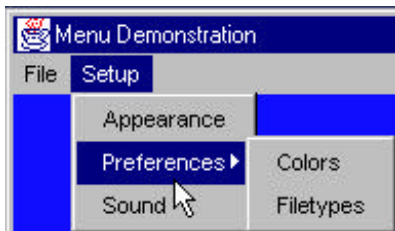


Figure 15-3: The Setup menu, showing second level menus.

## Other Menu Features

When you create a Menu, you can use the alternate constructor:

```
mnu = new Menu(String title, boolean tearoff);
```

The *tearoff* argument is supposed to determine whether the menu is a “tearoff menu.” Such a menu is supposed to stay down after you raise your finger from the mouse button. This, of course, is the normal behavior under Windows-95. As far as we can determine this behavior has only been implemented on the Solaris platform. The boolean argument is otherwise ignored at present.

## Checkbox Menus

The `CheckboxMenuItem` is a special type of menu item which can be checked or unchecked using the mouse or using the `setState(boolean)`

method. You can also find out whether it is checked using the `getState()` method. This object behaves as an ordinary menu item in all implementations of Java 1.0, and does not work in Windows applications even in Java 2.

## Receiving Menu Commands

Clicking on a menu item generates an **actionPerformed** event. As with other events, you must add an `ActionListener` to each `MenuItem`, and process the events in an `actionPerformed` method.

```
public void actionPerformed(ActionEvent evt) {
    Object obj = evt.getSource ();
    if (obj == Exit) {
        clickedExit();
    }
}
```

## Swing Menus and Actions

The `JMenuBar` and `JMenu` classes in Swing work just about identically to those in the AWT. However, the `JMenuItem` class adds constructors that allow you to include an image alongside the menu text. To create a menu, you create a menu bar, add top-level menus and then add menu items to each of the top-level menus.

```
JMenuBar mbar = new JMenuBar();           //menu bar
setJMenuBar(mbar);                         //add to JFrame
JMenu mFile = new JMenu("File");          //top-level menu
mbar.add(mFile);                           //add to menu bar
JMenuItem Open = new JMenuItem("Open");   //menu items
JMenuItem Exit = new JMenuItem("Exit");
mFile.add(Open);                           //add to menu
mFile.addSeparator();                      //put in separator
mFile.add(Exit);
```

The `JMenuItem` objects also generates an `ActionEvent`, and thus menu clicks causes these events to be generated. As with buttons, you simply add action listeners to each of them.

```
Open.addActionListener(this);             //for example
Exit.addActionListener(this);
```



## Action Objects

Menus and toolbars are really two ways of representing the same thing: a single click interface to initiate some program function. Swing also provides an Action interface that encompasses both.

```
public void putValue(String key, Object value);
public Object getValue(String key);
public void actionPerformed(ActionEvent e);
```

You can add this interface to an existing class or create a JComponent with these methods and use it as an object which you can add to either a JMenu or JToolBar. The most effective way is simply to extend the *AbstractAction* class. The JMenu and JToolBar will then display it as a menu item or a button respectively. Further, since an Action object has a single action listener built in, you can be sure that selecting either one will have exactly the same effect. In addition, disabling the Action object has the advantage of disabling both representations on the screen.

Let's see how this works. We can start with a basic abstract ActionButton class, and use a Hashtable to store and retrieve the properties.

```
public abstract class ActionButton extends AbstractAction {
    Hashtable properties;

    public ActionButton(String caption, Icon img) {
        properties = new Hashtable();
        properties.put(DEFAULT, caption);
        properties.put(NAME, caption);
        properties.put(SHORT_DESCRIPTION, caption);

        properties.put(SMALL_ICON, img);
    }
    public void putValue(String key, Object value) {
        properties.put(key, value);
    }
    public Object getValue(String key) {
        return properties.get(key);
    }
    public abstract void actionPerformed(ActionEvent e);
}
```

The properties that Action objects recognize by key name are

```
DEFAULT
LONG_DESCRIPTION
NAME
```

```
SHORT_DESCRIPTION
SMALL_ICON
```

The NAME property determines the label for the menu item and the button, and in theory the LONG\_DESCRIPTION should be used. The icon feature does work correctly.

Now we can easily derive an ExitButton from the ActionButton like this:

```
public class ExitButton extends ActionButton {

    JFrame fr;
    public ExitButton(String caption, Icon img) {
        super(caption, img);
    }
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
}
```

and similarly for the FileButton. We add these to the toolbar and menu as follows:

```
//Add File menu
    JMenu mFile = new JMenu("File");
    mbar.add(mFile);

    //create two Action Objects
    Action Open = new FileButton("Open",
        new ImageIcon("open.gif"), this);
    mFile.add(Open);

    Action Exit = new ExitButton("Exit",
        new ImageIcon("exit.gif"));
    mFile.addSeparator();
    mFile.add(Exit);

//now create toolbar that fixes up the buttons as you add
them
    toolbar = new JToolBar();
    getContentPane().add(jp = new JPanel());
    jp.setLayout(new BorderLayout());
    jp.add("North", toolbar);

    //add the two action objects
    toolbar.add(Open);
    toolbar.addSeparator();
    toolbar.add(Exit);
```

This code produces the program window shown in Figure 30.1.



Figure 30.1 – A menu with pictures using the same Action object as the toolbuttons.

The problem with his approach is that we don't usually want the images in the menu or the text on the toolbar. However, the *add* methods of the toolbar and menu have a unique feature when used to add an action object. They return an object of type `JButton` or `JMenuItem` respectively. Then you can use these to set the features the way you want them. For the menu, we want to remove the icon

```
Action Open = new FileButton("Open",
    new ImageIcon("open.gif"), this);
menuItem = mFile.add(Open);
menuItem.setIcon(null);
```

and for the button, we want to remove the text and add a tooltip:

```
JButton button = toolbar.add(act);
button.setText("");
button.setToolTipText(tip);
button.setMargin(new Insets(0,0,0,0));
```

This gives us the screen look we want: in Figure 30.2.



Figure 30.2 – The menu items with the Action object's images turned off.

## Dialogs

A dialog is a temporary window that comes up to obtain information from the user or to warn the user of some impending condition, usually an error. We have already seen the `FileDialog` in chapter 12. Let's see how we would implement it in this simple menu program.

### The File Dialog

The file dialog is a class which brings up that operating system's file-open or file-save dialog. You can set which mode it comes up in as part of the constructor:

```
FileDialog fdlg =
    new FileDialog(this, "Open file", FileDialog.OPEN);
or
```

```
FileDialog fdlg =
    new FileDialog(this, "Open file", FileDialog.SAVE);
```

Once you have created an instance of the file dialog, you can use the **`setDirectory()`** and **`setFile()`** methods to specify where it should start and what the default filename should be. The dialog doesn't actually appear until its **`show()`** method is called. The `FileDialog` object is always a *modal* dialog, meaning that

1. input to other windows is blocked while it is displayed, and
2. no code in the calling thread after the **`show`** method is called until the `FileDialog` object is dismissed by selecting OK or Cancel.

To call a dialog from our Open menu item, we simply insert the following in the action routine:

```
if (obj == Open) {
    FileDialog fdlg = new FileDialog(this,
                                   "Open", FileDialog.LOAD);
    fdlg.show ();                //start dialog display
    //display selected file name
    filelabel.setText(fdlg.getFile());
```

```
}

```

The file dialog will remain in control until it is closed with either OK or Cancel, as shown in Fig 15-4, and then control will return to the statement following the **show** method call, and you can obtain the selected filename using the **getFile()** method. In the example above, we display that filename in a label in the main window. The complete program for launching the file dialog from the File/Open menu is MenuFrame.java in the \chapter15 directory of your example disk.

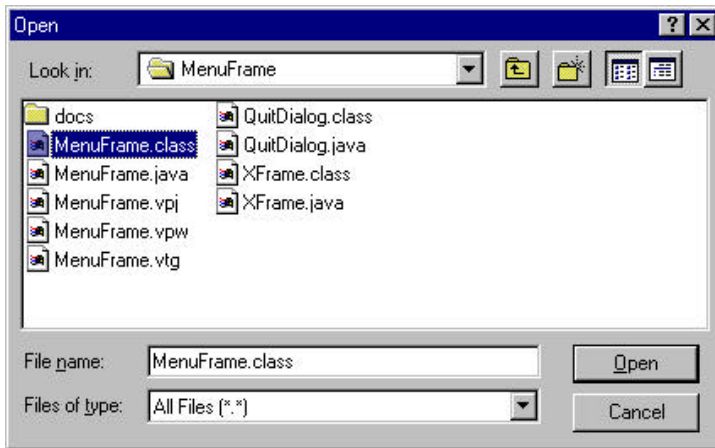


Figure 15-4: The File Dialog shown from the MenuDialog program.

## The Dialog Class

A dialog is a window that is meant to be displayed temporarily to obtain or impart information. In Java, the Dialog class is a window that can be created normally or more usually as a modal window. The default layout for a dialog window is the BorderLayout.

To create a modal dialog you invoke the constructor:

```
public Dialog(Frame p, String title, boolean modal);
```

and could do this directly by calling

```
Dialog qdialog = new Dialog(this, "Quit Yet?", true);
```

However, it is more common to extend the Dialog class and put all of the control and layout information inside the derived class.

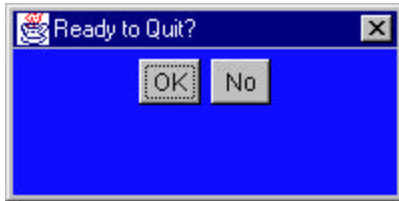
```
import java.awt.*;
import java.awt.event.*;

public class QuitDialog extends Dialog implements
ActionListener{
    boolean exitflag;
    Button OK, Cancel;
//-----
    public QuitDialog(Frame fr) {
        super(fr, "Ready to Quit?", true);
        Panel p1 = new Panel();
        OK = new Button("OK");
        Cancel = new Button("No");
        add("Center", p1);
        p1.add(OK);
        p1.add(Cancel);
        OK.addActionListener (this);
        Cancel.addActionListener (this);
        setSize(200,100);
        exitflag = false;
    }
//-----
    public void actionPerformed(ActionEvent evt) {
        Object obj = evt.getSource ();
        if (obj == OK) {
            exitflag=true;
            System.out.println(exitflag);
            setVisible(false);
        }
        else
            if (obj == Cancel) {
                exitflag = false;
                setVisible(false);
            }
    }
//-----
    public boolean getExitflag() {
        return exitflag;
    }
}
```

Then we create an instance of this new class in response to the File/Exit menu item selection:

```
private void clickedExit() {
    QuitDialog qdlg = new QuitDialog(this);
    qdlg.show();
}
```

Like the `FileDialog`, members of the `Dialog` class are invisible when created and appear as modal windows when their **show** method is called.



## Calling Dialogs from Applets

Since dialogs require a `Frame` as a parent, you might think that you can't pop up a dialog box from an applet. However, there is nothing to stop you from creating an invisible frame and using it as the parent of a dialog window.

Consider the following code from the example program `AppletDialog` on your example disk. This applet displays a single pushbutton labeled "Colors" and a label which is initially set to "no color." When we press this button, a dialog comes up as shown in Figure 15-5. Here we can select one of 3 colors and click on `OK` to pass that color choice back to the main applet. This requires an interface

```
interface SelMethod
{
    public void setColorName(String s);
}
```

that the dialog can call to pass the color name to the calling program.



Figure 15-5: The color dialog produced by the AppletDialog program.

We need to create a frame and pass the dialog an instance of the `SelectMethod` class:

```
Frame fr = new Frame(); //create invisible frame
SelectDialog sdlg = new SelectDialog(fr, this);
sdlg.show();
```

Note that here the frame argument is distinct from the parent argument. The frame is the invisible one just created, and the **this** reference refers to the applet.

We then create a dialog with the OK and Cancel buttons in the southern border and put a panel in the middle to hold the three option buttons:

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

public class SelectDialog extends Dialog implements
ActionListener {
    Button OK, Cancel;
    CheckboxGroup cbg;
    Checkbox red, green, blue;
    SelMethod selm;
//-----
    public SelectDialog(Frame fr, SelMethod sm) {
        super(fr, "Select Color", true);
        setFont(new Font("Helvetica", Font.PLAIN, 12));
        selm = sm;
        Panel pl = new Panel();
        OK = new Button("OK");
        Cancel = new Button("No");
        add("South", pl);
        pl.add(OK);
        pl.add(Cancel);
        OK.addActionListener (this);
```



```

Cancel.addActionListener (this);
cbg = new CheckboxGroup();
red = new Checkbox("red", cbg, true);
green = new Checkbox("green", cbg, false);
blue = new Checkbox("blue", cbg, false);
Panel p2 = new Panel();
add("Center", p2);

p2.add(red);
p2.add(green);
p2.add(blue);
setSize(200,100);
}
//-----
public void actionPerformed(ActionEvent evt) {
    Object obj = evt.getSource ();
    if (obj == OK) {
        clickedOK();
        dispose();
    }
    else
        if (obj == Cancel) {
            dispose();
        }
}
}
}

```

Then when the OK button is clicked, we send the color name back to the calling applet:

```

//-----
private void clickedOK() {
    String colorname="none";
    if (red.getState()) colorname="red";
    if (green.getState()) colorname="green";
    if (blue.getState()) colorname="blue";
    selm.setColorName(colorname);
}
}

```

## Summary

In this chapter, we've learned how to build menus and respond to menu item clicks. We've also learned how to display dialogs in both applications and applets, and how to write interfaces to allow calling programs to be accessed by the results of the dialog.

In the next chapters, we'll begin considering how to build actual, useful classes of the kind we might use in real programs.



## 18. Using the Mouse in Java

The mouse is an integral part of the graphical environment, and in this chapter we show you all of its capabilities in detail so you can appreciate how it can be used.

Mouse events have been divided into two categories with two different listeners. The methods of the `MouseListener` are

```
public void mouseEntered(MouseEvent evt);
public void mouseClicked(MouseEvent evt);
public void mouseExited(MouseEvent evt);
public void mousePressed(MouseEvent evt);
public void mouseReleased(MouseEvent evt);
```

and those of the `MouseMotionListener` are

```
public void mouseDragged(MouseEvent evt);
public void mouseMoved(MouseEvent evt);
```

You can add either listener to any `Container` class.

You can also change the mouse cursor to any of 14 different shapes named as constants in the `Cursor` class.

```
public final static int CROSSHAIR_CURSOR;
public final static int DEFAULT_CURSOR;
public final static int E_RESIZE_CURSOR;
public final static int HAND_CURSOR;
public final static int MOVE_CURSOR;
public final static int N_RESIZE_CURSOR;
public final static int NE_RESIZE_CURSOR;
public final static int NW_RESIZE_CURSOR;
public final static int S_RESIZE_CURSOR;
public final static int SE_RESIZE_CURSOR;
public final static int SW_RESIZE_CURSOR;
public final static int TEXT_CURSOR;
public final static int W_RESIZE_CURSOR;
public final static int WAIT_CURSOR;
```

using the method

```
setCursor(new Cursor(int ));
```

In Windows-95, several of these are identical, such as N\_RESIZE and S\_RESIZE.

## Changing the Mouse Cursor Type

Changing the cursor as you move the mouse over a form can be a valuable clue to the user that different activities will take place. To illustrate how this is done, the Curses.java program on your example disk creates an array of 14 panels in a 2 x 7 GridLayout and assigns a different cursor index to each of them. Since the numerical values which correspond to each of the cursor types vary between 0 and 13, we use these as index values.

```
import java.awt.*;
//-----
public class Curses extends XFrame {

//illustrates the 14 different cursor types

    public Curses() {
        super ("Cursor Demo");

        //lay out a 7 x 2 grid
        setLayout(new GridLayout(7,2));
        //Create 14 panels, each with a different index
        for (int i =0; i<14; i++) {
            Cpanel p = new Cpanel(i);
            add(p);
        }
        setBounds(100,100, 300,150);
        setVisible(true);
    }
//-----
    static public void main(String argv[]) {
        new Curses();
    }
//-----
}
```

This program works by creating 14 instances of the Cpanel class, derived from Panel. We also pass in the index of the particular cursor we want to display. Then when the **mouseEntered** method for that panel is executed by a passing mouse pointer, we change the shape of the pointer using

```
public void mouseEntered(MouseEvent evt) {
//as the mouse enters the panel
```

```
//set the cursor to this panels type
    setCursor(new Cursor(index));
}
```

to call the setCursor method. The resulting program display is shown in Figure 18-1.

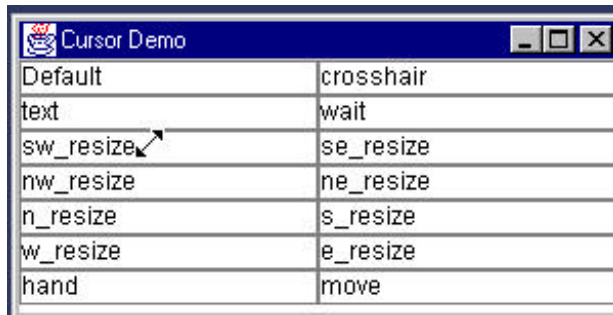


Figure 18-1: The display of the Curses.java program, listing the 14 possible cursor types.

## The MouseAdapter Class

It is sometimes a bother to have to define all five of the mouse events when you only need to use one or two of them. And yet, in order to implement the MouseListener interface, you must define all of them. The MouseAdapter convenience class is provided as an alternative. It is a complete class which defines all five event methods as empty functions. You can then subclass this class and implement only the methods you need. In the CurseAdapter program, we derive a simple CurseListener class that does just that.

```
//listens for mouse Entered event
//ignores all others
public class CurseListener extends MouseAdapter {
    int index;
    Panel pnl;
    public CurseListener(Panel p, int cursor_index) {
        index = cursor_index;
        pnl = p;
    }
    //sets cursor when mouse enters panel
    public void mouseEntered(MouseEvent evt) {
        pnl.setCursor(new Cursor(index));
    }
}
```

The only disadvantage to using this approach is that you may have to create a class where implementing an interface might have been less confusing. Note that in this simple class we have to pass in both the Panel and the cursor index so that when the mouseEntered event occurs, the method can set the cursor for that panel.

## Changing the Cursor in Separated Controls

The only reason that the Curses program cursor always produces the right cursor is that the CPanel containers are laid out next to each other with no separation. If there is a background frame where cursor is to take on another shape, you must specifically program this case. Figure 18-2 shows the display of the MousePanels.java program where there are three colored panels separated on the Frame background.

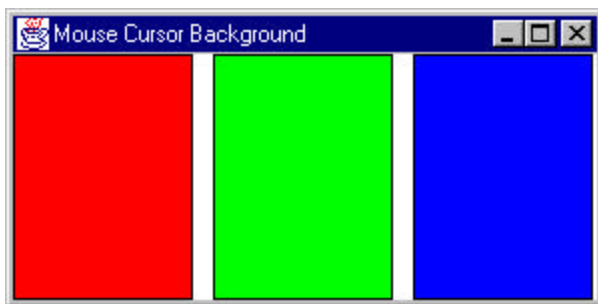


Figure 18-2: The Mouse1.java program display.

We create the panels in the usual way:

```
public class MousePanels extends JFrame {
    Cpanel Red, Green, Blue;

    public MousePanels() {
        super("Mouse Cursor Background");
        setLayout(new GridLayout(1,3,10,10));
        Red = new Cpanel(Color.red);
        Green = new Cpanel(Color.green);
        Blue = new Cpanel(Color.blue);
        add(Red);
        add(Green);
    }
}
```

```

        add(Blue);
        setBounds(100,100,300,150);
        setVisible(true);
    }
    //-----
    static public void main(String argv[]) {
        new MousePanels();
    }
}

```

Our Cpanel class simply fills the background in different colors and establishes a mouse listener.

```

public class Cpanel extends Panel {
    int index;           //cursor index
    Frame frm;
    CurserListener clisten;
    Color color;
    //-----
    public Cpanel(Color c) {
        super();
        color = c;       //save the color
        clisten = new CurserListener(this,
            Cursor.HAND_CURSOR);
        addMouseListener(clisten);
    }
    //-----
    public void paint(Graphics g) {
        Dimension sz = getSize();
        g.setColor(color);
        //fill with specified color
        g.fillRect(0,0, sz.width-1, sz.height-1);
        g.setColor(Color.black);
        //draw back border
        g.drawRect(0,0, sz.width-1, sz.height-1);
    }
}

```

The mouseAdapter class listens for both the mouseEntered and mouseExited events and saves the old cursor on entry and restores it on exit.

```

public class CurserListener extends MouseAdapter {
    int index;
    Panel pnl;
    Cursor oldCursor;

    public CurserListener(Panel p, int cursor_index) {
        index = cursor_index;
        pnl = p;
    }
}

```

```

//restores cursor on exit
public void mouseExited(MouseEvent evt) {
    pnl.setCursor(oldCursor);
}
//sets cursor when mouse enters panel
public void mouseEntered(MouseEvent evt) {
    oldCursor = pnl.getCursor ();
    pnl.setCursor(new Cursor(index));
}
}

```

## Capturing Mouse Clicks

The `mouseDown` and `mouseUp` methods can be received by any child of `Component`, which is to say by any visual control. If you want to take some action when the user clicks on a control, you can simply subclass the `mouseDown` event for that control.

It is advisable, however, to give the user some feedback to indicate that he has clicked on the control. In the `MouseClicked.java` program, whose display is shown in Fig 18-3, the program draws an outline square around the control when the mouse is down, and draws the square without the outline when the mouse is up.

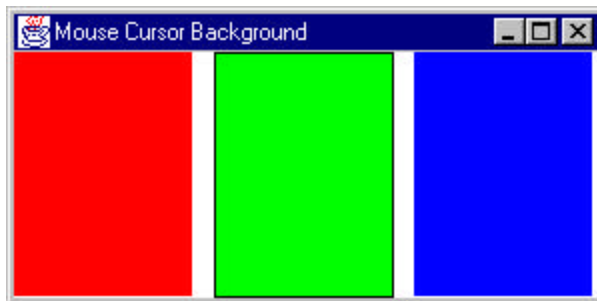


Figure 18-3: The display of the `MouseClicked.java` program, showing how we outline a square when we click the mouse on it.

To produce this outline, we set a flag when the `mouseDown` event is received and clear it when the `mouseUp` event occurs.

```

public void mousePressed(MouseEvent evt) {
    drawFrame = true;
}

```



```

        repaint();
    }
    public void mouseReleased(MouseEvent evt) {
        drawFrame = false;
        repaint();
    }
}

```

Then we force the panel to redraw itself by calling the **repaint** method, which in turn calls the **paint** method. In this paint method, we draw the outline square only if the **isFrame** flag is true:

```

public void paint(Graphics g) {
    Dimension sz = getSize();
    g.setColor(color);
    //fill with specified color
    g.fillRect(0,0, sz.width-1, sz.height-1);
    g.setColor(Color.black);
    //draw back border if flag set
    if (drawFrame) {
        g.drawRect(0,0, sz.width-1, sz.height-1);
    }
}

```

Since we are using 4 of the 5 events of the `MouseListener`, we implement them directly in the `Cpanel` class rather than in a separate `MouseAdapter` class. This also simplifies the interaction with drawing the outline since we can set the `drawFrame` flag in the same class.

## Double Clicking

There is no specific event for a double click of a mouse button. However, the `MouseEvent` has a `getClickCount` method that returns the number of consecutive `mouseDown` events, and you can use it to detect a double click:

```

public void mouseClicked(MouseEvent evt) {
    if (evt.getClickCount () > 1 )
        drawFrame = true;
    else
        drawFrame = false;
    repaint()
}

```

This is illustrated in the `MouseDouble.java` program.

## Double Clicking in List Boxes

In the case of list boxes, the `itemStateChanged` event occurs if a mouse is clicked on a line of the list box. The `actionPerformed` event occurs if you double-click on a line. The same action event also occurs if you press the Enter key after selecting a line in a listbox.

## Dragging with the Mouse

One powerful, but less used feature is the ability to drag controls around using the mouse. You might do this to rearrange controls for your own purposes at run time, or you might do it to establish communication between two controls.

The `mouseDrag` method is executed whenever the mouse button is depressed inside a control and then moved with the mouse down. It does not occur if the button is already down when the mouse enters the control boundary: instead this is treated as an attempt to drag the background. Mouse enter and exit methods are not called during a drag: all mouse movements are received by the control where the mouse was first depressed even if it moves outside the control.

The mouse movements a control receives during `mouseDrag` contain the mouse's current x-y coordinates *relative to that control*. Thus if you want a control to actually move when you are dragging the mouse over it, you must compute the mouse's position in the containing frame and move the mouse in that coordinate space.

For example, in the `MouseDrag.java` program we put up the same three colored panels, but allow them to be moved about by dragging them with the mouse. Our revised `Cpanel` class saves the position of the `mousePressed` event in two private variables:

```
public void mousePressed(MouseEvent evt) {
    drawFrame=true;
    oldX = evt.getX();
    oldY = evt.getY ();
    repaint();
}
//-----
```

Then whenever the mouse is dragged, we compute where to move the control by

1. getting the current mouse position
2. computing the delta from where the mouse was originally clicked,
3. getting the current control position in the parent frame
4. moving the control by adding the delta to the parent position.

We do this as follows:

```
//Mouse Motion events
    public void mouseMoved(MouseEvent evt) {}

    public void mouseDragged(MouseEvent evt) {
        Point locn = getLocation();
        thisX = locn.x ;
        thisY = locn.y;
        int dx = evt.getX ()- oldX;
        int dy = evt.getY() - oldY;
        setLocation(thisX+dx, thisY+dy);

        oldX += dx;
        oldY += dy;
        repaint();
    }
```

## Limitations on Dragging

In Java, while the dragged component in a frame can be moved anywhere you like, it cannot be moved “on top” of a component that was added to the frame later. We added the three instances to the frame in the order

```
add(Red); //add them to
add(Green); //the layout
add(Blue);
```

Thus, while we can drag the red square over the green or blue one and the green over the blue, we cannot drag the blue one over either the red or the green. This is illustrated in Figure 18-5, showing the overlapping panels:

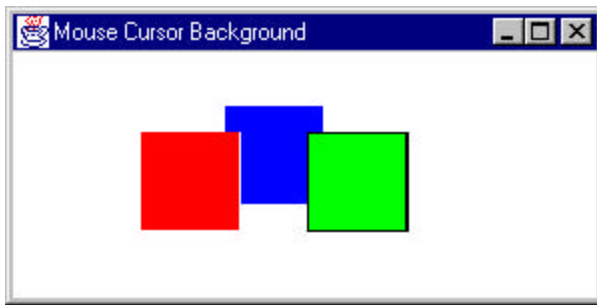


Figure 18-5: The MouseDrag.java program, showing that you cannot drag components on top of each other in a different order than you initially added them to the frame or container.

Another major limitation in dragging is that if you have placed controls inside a panel for purposes of layout, you cannot drag them outside that panel at all. Dragging only occurs within a single container, whether a frame, a panel or a window.

## Right and Center Button Mouse Clicks

There is no specific method in Java for receiving mouse clicks from more than one mouse button, because not all platforms Java runs on support more than one platform. Instead, you can AND the `BUTTON2_MASK` or `BUTTON3_MASK` with the `getModifiers()` value. Two button mice use Java buttons 1 and 3, not 1 and 2. In our `MouseRight.java` example program, we are going to pop up a `TipWindow` which displays some text. In our simple example, we'll simply display the name of the color of the window.

A Window is a kind of container like a Frame is, but without a title bar. It does have an independent existence however, and is not dependent on the positions of controls inside other panels or frames.

To pop up our Tip window, we modify the `mouseClicked` method as follows:

```
public void mousePressed(MouseEvent evt) {
    if((evt.getModifiers () &
        MouseEvent.BUTTON3_MASK ) != 0) {
        tip = new TipWindow(frm, color);
    }
}
```

```
    }
```

To create a Window, you need to make it the child of a Frame. This is clearly simple in the case of applications, but if you want a tip window in an applet, you must first create an invisible parent frame. Our TipWindow class is as follows:

```
public class TipWindow extends Window {
    private String cname = "";
    private Color color;
    public TipWindow(Frame frm, Color c) {
        super(frm);
        color = c;
        if(color == Color.red )
            cname="Red";
        if(color == Color.blue)
            cname = "Blue";
        if (color == Color.green)
            cname = "Green";

        add(new Label(cname));
        setBounds(frm.getBounds ().x+10,
                  frm.getBounds ().y+10,
                  100, 40);
        setVisible(true);
    }
}
```

Finally, we want to make sure our “tip” disappears when you raise the right mouse button. We don’t have to check which button was raised: if the reference to the TipWindow is not null, we’ll dispose of the window:

```
public void mouseReleased(MouseEvent evt) {
    if(tip != null)
        tip.setVisible(false);
    repaint();
}
```

Figure 18-5 shows a tip window popped up over our MouseRight frame.

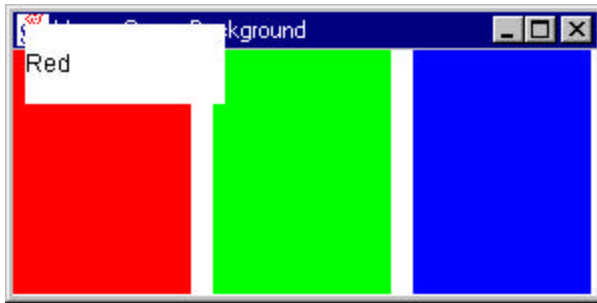


Figure 18-5: The TipWindow in the MouseRight.java example program.

## Summary

In this chapter we've looked in great detail at how to use the mouse. We've seen how to receive clicks, double clicks, drags, and right clicks. In the process we've extended a number of common controls to make them more useful. Now let's take a look at how to build our own controls from scratch.

## 19. Advanced Swing Concepts

### The JList Class

We briefly touched on the JList Swing class in our beginning chapter. There we simply noted that you can create a JList and give it an array or Vector holding the data you'd like to display. However, if you want to interact with the data in the list, you need to understand that the JList, as well as the JTable and JTree keep the data in a data model class which is then observed by the displaying list class. We'll look at the more advanced parts of these controls in this chapter.

### List Selections and Events

Regardless of how you manage the data that makes up a JList, you can set it to allow users to select a single line, multiple contiguous lines or separated multiple lines with the *setSelectionMode* method, where the arguments can be

```
SINGLE_SELECTION
SINGLE_INTERVALSELECTION
MULTIPLE_INTERVAL_SELECTION
```

You can then receive these events by using the *addListSelectionListener* method. Your program must implement the following method from the ListSelectionListener interface

```
public void valueChanged(ListSelectionEvent e)
```

In our JListLDemo.java example we display the selected list item in a text field:

```
public void valueChanged(ListSelectionEvent e) {
    text.setText((String)list.getSelectedValue());
}
```

This is shown in Figure 31.2.

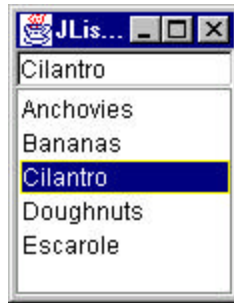


Figure 31.2 – A JList with the selected item displayed in the JTextField.

## Changing a List Display Dynamically

If you want a list to change dynamically during your program, the problem is somewhat more involved because the JList displays the data it is initially loaded with and does not update the display unless you tell it to. One simple way to accomplish this is to use the *setListData* method of JList to keep passing it a new version of the updated Vector after you change it. In the *JListDemo.java* program, we add the contents of the top text field to the list each time the Add button is clicked. All of this takes place in the action routine for that button:

```
public void actionPerformed(ActionEvent e)    {
    dlist.addElement(text.getText()); //add text from field
    list.setListData(dlist);           //send new Vector to list
    list.repaint();                   //and tell it to redraw
}
```

One drawback to this simple solution is that you are passing the entire Vector to the list each time and it must update its entire contents each time rather than only the portion that has changed. This brings us to the underlying ListModel that contains the data the JList displays.

When you create a JList using an array or Vector, the JList automatically creates a simple ListModel object which contains that data. Each ListModel object is a concrete subclass of the AbstractListModel class that defines the following simple methods:

```
void fireContentsChanged(Object source, int index0,
                          int index1)
void fireIntervalAdded(Object source, int index0, int index1)
void fireIntervalRemoved(Object source, int index0,
```



```
int index1)
```

The `AbstractListModel` class is declared as *abstract* so you must subclass it and create your own class, but there are no abstract methods you must implement. All of them have default implementations. You need only call those *fire* methods your program will be using. For example, as shown below we really only need to call the *fireIntervalAdded* method.

Our `ListModel` is an object that contains the data (in a `Vector` or other suitable structure) and notifies the `JList` whenever it changes. Here, the list model is just the following:

```
class JListData extends AbstractListModel {
    private Vector dlist;    //the color name list

    public JListData()      {
        dlist = new Vector();
        makeData();
    }
    public int getSize()    {
        return dlist.size();
    }
    private Vector makeData()
    {
        dlist = new Vector(); //create vector
        dlist.addElement("Anchovies"); //and add data
        //..add rest of names as before..
        return dlist;
    }
    public Object getElementAt(int index)    {
        return dlist.elementAt(index);
    }
    //add string to list and tell the list about it
    public void addElement(String s)        {
        dlist.addElement(s);
        fireIntervalAdded(this, dlist.size()-1, dlist.size());
    }
}
```

This `ListModel` approach is really an implementation of the Observer design pattern or the Model-View-Controller pattern. The data are in one class and the rendering or display methods in another class, and the communication between them triggers new display activity. We see this in the `JListMDemo` example program.

## Building an Adapter to Emulate the Awt List Class

Suppose you would like to write a simple list program that uses the `JList` class. Most of the methods you use for creating and manipulating the user interface remain the same. However, the `JFC JList` class is markedly different than the `AWT List` class. In fact, because the `JList` class was designed to represent far more complex kinds of lists, there are virtually no methods in common between the classes:

awt List class	JFC JList class
<code>add(String);</code>	---
<code>remove(String)</code>	---
<code>String[] getSelectedItems()</code>	<code>Object[] getSelectedValues()</code>

Both classes have quite a number of other methods and almost none of them are closely correlated. However, it is perfectly possible to write an adapter to make the `JList` class look like the `List` class and provide a rapid solution to our problem.

The `JList` class is a window container which has an array, vector or other `ListModel` class associated with it. It is this `ListModel` that actually contains and manipulates the data. Further, the `JList` class does not contain a scrollbar, but instead relies on being inserted in the viewport of the `JScrollPane` class. Data in the `JList` class and its associated `ListModel` are not limited to strings, but may be almost any kind of objects, as long as you provide the cell drawing routine for them. This makes it possible to have list boxes with pictures illustrating each choice in the list.

In our case, we are only going to create a class that emulates the `List` class, and that in this simple case, needs only the three methods we showed in the table above.

We can define the needed methods as an interface and then make sure that the class we create implements those methods:

```
public interface awtList {
    public void    add(String s);
    public void    remove(String s);
    public String[] getSelectedItems()
}
```

Interfaces are important in Java, because Java does not allow multiple inheritance as C++ does. So you cannot create a class that inherits from

JList and List at the same time. However, you can create a class which inherits from one class hierarchy and *implements* the methods of another. In most cases this is quite a powerful solution. Thus, by using the *implements* keyword, the class can take on methods and the appearance of being a class of either type.

## The Adapter

In this adapter approach, we create a class that *contains* a JList class but which implements the methods of the `awtList` interface above. Since the outer container for a JList is not the list element at all, but the JScrollPane that encloses it, we are really adding methods to a subclass of JScrollPane which emulate the methods of the List class. These methods are in the interface *awtList*.

So, our basic `JawtList` class looks like this:

```
public class JawtList extends JScrollPane
implements ListSelectionListener, awtList {
    private JList    listWindow;
    private JListData listContents;

    public JawtList(int rows) {
        listContents = new JListData();
        listWindow =  new JList(listContents);
        listWindow.setPrototypeCellValue("Abcdefg Hijkmnop");
        getViewport().add(listWindow);
    }
    //-----
    public void add(String s) {
        listContents.addElement(s);
    }
    //-----
    public void remove(String s) {
        listContents.removeElement(s);
    }
    //-----
    public String[] getSelectedItems() {
        Object[] obj = listWindow.getSelectedValues();
        String[] s  = new String[obj.length];
        for (int i =0; i < obj.length; i++)
            s[i] = obj[i].toString();
        return s;
    }
}
```

Note, however, that the actual data handling takes place in the `JListData` class. This class is derived from the `AbstractListModel`, which defines the following methods:

<code>addListDataListener(l)</code>	Add a listener for changes in the data.
<code>removeListDataListener(l)</code>	Remove a listener
<code>fireContentsChanged(obj, min,max)</code>	Call this after any change occurs between the two indexes min and max
<code>fireIntervalAdded(obj,min,max)</code>	Call this after any data has been added between min and max.
<code>fireIntervalRemoved(obj, min, max)</code>	Call this after any data has been removed between min and max.

The three *fire* methods are the communication path between the data stored in the `ListModel` and the actual displayed list data. Firing them causes the displayed list to be updated.

In this case, the `addElement`, `removeElement` methods are all that are needed, although you could imagine a number of other useful methods. Each time we add data to the *data* vector, we call the *fireIntervalAdded* method to tell the list display to refresh that area of the displayed list.

```
public class JListData extends AbstractListModel {
    private Vector data;

    public JListData() {
        data = new Vector();
    }
    //-----
    public int getSize() {
        return data.size();
    }
    //-----
    public Object getElementAt(int index) {
        return data.elementAt(index);
    }
    //-----
    public void addElement(String s) {
        data.addElement(s);
    }
}
```

```

        fireIntervalAdded(this, data.size()-1, data.size());
    }
//-----
    public void removeElement(String s) {
        data.removeElement(s);
        fireIntervalRemoved(this, 0, data.size());
    }
}

```

## A Sorted JList with a ListModel

If you would like to display a list that is always sorted, one simple way is to use the `DefaultListModel` to contain the data. This class implements the same methods as the `Vector` class, and notifies the `JList` whenever the data changes. So a complete program for a non-sorted list display can be as

```

// Creates a JList based on an unsorted DefaultListModel
public class ShowList extends JxFrame implements
ActionListener {
    String names[]= {"Dave", "Charlie", "Adam", "Edward",
"Barry"};
    JButton        Next;
    DefaultListModel ldata;
    int            index;

    public ShowList() {
        super("List of names");
        JPanel jp = new JPanel();
        getContentPane().add(jp);
        jp.setLayout(new BorderLayout());

        //create the ListModel
        ldata = new DefaultListModel();
        //Create the list
        JList nList = new JList(ldata);
        //add it to the scroll pane
        JScrollPane sc = new JScrollPane();
        sc.setViewportView().setView(nList);
        jp.add ("Center", sc);
        JButton Next = new JButton("Next");

        //add an element when button clicked
        JPanel bot = new JPanel();
        jp.add("South", bot);
        bot.add(Next);
        Next.addActionListener (this);
    }
}

```

```

        setSize(new Dimension(150, 150));
        setVisible(true);
        index = 0;
    }
    public void actionPerformed(ActionEvent evt) {
        if(index < names.length)
            ldata.addElement (names[index++]);
    }
}

```

In this program we add a “Next” button along the bottom which adds a new name each time it is clicked. The data are not sorted here, but it is pretty obvious that if we just subclass the `DefaultListModel`, we can have our sorted list and have the elements always sorted, even after new names are added.

So, if we create a class based on `DefaultListModel` which extends the `addElement` method and re-sorts the data each time, we’ll have our sorted list:

```

// This simple list model re-sorts the data every time
public class SortedModel extends DefaultListModel {
    private String[] dataList;

    public void addElement(Object obj) {
        //add to internal vector
        super.addElement(obj);
        //copy into array
        dataList = new String[size()];
        for(int i = 0; i < size(); i++) {
            dataList[i] = (String)elementAt(i);
        }
        //sort the data and copy it back
        Arrays.sort (dataList); //sort data
        clear(); //clear out vector

        //reload sorted data
        for(int i = 0; i < dataList.length; i++)
            super.addElement(dataList[i]);

        //tell JList to repaint
        fireContentsChanged(this, 0, size());
    }
}

```

We see this list in Figure 31.3. The names are added one at a time in non-alphabetic order each time you click on the Next button, but sorted before being displayed.



Figure 31.3: Sorted data using SortedListModel

## Sorting More Complicated Objects

Suppose, now that we want to display both first and last names, and want to sort by the last names. In order to do that we have to create an object which holds first and last names, but which can be sorted by last name. And how do we do this sort? Well, we could do it by brute force, but in Java any class which implements the Comparable interface can be sorted by the Arrays.sort method. And the Comparable interface is just one method:

```
public int compareTo(Object obj)
```

where the class returns a negative value, zero or a positive value depending on whether the existing object is less than, equal to or greater than the argument object. Thus, we can create a Person class with this interface just as simply as

```
public class Person implements Comparable {

    private String fname, lname;

    public Person(String name) {
        //split name apart
        int i = name.indexOf (" ");
        fname = name.substring (0, i).trim();
        lname = name.substring (i).trim();
    }
    public int compareTo(Object obj) {
        Person to = (Person)obj;
        return lname.compareTo (to.getLname ());
    }
    public String getLname() {
        return lname;
    }
}
```

```

    }
    public String getFrname() {
        return frname;
    }
    public String getName() {
        return getFrname()+" "+getLname();
    }
}

```

Note that the **compareTo** method simply invokes the compareTo method of the last name String objects.

The other change we have to make is that our data model has to return both names, so we extend the getElementAt method:

```

// Data model which uses and sorts Person objects
public class SortedModel extends DefaultListModel {
    private Person[] dataList;

    public void addElement(Object obj) {
        Person per = new Person((String) obj);
        super.addElement(per);
        dataList = new Person[size()];

        //copy the Persons into an array
        for(int i = 0; i < size(); i++) {
            dataList[i] = (Person)elementAt(i);
        }

        //sort them
        Arrays.sort (dataList);

        //and put them back
        clear();
        for(int i = 0; i < dataList.length; i++)
            super.addElement(dataList[i]);
        fireContentsChanged(this, 0, size());
    }
    public Object getElementAt(int index) {
        //returns both names as a string
        Person p = dataList[index];
        return p.getName();
    }
    public Object get(int index) {
        return getElementAt(index);
    }
}

```

You see the resulting sorted names below:





Figure 31.4 – Sorted list using Comparable interface to sort on last names.

## Getting Database Keys

Now one disadvantage of a sorted list is that clicking on the  $n$ th element does not correspond to selecting the  $n$ th element, since the sorted elements can be in an order that is different from the order you added them to the list. This, if you want to get a database key corresponding to a particular list element (here a person) in order to display detailed information about that person, you have to keep the database key inside the person object. This is analogous to but considerably more flexible than the Visual Basic approach where you can keep only one key value for each list element. Here you could keep several items in the person object if that is desirable. In the figure below, we double click on one person's name and pop up a window containing his phone number.

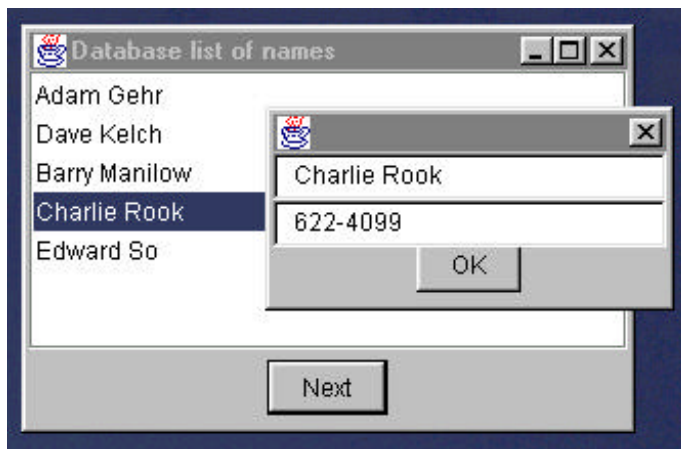


Figure 31.5 – A pop up list showing details appears when you double click on a name.

In order to pop up a window when you double click on a JList, you must add a mouse listener to the JList object. In our code, we created a class called `MouseListener` which carries out the listening and produces the popup. First we add the `MouseListener` by

```
nList.addMouseListener(new mouseListener(nList, ldata, db,
this));
```

where **db** represents our database and **ldata** the list data model. The complete `MouseListener` class is shown below:

```
public class mouseListener extends MouseAdapter {
    private JList          nList;
    private DataBase       db;
    private SortedModel    lData;
    private JFrame         jxf;

    public mouseListener(JList list, SortedModel ldata,
                        DataBase dbase, JFrame jf) {
        nList = list;
        db    = dbase;
        jxf   = jf;
        lData = ldata;
    }
    //
    public void mouseClicked(MouseEvent e) {
        if (e.getClickCount () == 2) {
            //mouse double clicked-
            //get database key for this person
            int index = nList.locationToIndex (e.getPoint());
            int key   = lData.getKey (index);
            //display pop up dialog box
            Details details = new Details(jxf,
                db.getName (key), db.getPhone (key));
            details.setVisible(true);
        }
    }
}
```

Note that since the JList control has no specific methods for detecting a mouse double click, we check the `mouseClicked` event method and see if the click count is 2. If it is, we query the database for the name and phone number of the person with that key. In the example code accompanying this article, we simulate the database with a simple text file so that the code example does not become unwieldy.

## Pictures in our List Boxes

The JList is flexible in yet another way. You can write your own cell rendering code to display anything you want in a line of a list box. So you can include images, graphs or dancing babies, if you want. All you have to do is create a cell rendering class that implements the ListCellRenderer interface. This interface has but one method called **getListCellRendererComponent** and is quite simple to write. By simply extending the JLabel class, which itself allows for images as well as text, we can display names and images alongside each name with very little effort. We assume that each Person object now contains the image to display:

```
public class cellRenderer extends JLabel implements
ListCellRenderer {

    public Component getListCellRendererComponent(JList list,
        Object value, int index, boolean isSelected,
        boolean hasFocus) {
        Person p = (Person)value; //get the person
        setText(p.getName ()); //display name
        setIcon(p.getIcon ()); //and image
        if(isSelected)
            setBackground(Color.lightGray );
        else
            setBackground(Color.white );
        return this;
    }
}
```

Of course we also connect this cell renderer class to the JList with this simple method call:

```
nList.setCellRenderer (new cellRenderer());
```

The resulting display is shown in Figure 31.6



Figure 31.6—A sorted list with pictures added using a custom cell renderer.

## The JTable Class

The JTable class is much like the JList class, in that you can program it very easily to do simple things. Similarly, in order to do sophisticated things, you can create a class derived from the AbstractTableModel class to hold your data.

## A Simple JTable Program

In the simplest program, you just create a rectangular array of objects and use it in the constructor for the JTable. You can also include an array of strings to be used as column labels for the table.

```
public class SimpleTable extends JFrame {
    public SimpleTable() {
        super("Simple table");
        JPanel jp = new JPanel();
        getContentPane().add(jp);
        Object[] [] musicData = {
            {"Tschaikovsky", "1812 Overture", new
Boolean(true)},
            {"Stravinsky", "Le Sacre", new Boolean(true)},
            {"Lennon", "Eleanor Rigby", new Boolean(false)},
            {"Wagner", "Gotterdammerung", new Boolean(true)}
        };
        String[] columnNames = {"Composer", "Title",
            "Orchestral"};
        JTable table = new JTable(musicData, columnNames);
    }
}
```

```

JScrollPane sp = new JScrollPane(table);
table.setPreferredScrollableViewportSize(
    new Dimension(250,170));
jp.add(sp);

setSize(300,200);
setVisible(true);
}

```

This produces the simple table display in Figure 32.1.:

Composer	Title	Orchestral
Tschaikovsky	1812 Overture	true
Stravinsky	Le Sacre	true
Lennon	Eleanor Rigby	false
Wagner	Gotterdam...	true

Figure 32.1 – A simple table display

This table has all cells editable and displays all the cells using the *toString* method of each object. Of course, like the *JList* interface, this simple interface to *JTable* creates a data model object under the covers. In order to produce a more flexible display you need to create that data model yourself.

You can create a table model by extending the *AbstractTableModel* class. All of the methods have default values and operations except the following 3 which you must provide:

```

public int      getRowCount();
public int      getColumnCount();
public Object   getValueAt(int row, int column);

```

However, you can gain a good deal more control by overriding a couple of other methods. You can use the method

```

public boolean isCellEditable(int row, int col)

```

to protect some cells from being edited. If you want to allow editing of some cells, you must override the implementation for the method

```
public void setValueAt(Object obj, int row, int col)
```

Further, by adding a method which returns the data class of each object to be displayed, you can make use of some default cell formatting behavior. The `JTable`'s default cell renderer displays

- Numbers as right-aligned labels
- `ImageIcons` as centered labels
- Booleans as checkboxes
- Objects using their *toString* method

To change this, you simply need to return the class of the objects in each column:

```
public Class getColumnClass( int col) {
    return getValueAt(0, col).getClass();
}
```

Our complete table model class creates exactly the same array and table column captions as before and implements the methods we just mentioned:

```
public class MusicModel extends AbstractTableModel {

    String[] columnNames = {"Composer", "Title",
"Orchestral"};

    Object[] [] musicData = {
        {"Tschaikovsky", "1812 Overture", new Boolean(true)},
        {"Stravinsky", "Le Sacre", new Boolean(true)},
        {"Lennon", "Eleanor Rigby", new Boolean(false)},
        {"Wagner", "Gotterdammerung", new Boolean(true)}
    };

    private int rowCount, columnCount;
    //-----
    public MusicModel(int rowCnt, int colCnt) {
        rowCount = rowCnt;
        columnCount = colCnt;
    }
    //-----
    public String getColumnName(int col) {
        return columnNames[col];
    }
}
```

```

//-----
public int getRowCount() {
    return rowCount;
}
public int getColumnCount() {
    return columnCount;
}
//-----
public Class getColumnClass( int col) {
    return getValueAt(0, col).getClass();
}
//-----
public boolean isCellEditable(int row, int col) {
    return(col > 1);
}
//-----
public void setValueAt(Object obj, int row, int col) {
    musicData[row][col] = obj;
    fireTableCellUpdated(row, col);
}
//-----
public Object getValueAt(int row, int col) {
    return musicData[row][col];
}
}

```

The main program simply becomes the code called in the constructor when the class is instantiated from *main*.

```

public class ModelTable extends JxFrame {
    public ModelTable() {
        super("Simple table");
        JPanel jp = new JPanel();
        getContentPane().add(jp);
        JTable table = new JTable(new MusicModel(4, 3));
        JScrollPane sp = new JScrollPane(table);
        table.setPreferredSize(new
            Dimension(250,170));

        jp.add(sp);

        setSize(300,200);
        setVisible(true);
    }
//-----
    static public void main(String argv[]) {
        new ModelTable();
    }
}

```

As you can see in our revised program display in Figure 32.2, the boolean column is now automatically rendered as check boxes. We have also only allowed editing of the right-most column by overriding the *isCellEditable* method to disallow it for columns 0 and 1.



Composer	Title	Orchestral
Tschaikovsky	1812 Overture	<input checked="" type="checkbox"/>
Stravinsky	Le Sacre	<input checked="" type="checkbox"/>
Lennon	Eleanor Rigby	<input type="checkbox"/>
Wagner	Gotterdam...	<input checked="" type="checkbox"/>

Figure 32.2 – A JTable display using a datamodel which returns the datatype of each column and controls the data display accordingly.

Just like the Listmodel does for JList objects, the TableModel class is a class which holds and manipulates the data and notifies the Jtable whenever it changes. Thus, the JTable is an Observer pattern, operating on the TableModel data.

## Cell Renderers

Each cell in a table is rendered by a cell renderer. The default renderer is a JLabel, and it may be used for all the data in several columns. Thus, these cell renderers can be thought of as Flyweight pattern implementations. The JTable class chooses the renderer according to the object's type as we outlined above. However, you can change to a different rendered, such as one that uses another color, or another visual interface quite easily.

Cell renderers are registered by type of data:

```
table.setDefaultRenderer(String.class, new ourRenderer());
```

and each renderer is passed the object, selected mode, row and column using the only required public method:

```
public Component getTableCellRendererComponent(JTable jt,
        Object value, boolean isSelected,
```



```
boolean hasFocus, int row, int column)
```

One common way to implement a cell renderer is to extend the JLabel type and catch each rendering request within the renderer and return a properly configured JLabel object, usually the renderer itself. The renderer below displays cell (1,1) in boldface red type and the remaining cells in plain, black type:

```
public class ourRenderer extends JLabel
    implements TableCellRenderer {
    Font bold, plain;
    public ourRenderer() {
        super();
        setOpaque(true);
        setBackground(Color.white);
        bold = new Font("SansSerif", Font.BOLD, 12);
        plain = new Font("SansSerif", Font.PLAIN, 12);
        setFont(plain);
    }
    public Component getTableCellRendererComponent(JTable jt,
        Object value, boolean isSelected,
        boolean hasFocus, int row, int column) {
        setText((String)value);
        if (row == 1 && column == 1) {
            setFont(bold);
            setForeground(Color.red);
        } else {
            setFont(plain);
            setForeground(Color.black);
        }
        return this;
    }
}
```

The results of this rendering are shown in Figure 32.3.



Composer	Title	Orchestral
Tschaikovsky	1812 Overture	<input checked="" type="checkbox"/>
Stravinsky	<b>Le Sacre</b>	<input checked="" type="checkbox"/>
Lennon	Eleanor Rigby	<input type="checkbox"/>
Wagner	Gotterdam...	<input checked="" type="checkbox"/>

Figure 32.3 – The music data using a CellRenderer that changes the color and font of row 1, column 1.

In the simple cell renderer shown above the renderer is itself a JLabel which returns a different font, but the same object, depending on the row and column. More complex renderers are also possible where one of several already-instantiated objects is returned, making the renderer a Component Factory.

## Rendering Other Kinds of Classes

Now lets suppose that we have more complicated classes that we want to render. Suppose we have written a document mail system and want to display each document according to its type. However, the documents don't have easily located titles and we can only display them by author and type. Since we intend that each document could be mailed, we'll start by creating an interface Mail to describe the properties the various document types have in common:

```
public interface Mail {
    public ImageIcon getIcon();
    public String getLabel();
    public String getText();
}
```

So each type of document will have a simple label (the author) and a method for getting the full text (which we will not use here). Most important, each document type will have its own icon, which you can obtain with the *getIcon* method.

For example, the NewMail class would look like this:

```
public class NewMail implements Mail {
    private String label;

    public NewMail(String mlabel) {
        label = mlabel;
    }
    public ImageIcon getIcon () {
        return new ImageIcon("images/mail.gif");
    }
    public String getText() {
        return "";
    }
    public String getLabel() {
        return label;
    }
}
```

```
}

```

The renderer for these types of mail documents is just one which gets the icon and label text and uses the `DefaultTableCellRenderer` (derived from `JLabel`) to render them:

```
public class myRenderer extends DefaultTableCellRenderer {
    private Mediator md;

    public myRenderer(Mediator med) {
        setHorizontalAlignment(JLabel.LEADING);
        md = med;
    }
    //-----
    public Component getTableCellRendererComponent(
       .JTable table, Object value, boolean isSelected,
        boolean hasFocus, int row, int col) {
        if (hasFocus) {
            setBackground(Color.lightGray );
            md.tableClicked ();
        } else
            setBackground(new Color(0xffffce));
        if (value != null) {
            Mail ml = (Mail) value;
            String title = ml.getLabel ();
            setText(title);           //set the text
            setIcon(ml.getIcon ());  //and the icon
        }
        return this;
    }
}

```

Since one of the arguments to the `getTableCellRendererComponent` method is whether the cell is selected, we have an easy way to return a somewhat different display when the cell is selected. In this case, we return a gray background instead of a white one. However, we could set a different `Border` as well if we wanted to. A display of the program is shown in Figure 32.4.

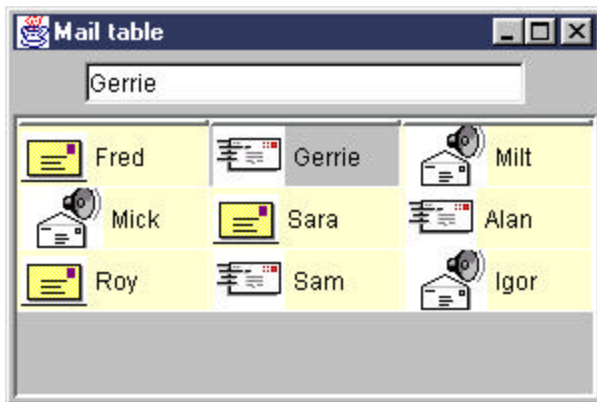


Figure 32.4 - Rendering objects which implement the Mail interface, using their icon and text properties. Note the selected cell is shown with a gray background.

## Selecting Cells in a Table

You can select a row, separated rows, a contiguous set of rows or single cells of a table, depending on the list selection mode you choose. In this example, we want to be able to select single cells, so we choose the single selection mode:

```
setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

If we want to take a certain action when table cell is selected we get the `ListSelectionModel` from the table and add a list selection listener to it.

```
ListSelectionModel lsm = getSelectionModel();
lsm.addListSelectionListener (new
TableSelectionListener (med));
```

The `TableSelectionListener` class we create just implements the `valueChanged` method and passes this information to a Mediator class:

```
public class TableSelectionListener implements
ListSelectionListener {

    private Mediator md;

    public TableSelectionListener(Mediator med) {
        md= med;
    }
}
```

```

public void valueChanged(ListSelectionEvent e) {
    ListSelectionModel lsm =
        (ListSelectionModel)e.getSource();
    if( ! lsm.isSelectionEmpty ())
        md.tableClicked();
}
}

```

A Mediator class is one of the best ways to deal with interactions between separate visual objects. In this case, it just fetches the correct label for this class and displays it in the text field at the top of the window, as shown in Figure 4. Here is the entire Mediator class:

```

public class Mediator {
    Ftable          ftable;
    JTextField      txt;
    int             tableRow, tableColumn;
    FolderModel     fmodel;

    public Mediator ( JTextField tx) {
        txt = tx;
    }
    public void setTable(Ftable tbl) {
        ftable = tbl;
    }
    //-----
    public void tableClicked() {
        int row = ftable.getSelectedRow ();
        int col = ftable.getSelectedColumn ();
//don't refresh if not changed
        if ((row != tableRow) || (col != tableColumn)) {
            tableRow = row;
            tableColumn = col;
            fmodel = ftable.getTableModel ();
            Mail ml = fmodel.getDoc(row, col);
            txt.setText (ml.getLabel ());
        }
    }
}
}

```

## Programs on the CD-ROM

\Swing\Table\SimpleTable\SimpleTable.java	Creates simple table from 2 dimensional string array.
\Swing\Table\RenderTable\ModelTable.java	Creates table using a TableModel, allowing the boolean column to be

	displayed as check boxes.
<code>\Swing\Table\RenderTable\RenderTable.java</code>	Creates simple table with <code>TableModel</code> and cell renderer which renders one cell in bold red.
<code>\Swing\Table\ImageTable\MailTable.java</code>	Creates a set of mail images in a 3 x 2 table.

## The JTree Class

Much like the `JTable` and `JList`, the `JTree` class consists of a data model and an observer. One of the easiest ways to build up the tree you want to display is to create a root node and then add child nodes to it and to each of them as needed. The `DefaultMutableTreeNode` class is provided as an implementation of the `TreeNode` interface.

You create the `JTree` with a root node as its argument

```
root = new DefaultMutableTreeNode("Foods");
JTree tree = new JTree(root);
```

and then add each node to the root, and additional nodes to those to any depth. The following simple program produces a food tree list by category. Note that we use the `JxFrame` class we derived from `JFrame` at the beginning of this chapter.

```
public class TreeDemo extends JxFrame {
    private DefaultMutableTreeNode root;
    public TreeDemo() {
        super("Tree Demo");
        JPanel jp = new JPanel(); // create interior panel
        jp.setLayout(new BorderLayout());
        getContentPane().add(jp);

        //create scroll pane
        JScrollPane sp = new JScrollPane();
        jp.add("Center", sp);

        //create root node
        root = new DefaultMutableTreeNode("Foods");
        JTree tree = new JTree(root); //create tree
        sp.getViewport().add(tree); //add to scroller

        //create 3 nodes, each with three sub nodes
```

```

addNodes("Meats", "Beef", "Chicken", "Pork");
addNodes("Vegies", "Broccoli", "Carrots", "Peas");
addNodes("Desserts", "Charlotte Russe",
        "Bananas Flambe", "Peach Melba");

setSize(200, 300);
setVisible(true);
}
//-----
private void addNodes(String b, String n1, String n2,
                    String n3) {
    DefaultMutableTreeNode base =
    new DefaultMutableTreeNode(b);
    root.add(base);
    base.add(new DefaultMutableTreeNode(n1));
    base.add(new DefaultMutableTreeNode(n2));
    base.add(new DefaultMutableTreeNode(n3));
}

```

The tree it generates is shown in Figure 33.1.

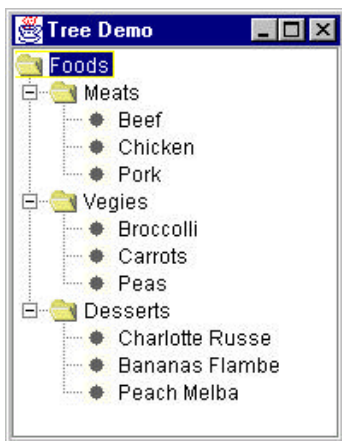


Figure 33.1 – A simple JTree display.

If you want to know if a user has clicked on a particular line of this tree, you can add a `TreeSelectionListener` and catch the `valueChanged` event. The `TreePath` you can obtain from the `getPath` method of the `TreeSelectionEvent` is the complete path back to the top of the tree. However the `getLastPathComponent` method will return the string of the line the user actually selected. You will see that we use this method and display in the Composite pattern example.

```

public void valueChanged(TreeSelectionEvent evt) {

```

```

TreePath path = evt.getPath();
String selectedTerm =
    path.getLastPathComponent().toString();

```

## The TreeModel Interface

The simple tree we build above is based on adding a set of nodes to make up a tree. This is an implementation of the `DefaultTreeModel` class which handles this structure. However, there might well be many other sorts of data structure that you'd like to display using this tree display. To do so, you create a class of your own to hold these data which implements the `TreeModel` interface. This interface is very relatively simple, consisting only of

```

void addTreeModelListener(TreeModelListener l);
Object getChilds(Object parent, int index);
int getChildCount(Object parent);
int getIndexof Child(Object parent, Object child);
Object getRoot();
boolean isLeaf(Object);
void removeTreeModelListener(TreeModelListener l);
void value ForPathChanges(TreePath path, Object newValue);

```

Note that this general interface model does not specify anything about how you add new nodes, or add nodes to nodes. You can implement that in any way that is appropriate for your data.

## Programs on the CD-ROM

\Swing\Tree\TreeDemo.java	Creates a simple tree from <code>DefaultMutableTreeNode</code> .
---------------------------	--

## Summary

In this brief chapter, we've touched on some of the advanced functions of the more common JFC controls, and noted how similarly they all use a data model class which the visual control renders.





## 20. Using Packages

The Java package provides a convenient way to group classes in separate name spaces. This makes it possible to have classes with the same name in each package. To make use of classes within a given package, you need to either refer to them by a name which includes the package name:

```
pkg.myClass.method();
```

or use the *import* statement to indicate that you will be using files from a given package:

```
import pkg;
myClass.method();
```

We've already seen examples of this, of course, in using the java package classes. We could write

```
lb = new java.awt.Label("label");
```

or we more conventionally write

```
import java.awt.*;
lb = new Label("label");
```

### The Java Packages

The Java classes are divided into several packages as we have seen

```
java.lang
java.awt
java.io
java.net
java.util
java.applet
java.awt.image
java.awt.event
javax.swing
```

You can import an entire package as we have been doing by

```
import java.awt.*;
```

or just import the classes you will be using

```
import java.awt.Label;
```

There is no difference between these two approaches in the size of the executables or their speed, but the compilation speed may be somewhat faster if you only import the classes from a package that you will be using. Considering how fast Java compilers have become, this is of much less import than it was a few years earlier.

While Java compilers in general require that you specifically import any packages you use, all compilers automatically import the `java.lang.*` class since all of the basic language elements are defined in this class.

## The Default Package

So far, we have not been using any package statements and have not written any programs that are part of a named package. All classes without a package statement are automatically made part of the default package.

## How Java Finds Packages

Java uses two methods to locate packages you want to include in your program: the extension path and the class path. You can set the class search path using the `CLASSPATH` environment variable

```
set CLASSPATH = d:\appClasses;
```

You can set these by putting the above statement into your `autoexec.bat` file, or by typing it on the keyboard. If you want to add the path `c:\testlib`, you can type

```
set CLASSPATH = d:\testlib;%CLASSPATH%
```

to append the `testlib` directory to the front of the classpath. You can do the same thing from the `java` command line:

```
java -classpath c:\testlib filename
```

In Windows NT and 2000, you set the `CLASSPATH` variable in the System application of the Control Panel.

## The Extension Folder

You can also place any classes or libraries you want to access in the

`\Javasoft\jre\lib\ext`

folder. This is usually installed in Windows systems under `c:\Program Files`. Any classes or jar files placed there will be accessible to both the compiler and the java virtual machine.

## Jar Files

Java uses a special kind of Zip file, called a Jar file (for Java Archive file). These files can contain several packages of classes in a single unit, and thus provide you with a simple way to distribute your code. Java also uses Jar files for the Java runtime classes. You will find that the file `rt.jar` contains all these classes. Since this is a file in Zip file format, you can view this file (or even change it) using programs like WinZip.

The Java file `rt.jar` is usually in the `\Javasoft\lib` directory and is not a zip file in the usual sense: the files are not compressed. Figure 20-1 shows the WinZip display of the contents of this file. Note that it is not compressed and that the directory structure is clearly preserved.

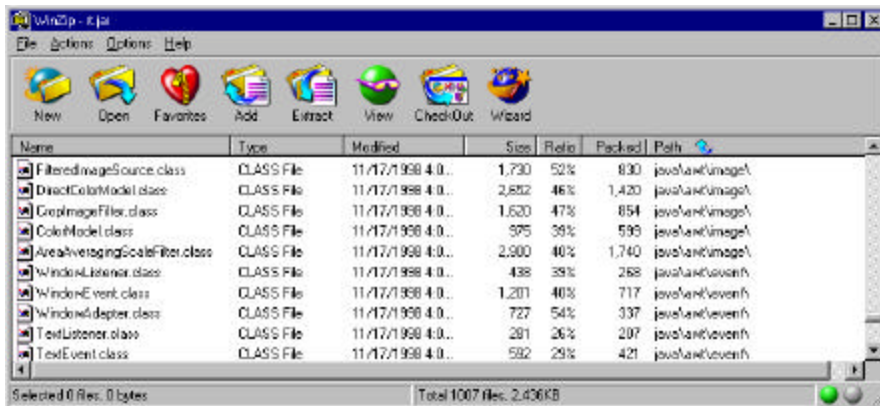


Figure 20-1: A WinZip display of the directory structure of part of the Java classes.zip.

## Creating Your Own Package

Let's suppose we want to create our own package, named `com.labsoftware.filio` which contains our `InputFile` and `OutputFile` classes, and another package called `com.labsoftware.gui` which contains our `Xframe` and `JxFrame` classes. We start by creating a directory structure:

```
d:\projects\com\labsoftware\filio
```

and

```
d:\projects\com\labsoftware\gui
```

We add to the classpath a pointer to the directory above: to the projects directory:

```
set classpath=d:\projects;%classpath%
```

Then we can put classes directly in the projects directory or in directories under that directory. If the classes are *in* that directory, we add an import statement pointing to that directory:

```
import com.labsoftware.filio.*;
```

Despite the asterisk, indicating all the classes it finds, this only includes reference to classes in the `\filio` directory. It does not include directories under that directory.

If you want to create additional classes in directories under that directory, you need to import those directories specifically as well:

```
import com.labsloftware.filio.events.*;
```

where this implies that the classes to be imported are in the

```
d:\projects\com\labsoftware\filio\events
```

directory. Note that the import statement is case sensitive and must accurately reflect the case of the directories.

## The package Statement

To put classes in this package, you must include a package statement in each class file which reflects its directory position. The package statement must be the first non-comment line in each source file and must include the directory structure of the package.

If the class is in the filio directory, each file must start with the statement:

```
package com.labsoftware.filio;
```

If the classes are in subsidiary directories they must start with, for example:

```
package com.labsoftware.filio.events;
```

You are not restricted to a single class in any directory but all of them must include the package statement to be recognized during compilation of any programs that import them.

## An Example Package

On your example disk you will find the directories com\labsoftware\filio and com\labsoftware\gui.. In the filio directory is a version of our InputFile and OutputFile classes which contains the package statement

```
package com.labsoftare.filio;
```

In the pacakges directory of your example disk is the program FileTest.java which contains the statement

```
import java.io.*;
import com.labsoftware.filio.*;
//Illustrates use of packages
public class FileTest {
    public FileTest() {
        try {
            InputFile inf = new InputFile("foo.txt");
            OutputFile outf = new OutputFile("fout.txt");
            String s = inf.readLine();
            while (s != null) {
                outf.println(s);
                s=inf.readLine();
            }
            inf.close();
        }
    }
}
```

```

        outf.close();
    }
    catch (IOException e) {
        System.out.println("File IO error:"+
            e.getMessage());
    }
}
//-----
static public void main(String argv[]) {
    new FileTest();
}
}

```

Since the directories `com\labsoftware\filio` are right under the `FileTest` directory, you will be able to compile and run this program without resorting to any class path manipulation.

## Class Visibility within Packages

Packages provide an additional level of visibility of methods between classes within the class. Any method or variable which is declared as `public` is of course visible to any other class. And, as before any method or variable declared as `private` is as usual only visible within the class. However, any variable or method is neither declared as `public` or `private` is said to have *package* visibility. Any other class in the package has access to these variables and methods as if they were `public`. This is selfdom a good idea, and you should resolutely declare all class level variables `private` to make sure they aren't inadvertently accessed through this mechanism.

## Making Your Own Jar Files

Combining a set of class files into a jar file can be more efficient. While the total space they occupy is the same, disk file and network overhead in reading a single jar file during compilation or the runtime loading of classes will probably be less. More to the point, you don't have to require the user of your program to copy a set of classes into a directory structure on his computer to use your program. Instead he need only copy one jar file to any place specified in his classpath or into the `jre\lib\ext` directory.

Let's suppose we want to make a jar file consisting of the two `filio` classes and the two `gui` classes. The file structure is

- com
  - labsoftware
    - filio
- InputFile.class
  - OutputFile.class
- gui
  - Xframe.class
  - JxFrame.class

We can create the jar file using the jar command line utility. If we issue a command from the command window in the directory just above the com directory, we can create a jar file of all the data in the directories below it:

```
jar -c0f lsoft.jar com
```

This creates a new file lsoft.jar without zip compression, using all the files in the folders below com. This is not limited to the class files, but we can easily use WinZip to sort them by file type and delete those we don't need as shown in Figure 20-2.

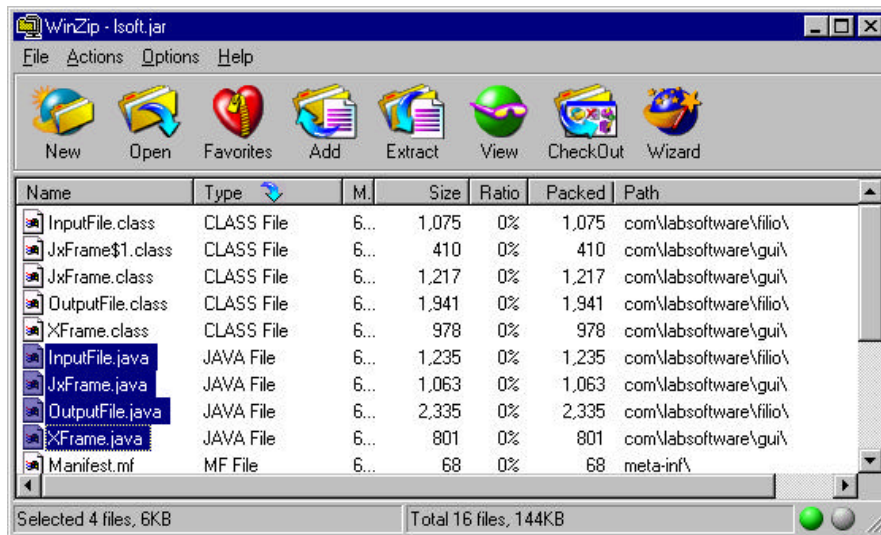


Figure 20-2: The WinZip display of our simple lsoft.jar file.



## Using a Jar File

You use the classes in a jar file by simply referring to them. The simple program `FileTest.java` we showed above will work the same with a jar file in the same directory as the `FileTest.class` file, or with the `ltest.jar` file in the extensions directory.

## Summary

In this chapter, we've looked at how packages and how jar archive files are constructed. We've seen that packages allow us to subdivide the name space as well as to group like, useful functions.

## 21. Writing Better Object Oriented Programs

A simple Java program with 3 radio buttons and a list box does not seem like much of a challenge for good programming techniques. But it turns out that it can be quite a useful illustration of how to write better programs. The point of this program is to display different things in the list box depending on which button was selected, and to print out that list when a Print button was selected. This is easy to do in Java, but it can be used to exemplify a whole bunch of OO programming concepts.

Let's start by defining a simple problem. We have a list of kids, both boys and girls, and we'd like the option of displaying the girls, the boys, or both. A simple GUI for this problem is shown in Figure 1.

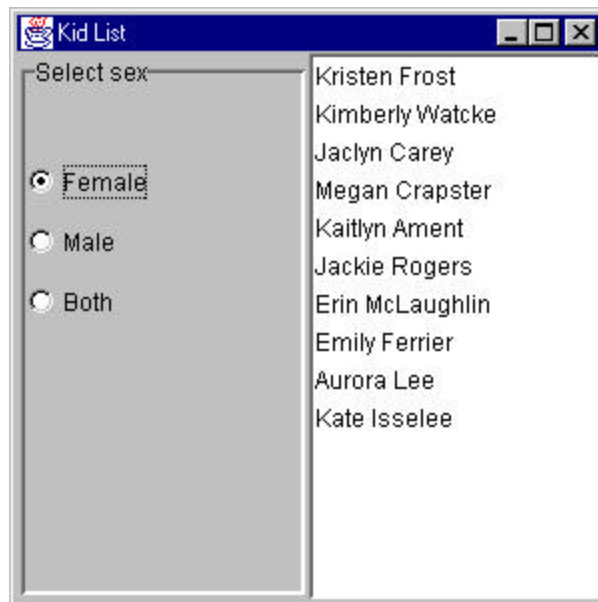


Figure 1 – The Swing window showing the results of clicking on the Female radio button.

## A Simple Implementation

Let's write this program in the simplest way possible, the way it might occur to you when you start out. We'll use the Swing classes, because they look a lot nicer on the screen, but we'll take some shortcuts. We have previously described a simple wrapper that makes the JList easier to use, in a class called JawsList, because it has similar properties to the original awt List object. We'll use that for our list box. Then, we'll put the three JRadioButtons in a BorderLayout on the left side as shown above, and surround it with a TitledBorder. This is done below, and represents a fairly straightforward way of constructing a Swing window.

```
public class ShowList extends JFrame implements
ActionListener {
    private JRadioButton female, male, both;
    private JawsList kidList;
    private Swimmer sw;
    private Vector swimmers;

    public ShowList() {
        super("Kid List");
        JPanel jp = new JPanel(); // create interior panel
        jp.setLayout(new GridLayout(1,2));
        getContentPane().add(jp);
        JPanel lp = new JPanel();
        lp.setLayout(new BorderLayout(lp, BorderLayout.Y_AXIS));
        jp.add(lp);

        //create titled border around radio button panel
        Border bd = BorderFactory.createBevelBorder
(BevelBorder.LOWERED );
        TitledBorder tl = BorderFactory.createTitledBorder
(bd, "Select sex");
        lp.setBorder(tl);

        //put bevel border around list
        kidList = new JawsList(20);
        kidList.setBorder (BorderFactory.createBevelBorder
(BevelBorder.LOWERED));
        jp.add(kidList);

        //add in the radio buttons
        female = new JRadioButton("Female");
        male = new JRadioButton("Male");
        both = new JRadioButton("Both");

        //keep them all together
        ButtonGroup grp = new ButtonGroup();
```

```

grp.add(female);
grp.add(male);
grp.add(both);

//make sure they all receive clicks
female.addActionListener (this);
male.addActionListener (this);
both.addActionListener (this);

//space the buttons out
lp.add(Box.createVerticalStrut (30));
lp.add(female);
lp.add(Box.createVerticalStrut (5));
lp.add(male);
lp.add(Box.createVerticalStrut (5));
lp.add(both);

```

### Listing 1 – Setting up the Swing

Note that the base class is derived from `JxFrame`, a class described previously which includes automatic setting of the look and feel, and setting the window to exit when the Close box is clicked. This class implements the `ActionListener` interface, and each of the radio buttons has been told that this frame is the listener for their actions. We also can use the `InputFile` class to read in a list of kids from a file along with their sexes along with other information we don't use here. We bury this in a `Swimmer` class which parses each line of that file.

Now, the simplest way to write this program is to carry out the loading of the data into the list box in the `actionPerformed` method:

```

public void actionPerformed(ActionEvent evt) {
    Object obj = evt.getSource ();
    if(obj == female)
        loadFemales();
    if(obj == male)
        loadMales();
    if(obj == both)
        loadBoth();
}

```

Then, we have three code loading methods for females, males and both. They look like this:

```

private void loadFemales() {
    kidList.clear();
    Enumeration enum = swimmers.elements();
}

```

```

        while(enum.hasMoreElements ()) {
            Swimmer sw = (Swimmer)enum.nextElement ();
            if (sw.isFemale ()) {
                kidList.add (sw.getName ());
            }
        }
    }
}

```

H.L. Mencken noted that for every problem, there is a solution that is neat, simple, and wrong. This is such a case. Even though the code works fine in this example, it is about as far from being object oriented as you can imagine. Whenever you see a set of if statements deciding which thing to do next, as we have in the `actionPerformed` method above, you should immediately suspect that you have not written the best possible OO program.

## Taking Command

Let's see what we can do to make this a little less tacky. Suppose we derive 3 button subclasses from `JRadioButton` and have each implement the `Command` interface. The `Command` interface just says that each class will have an `Execute` method:

```

public interface Command {
    public void Execute();
}

```

So, we'll make a `FemaleButton`, a `MaleButton` and a `BothButton`, each of which have an `Execute` method that loads the list with the right data. Here's the female version:

```

public class FemaleButton extends JRadioButton implements
Command {
    Vector swimmers;
    JawsList kidList;

    public FemaleButton(String title, Vector sw, JawsList
klist) {
        super(title);
        swimmers = sw;
        kidList = klist;
    }
    public void Execute() {
        Enumeration enum = swimmers.elements();
        kidList.clear();
        while(enum.hasMoreElements ()) {

```

```

        Swimmer sw = (Swimmer)enum.nextElement ();
        if (sw.isFemale ()) {
            kidList.add (sw.getName ());
        }
    }
}

```

Note that we pass in the instance of the kidList list box and of the Vector of kids and add the right ones to the list box when Execute is called. This approach greatly simplifies the actionPerformed method. Since all 3 buttons have an Execute method, they can all be Command objects, and we can just call that Execute method.

```

public void actionPerformed(ActionEvent evt) {
    Command cmd = (Command)evt.getSource ();
    cmd.Execute ();
}

```

Now there is no testing of buttons, since each one knows the right thing to do in its Execute method.

## Making Better Choice

But we can still do better than this. We have more or less the same code in the 3 button classes, each of which loads the list with something based on a decision that the button makes. We really ought to separate the interface from the data better than that. Buttons themselves should not be making decisions. They ought only to implement the visual logic needed to display the results of decisions made elsewhere.

So, we should consider replacing that vector of kids names with classes that make the decisions. Let's start with a Kids class which holds the data and loads the list:

```

public class Kids {
    protected Vector swimmers;

    //set up the vector
    public Kids(){
        swimmers = new Vector();
    }
    //add a kid to the list
    public void add(String line) {
        Swimmer sw = new Swimmer(line);
        swimmers.add(sw);
    }
}

```

```

//return the vector
public Vector getKidList() {
    return swimmers;
}
//get an enumeration of the kids
public Enumeration getKids() {
    return swimmers.elements();
}
}

```

This class creates a Vector of Swimmers and returns an enumeration of them as needed. The enumeration is returns is of the whole list of kids. However, we can derive classes from Kids that return enumerations of males or females by simply extending the getKids method. So we create a FemaleKids class just like the one above, except that the getKids methods returns only girls:

```

public class FemaleKids extends Kids {
    //set up vector
    public FemaleKids(Kids kds) {
        swimmers = kds.getKidList();
    }
    //return female only
    public Enumeration getKids() {
        Vector kds = new Vector();
        Enumeration enum = swimmers.elements();
        while(enum.hasMoreElements ()) {
            Swimmer sw = (Swimmer)enum.nextElement ();
            if(sw.isFemale ())
                kds.add (sw);
        }
        return kds.elements();
    }
}

```

This is the whole class: the remaining methods are in the base class. Similarly, we create a MaleKids class which differs only in the line

```

if(! sw.isFemale ())

```

The buttons themselves then instantiate an instance of the correct class, with each using only that class. This gets away from having to have a getAll, a getFemales and a getMales method, when they are really all the same. Here's the MaleButton class as we recast it to use the MaleKids class.

```

public class MaleButton extends JRadioButton implements
Command {

```

```

MaleKids kds;
JawtList kidList;

//constructor
public MaleButton(String title, Kids sw, JawtList klist) {
    super(title);
    kds = new MaleKids(sw);
    kidList = klist;
}
//The getKids method is the same in all three classes
public void Execute() {
    Enumeration enum = kds.getKids();
    kidList.clear();
    while(enum.hasMoreElements ()) {
        Swimmer sw = (Swimmer)enum.nextElement ();
        kidList.add (sw.getName ());
    }
}
}

```

Now, not only have we gotten rid of that awkward set of if statements in the `actionPerformed` routine, we've replaced three methods in one awkward class with a single method in 3 simpler classes. This is a much simpler and even more object-oriented approach. Having three button classes like this, each of which instantiates a different instance of the `Kids` class is an example of the `Factory Method` pattern, and having the 2 classes derived from the base `Kids` class is an example of the `Template` pattern.

## Mediating the Final Difference

But, we're still not done. Our 3 button classes all have to know about the `kidList` list box and add the names to it in the `Execute` method. This means that each button object has to know the details of the `kidList` object, and this is also poor design. It makes the program hard to change and maintain. Suppose we wanted to replace that list with a table or a tree list. We'd have to change 3 classes. Clearly that is a terrible idea, especially if the number of buttons grows.

But if the `Execute` method has to be in the button class, and the list has to be elsewhere, how do we resolve this? We resolve it by creating another class called a `Mediator`. This `Mediator` is the only class that knows about the details of the `kidList`. And all other classes only have to know about the `Mediator`. Here's the entire class:

```
//this mediator is used to get the enumeration
```



```
//and load the list
public class Mediator {
    JawsList kidList;

    //save the list in the constructor
    public Mediator(JawsList klist) {
        kidList = klist;
    }
    //load the list from the enumeration
    public void loadList(Enumeration enum) {
        kidList.clear();
        while(enum.hasMoreElements ()) {
            Swimmer sw = (Swimmer)enum.nextElement ();
            kidList.add (sw.getName ());
        }
    }
}
```

The way we use this, is that we create an instance of the kidList class and then create a Mediator:

```
kidList = new JawsList(20);
kidList.setBorder (BorderFactory.createBevelBorder
(BevelBorder.LOWERED));

loadSwimmers(); //read in file
med = new Mediator(kidList); //create Mediator
```

Then, we pass an instance of the Mediator to each button when we create it

```
female = new FemaleButton("Female", kds, med);
male = new MaleButton("Male", kds, med);
both = new BothButton("Both", kds, med);
```

and each Execute method just tells the Mediator what to do.

```
public class FemaleButton extends JRadioButton implements
Command {
    Kids kds;
    Mediator med;

    public FemaleButton(String title, Kids sw, Mediator md) {
        super(title);
        kds = sw;
        med = md;
    }
    public void Execute() {
```

```
        Enumeration enum = kds.getFemales ();  
        med.loadList (enum);  
    }  
}
```

## Summary

This brings us to the end of our design exercise. We've taken a pretty poor first program example and made it more object oriented and more extensible, and made it simpler as well. It also turns out that we used examples several well-known design patterns. Specifically, we used the Command pattern, the Mediator pattern and, for the JawtList class, the Adapter pattern. These are all patterns you'll find uses for many times over in your programming. For a longer treatment of these patterns, see the books on design patterns in the bibliography.



## 22. Building Web Pages

One of the most common reasons for writing Java programs is to enhance the look and functionality of world-wide web pages. For you to appreciate how a Java applet and the web page can interact, we'll take a few pages here to review the basics of web page construction. If you already know how to construct web pages, you can skip right on to Chapter 22 where we'll begin discussing how applets interact with web pages.

### HTML Documents

A world wide web page consists of formatted text, laid out in various ways along with images and, perhaps, Java applets. Throughout the page, there are underlined terms called *links*. When you click on these links, the browser loads the new page that this link refers to. These new pages can be additional pages on the same topic or they can be links to any page anywhere on the network: hence the name "world wide" web.

Every web page is a text file with formatting tags enclosed in angle brackets (<tag>) which describe how that text is to be drawn. These tags comprise the Hypertext Markup Language, or HTML. To indicate that text is to appear in a particular format, you enclose it between a start tag and an end tag. The start tag consists of one or more characters inside the angle brackets, and the end tag contains the same characters preceded by a forward slash. For example, to indicate the text which is to appear as a #1 head, you write

```
<h1>This is the header</h1>
```

Web browsers such as Netscape Navigator and Microsoft Internet Explorer read in the HTML text file, interpret these tags and format and display the text accordingly. Spaces and carriage returns are ignored in HTML. The formatting is controlled exclusively by the tags themselves. The tags may be in upper, lower or mixed case, since case is ignored by the browsers.

There are any number of tools for building non-interactive web pages: most of the popular word processors allow you to export any document into an HTML file. However, since tools for building interactive pages containing frames, forms and applets have not yet emerged, you need to

understand the rudiments of HTML syntax so that you can edit these files to produce the exact layout you have in mind.

While many people create their web pages from scratch as we do in this chapter, you can also start using templates from a number of sources. The Netscape web site *home.netscape.com* provides a dozen or so web page templates as well as a number of useful images. In addition, the complete specifications for all of the HTML tags is available at the Netscape site. There are also any number of programs for constructing web pages, such as Microsoft Front Page.

## Creating a Web Page With a Word Processor

Even if you plan to edit the web page later to contain more complex elements, you can often get started by typing the basic text into a word processor and saving it as an HTML file. For example, if you are using Microsoft Word, you can select File/New and select the HTML.dot template. Then you can type in the rudiments of your web page:

### Our Swim Team

The best in the state!

- We'd like you to join us
- See our address below.

---

Our address.

Then, if you save this text as an HTML file, you will find that the file is a text file, containing the following tags and text:

```
<HTML>
<HEAD>
<META NAME="GENERATOR" CONTENT="Internet Assistant">
<TITLE>Untitled</TITLE>
</HEAD>
<BODY>
<H1>Our Swim Team</H1>
<P>
The best in the state!
```

```

<UL>
<LI>We'd like you to join us.
<LI>See our address below.
</UL>
<HR>
<P>
Our address
</BODY>
</HTML>

```

Now let's see what these tags mean and how we can improve on them by editing this text.

## The Structure of an HTML Page

Every HTML page must start with the `<html>` tag and should end with the `</html>` tag. Within a page, there are two regions: the head and the body. The head section contains the title that appears in the browser window's title bar, and may include *meta information* about which tools were used to produce the page. Note that our word processor generated text above didn't fill in the `<title>` tag.

The body section consists of the text and all the tags which produce the visual layout, images and tags that insert Java applets. The following is a very simple HTML web page

```

<html>
<head> <title>A Basic Web Page</title></head>
<body>
<H1>Our Swim Team</h1>
A description of the team
<p>
Some details on joining the team.
</body>
</html>

```

This page is given in the file `basicpage.html` and is illustrated in Figure 21-1.



Figure 21-1: The Basicpage.html web page

## Headers

The HTML language allows you to select headers at six different levels. Realistically, no one uses more than 3 or 4, but you can see the effects of all 6 in the file headers.html. Note that we also use the `<hr>` tag to draw a horizontal line under the `<h1>` header.

```
<html>
<head> <title>Showing Headers</title></head>
<body>
<H1>Our Swim Team</h1>
<hr>
A description of the team
<h2>Joining the Team</h2>
Some details on joining the team.
<h3>Third level info</h3>
<h4>Fourth level info</h4>
<h5>Fifth level info</h5>
<h6>Sixth level info</h6>
</body>
</html>
```

These headers are displayed in Fig 21-2.



Figure 21-2: Illustration of the HTML header tags

## Line Breaks

There are two ways to start a new line on the screen:

```
<p>The paragraph tag, and
<br>the line break tag.
```

These differ in that a paragraph causes a line break and skips a blank line, while `<br>` simply starts a new line. Neither `<br>` nor `<p>` require an corresponding end tag `</br>` or `</p>` and these end tags are ignored by browsers.

A line break is also create automatically by using the `<center>` tag to center text that follows. If you want to center several lines, you need to specify breaks between them:

```
<center>
Our Team<br>
123 Aqua St. <br>
```



```
Anytown, IA<br>
</center>
```

## Lists in HTML

HTML supports two kinds of lists: ordered (or numbered) and unordered (or bullet) lists. Numbered lists automatically increment the item number for each successive element, and bullet lists are indented and prefixed with a bullet. You can nest and intermingle these to any level.

Ordered lists start with the `<ol>` tag and unordered lists with the `<ul>` tag. Each list line is introduced with an `<li>` tag, and may end with an `</li>` end tag. A list line is also terminated by a new `<li>` tag or by the end of the list `</ol>` or `</ul>`.

For example, the following is a numbered list with two levels of bullet lists as sublists:

```
<html>
<head> <title>List Elements</title></head>
<body>
<ol>
<li>Our team is the strongest in the state.
  <ul>
    <li>We've won 5 state championships
    <li>We sent a swimmer to the Olympics
      <ul>
        <li>Silver medal in Seoul in 1988
      </ul>
    <li>We all lift weights
  </ul>
<li>Our coaches are the handsomest in the state
<li>Our parents are the most fecund.
  <ul>
    <li>Most families have 3 or more kids on the team.
  </ul>
</ol>
</body>
</html>
```

The indenting in the HTML source above is for readability, it has no effect on the alignment of the displayed text. The resulting web page is shown in Fig 21-3.

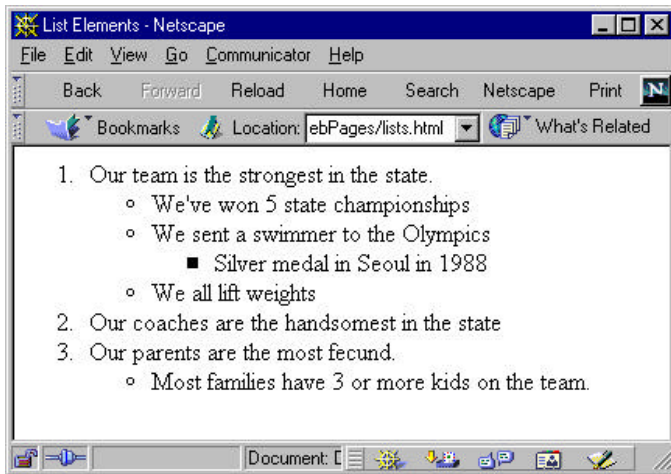


Figure 21-3: Illustration of nested ordered and unordered lists.

## Fonts and Colors

You can change the font size and emphasis using the bold, italic and underline tags

Begin `<b>boldface</b>` plain text `<i>italics</i>` and the following is `<u>underlined</u>`.

This produces the displayed text:

Begin **boldface** plain text *italics* and the following is underlined.

You can also nest the tags, creating bold plus italic, for example

`<b>Bold <i>bold and italic</i> only bold</b>` plain.  
Which displays on a web page as

***bold and italic*** only bold plain.

A font can appear in one of seven sizes, which you can specify by the tag `<Font size= n>` tag where *n* is a number between 1 and 7. You can also increase or decrease the font size by preceding the number with a plus or minus sign:

```
<Font size=-1>
```

To revert to the previous font size, simply terminate that font with the

```
</Font>
```

tag.

In the same fashion, you can change the color of a font with the `<Font color=name>` tag where the color can be one of the common color names red, green, orange, yellow, blue, magenta, pink, cyan, silver, gray, or (believe it or not) teal. You can also specify any color you can describe in terms of its *rgb* value by enclosing that hexadecimal number in quotes. For example, the statement

```
<Font color ="#8000ff">
```

produces a medium purple font color. Some browsers require that you precede the 6-digit hexadecimal value with the #-sign and others will work even if you omit it.

These colors are hard to predict if you aren't really experienced in color mixing, so some common ones are shown in Table 21-1.

<b>Table 21-1: Hexadecimal values for common colors .</b>	
000000	black
ff0000	red
00ff00	green
0000ff	blue

ffff00	yellow
40e0d0	turquoise
00ffff	cyan
ff0080	pink
ffd700	gold
ff8000	orange
60e000	mint green
add8e6	powder blue
e00080	purple mist
7fff00	chartreuse
228b22	racing green
a0522d	sienna
006400	dark green
ffa0c0	salmon
ff2020	radish
191970	midnight blue

Some of the colors are named by Joe Burns whose web site describes coloring your web page, at <http://www.htmlgoodies.com/colors.html>. Of course, if your computer display is set to 16 or 256 colors, you won't see the differences among some of the more subtle shades above.

## Background Colors

You can spruce up your web page by specifying the background color as part of the `<body>` tag. As with the font color tag, you can specify the color either by name or by hexadecimal color value. The statement

```
<body bgcolor = "#0000f0">
```

produces a medium blue background for your web page.

## Images

You can insert images in a web page to illustrate your point. These images can be icons, drawings, cartoons or even scanned photographs. Web page browsers support two standard image formats: GIF and JPEG. The GIF encoding scheme is more suitable for icons, cartoons and line drawings, while the JPEG scheme does a better job on photographs. For the same image, a JPEG file will be a little smaller than a GIF, but may take longer for your browser to decompress.

To insert an image, you simply refer to its filename inside the *img* tag:

```

```

and your browser will insert it at that point on your web page. For example, you might have a little “new” icon which you’d like to mark new parts of your web page with when you change it. You can do this by simply adding the image to that line:

```
<ul>  
  <li>How we did this year!  
  <li>We've won 5 state championships  
  <li>We sent a swimmer to the Olympics  
</ul>
```

The web page resulting from this image tag is shown in Fig 21-4 and is provided as an example file `img.html`.

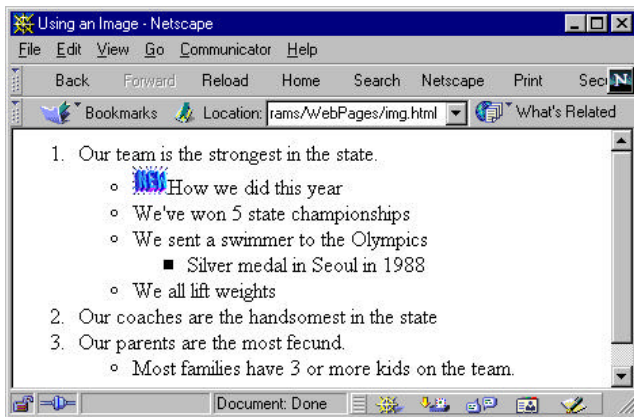


Figure 21-4: A list showing an image inserted in it,

## Where to get Images

There are huge numbers of images available on the World Wide Web. Search for "GIFs and Images." One popular source is IconBazaar. The web address changes frequently, so use a search engine to locate it.

You can also make a copy of any image you see in your browser on any page from the web, by right-clicking on the image. A menu will pop up that allows you to save the image to a disk file. Of course you should be careful not to re-use copyrighted material or trademarks.

## Putting Your Page on the Web

In order to give others access to your web pages, you must put them "on the web." This means that you must make some arrangement with an Internet provider who make space available on his servers for your web page(s). This may or may not be the same provider that you use for your Internet access. For example, you might be dialing in to a provider such as Netcom or Worldnet to get your mail and access to the Internet, but using a different provider to host your web pages.

The web server provider will give you the filename and internet address where you can put your web page. Typically, your home page is named

default.html and if the provider is named “turkeynet.com” then your web page address might be something like

```
http://www.turkeyweb.com/ourteam/
```

You can transfer the page to that address by signing on to your internet provider and accessing the address your provider gave you using the ftp command or the ftp feature of your browser. For example in Netscape Navigator, if you type in the address:

```
ftp://userid.password@ftp.turkeynet.com/ourteam
```

you can use the File/Upload menu item to transfer your web page to the webpage server..

## Hyper Links

So far, our little prototype web page has been pretty boring. It’s just a laundry list of little facts about “our team.” These pages become more interesting if you provide links to related information. And now that we know how web page addresses are specified, we can include some. To continue with our team example, let’s assume it might be advantageous to include a link to the national swimming organization’s web page as well. To do this, we enclose the text which is to make up the link in an *anchor* tag, specifying the link as part of the beginning tag:

```
<li>Our team is a member of  
<a href="http://www.usswim.org">  
United States Swimming</a>
```

This produces a link, usually indicated by blue underlining that allows you to jump to the US Swimming home page by just clicking on it.

## Links to More Pages of Your Own

In the same fashion, you could write more pages describing facets of your team and link them to your home web page. Links to these local pages do not need to contain the complete internet address, only the name of the file itself. This makes it easy to test the pages on your own computer using a browser and then move all of the pages to the web server when they are completed.

For example, you might want to make another page called `champs.html` listing your team's accomplishments:

```
<html>
<head>
<title>State Championships</title>
</head>
<body bgcolor="00f0ff">
<h1>State Championships</h1>
<ul>
<li>1989 Girls Senior Champions
<li>1992 13-14 and 15-18 Boys Age Group Champion
<li>1994 Overall Age Group Girls Champion
<li>1995 Boys Senior Champions
</ul>
</body>
</html>
```

You can add this link to the line mentioning championships on the home page by simply referring to the file name: no web address is needed, since you'll be uploading all of them to the same directory:

```
<li>We've won <a href="champs.html">
    5 state championships</a>
```

## Links Within the Same Page

When your web page is more than 2 or 3 screens long, it is not uncommon to insert a link from some spot near the top of the page to an area later in the page. The destination spot is tagged using an anchor tag with a *name* attribute rather than an *href* attribute.

```
<a name="address"><h2>Our address</h2></a>
```

The hyperlink reference within the same page must start with a #-sign:

```
<li>Our <a href="#address">team</a> is the
strongest in the state.
```

In the same fashion, if you want to refer to a particular spot in one page from another page, you include a number-sign (#) and the name tag in the hyperlink reference:



```
<a href="ourteam.html#address">See our address</a>
```

## Mailto Links

You might want to include your team's Email address on your home page. You can make this a link to your actual address using the special *mailto* form of the anchor tag:

```
<a href="mailto:ourteam@turkeynet.com">Our Team</a><br>
```

When someone reading your page clicks on a mailto link, it will bring up the Email section of your browser and allow you to send Email to that address

The complete web page is given on your example disk as ourteam.html and is shown in Fig 21-6.

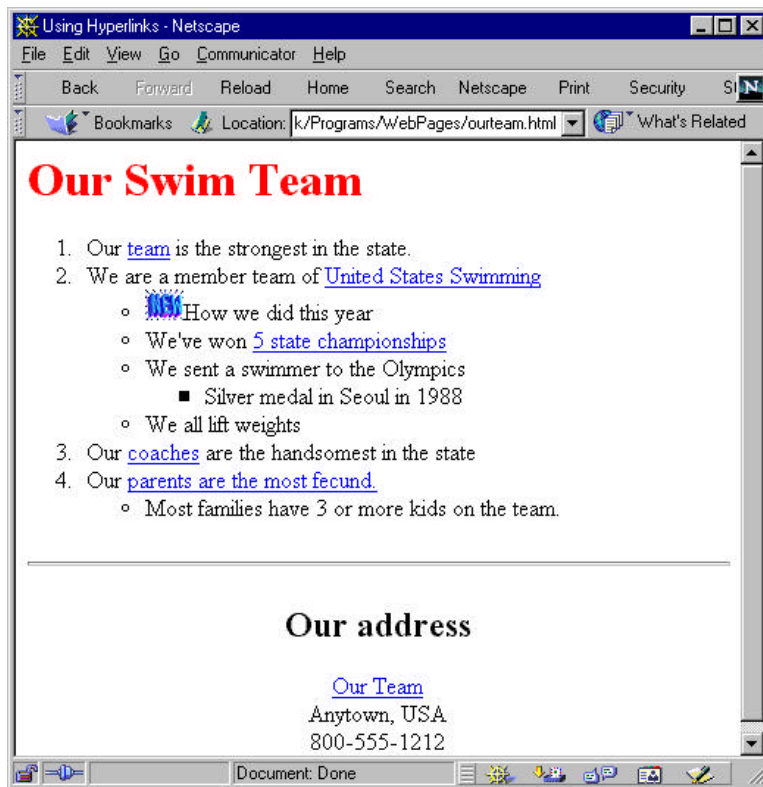


Figure 21-6: The completed ourteam.html home page.

## Tables

Tables in HTML provide a convenient way to group text on a page so that the layout stays as you intended no matter what size the page is displayed.

Tables are most easily generated using a word processor and saving the file as an HTML document. You can group HTML tags and text in table cells, and these cells can either be outlined or hidden. The markup for tables is somewhat elaborate and we outline it in Table 21-2..

<b>Table 21-2: Table tags</b>	
<code>&lt;table&gt; &lt;/table&gt;</code>	The entire table is surrounded by the table tags. This one has an invisible outline.
<code>&lt;table border=1&gt; &lt;/table&gt;</code>	This table has a border and outline for each cell which is one unit wide. Wider borders are possible but seldom used.
<code>&lt;td&gt;&lt;/td&gt;</code>	The text in each cell of the table is enclosed in the Table Data tags.
<code>&lt;tr&gt;&lt;/tr&gt;</code>	Each row is separated by the Table Row tag. The end tag is seldom used.
<code>&lt;caption&gt; &lt;/caption&gt;</code>	These tags surround a table caption which appears just above the table.
<code>&lt;th&gt;&lt;/th&gt;</code>	Highlighted table cell.
<b>Attributes</b>	
valign	inside a <code>&lt;tr&gt;</code> , <code>&lt;td&gt;</code> or <code>&lt;th&gt;</code> cell, it specifies how the text is aligned. The values can be <i>top</i> , <i>middle</i> , <i>bottom</i> , or <i>baseline</i> . The syntax is <code>&lt;td valign=middle&gt;</code>

colspan	specifies the number of columns a cell can span. The default, of course, is colspan=1.
Rowspan	specifies the number of rows a cell can span. The default, of course, is rowspan=1.

A simple table is shown in the file table.html, which is given below:

```
<HTML>
<HEAD>
<META Name="Generator" Content="Lotus Word Pro">
<TITLE>Document Title</TITLE>
</HEAD>
<BODY>
<TABLE border=1 >
<caption>Our team is summarized below </caption >
<TD valign=top>Four coaches
<TD valign=top>Five squads
<TR>
<TD valign=top>An active<br>
Parents club<br>
who help run swim meets.
<TD valign=top><i>and a partridge in a pear tree</i>
</TABLE>
</BODY>
</HTML>
```

This table is shown in Figure 21-7.

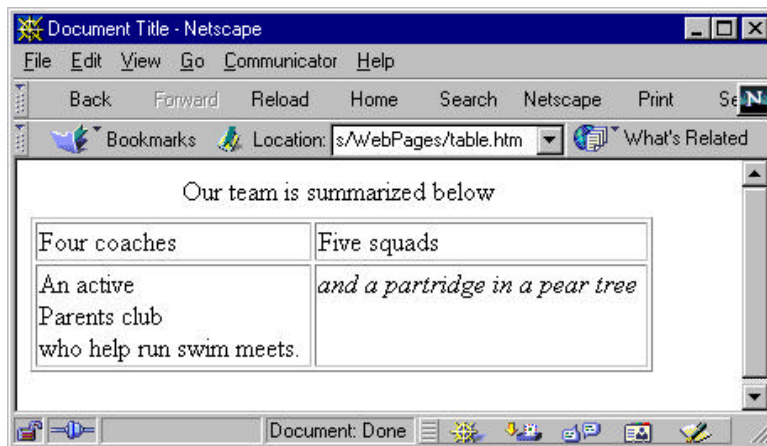


Figure 21-7: A table created using the tags in the table.html file.

## Frames

Frames provide a mechanism to divide your browser page into sections where each section is displaying a separate web page. Typically, we use frames to keep a fixed index along the side or bottom and change the main pane as the user clicks on the index frame.

A page consisting of frames is defined in a frameset document. Frameset documents have a <header> section, but have a FRAMESET section instead of a body section. Such documents define the layout of the frames on the page and give the names of the html files that will initially occupy them.

The Frameset tag has two possible attributes: ROWS and COLS. You can specify the size of the frames in pixels or in percentages:

```
<frameset cols=20%,80%>
<frameset cols = 50, *, 100>
```

In the first case, there are two frames, one will occupy 20% of the horizontal space and the other 80%. In the second case, the frames are 50 pixels wide, 100 pixels wide, and whatever is left.

The Frame tag then defines the attributes of each of the frames in the order the frames are defined. The *src* attribute tells which file is first to be displayed in that frame and the *name* attribute names that frame so that links can specify which frame the referred to document is to be displayed in. This is the contents of the file fset.html, a typical frameset document.

```
<html>
<head><title>Framed Document</title></head>
<frameset cols="30%,*" >
<frame src="index.html" name="index">
<frame src="ourteam.html" name="main">
</frameset>
</html>
```

The file index.html illustrates how this name attribute is used.

```
<html>
```

```

<body>
<a href="ourteam.html" target="main">Our team</a> <br>
<a href="champs.html" target="main">
Our winning record</a><br>
<a href="coach.html" target="main">Our coach</a>
</body>
</html>

```

The *target* attribute of the *href* tag in this index file says that when that link is selected, the contents of the “main” frame are replaced with the new document. The index frame remains unchanged. This is illustrated in Figure 21-8.

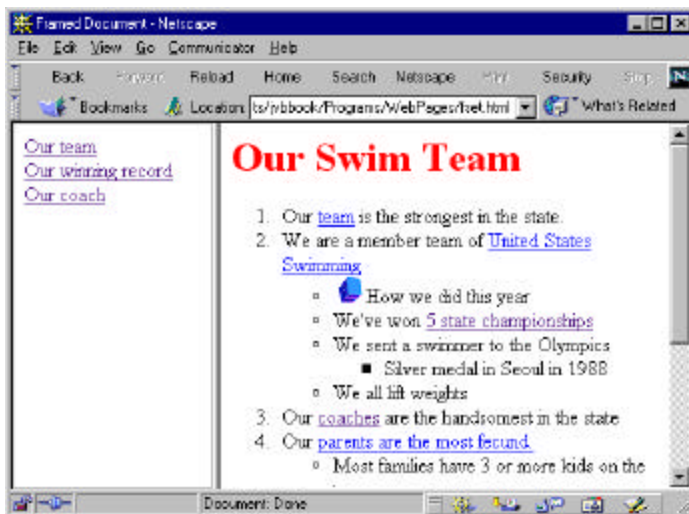


Figure 21-8: The display of frames produced by fset.html.

## Summary

In this chapter, we’ve reviewed how HTML web pages are constructed. We’ve looked at the header, list and font control tags and how hypertext links are constructed. We also covered how tables and frames are constructed. Now we’ve laid the groundwork for how web pages work, we’ll get back to Java and see how applets are integrated into more interesting web pages.



## 23. What is JavaScript?

Many people who are new to web page programming confuse the Java language with the JavaScript language. They are actually the result of two separate development paths at two different companies: Sun and Netscape. JavaScript was originally named LiveScript and was designed to give a little more interactivity to web pages. After Java became so popular, the language was renamed JavaScript and some attempts were made to give it a slightly object oriented flavor and to allow it to communicate with Java applets. In fact, they are two different languages, with different purposes and widely different styles.

### **Differences Between Java and JavaScript**

JavaScript exists only within web pages. It is not compiled: its statements are interpreted when the page is loaded, and you cannot write stand alone JavaScript programs. Since it is not compiled, only a cursory syntax check is performed when the web page is loaded by the browser, and errors can still occur when the program is executed which would be caught in compiled languages. Thus, you need to test JavaScript programs very carefully before making them available on your web pages.

JavaScript is primarily for computing strings and values: it does not itself have any GUI components, although you can refer to the basic Form controls within JavaScript. JavaScript has a close relationship to the browser and the web page. You can determine the history of pages the user has visited and change links accordingly and can change such things as background color dynamically.

While JavaScript has a C or Java-like syntax, using braces to set off blocks of code, it doesn't really allow you to create objects or to derive new objects from existing ones. The main purpose of JavaScript is to allow you to perform some computations affecting the appearance of your web page, and to validate input data before sending it off to a CGI server. Since JavaScript does allow you to check for the consistency of your input data, we will spend the rest of this brief chapter explaining what you can do with this useful little language.

## Embedding JavaScript Programs

A JavaScript program is a part of an HTML document, just like a frame or a table. It starts with the `<script>` tag and ends with the `</script>` tag.

```
<html>
<body>
<Script>
  //JavaScript statements go here
</script>
</body>
</html>
```

You can also optionally specify the language which you are embedding. This will become more important as other web languages appear:

```
<script language="Javascript">
```

JavaScript programs can appear anywhere on a web page, but if you write functions which are called from within those programs, the functions must appear in the `<head>` section of the web page. This guarantees that the functions will have been loaded by the time the JavaScript main program is detected and interpreted later on the web page.

```
<html>
<head>
<script>
  function square(x)
  {
    return x*x;
  }
</script>
</head>
```

You can also embed a JavaScript program from a separate file using the same “src=” tag we used to load images:

```
<script src="myscript.js">
```

For this feature to work properly, your web server must map the “.js” extension to the type “application/x-javascript” and send this type information back in the HTML header.



## JavaScript Variables

Variables in JavaScript are not typed and do not need to be declared in advance. You can just use them as you need them:

```
x = 5;
y = x/3;
z = "Fred";
```

However, illegal operations such as

```
a = z/2;           //where z="Fred"
```

are not discovered until you execute the program. All of the operators that you can use in Java, such as “++”, “+=”, and “>>” are also legal in JavaScript.

Variables are created where you first declare them and are all global: they hold the same values even between separate functions. You can create a local variable, whose scope is limited to the current function by preceding it with the *var* declaration:

```
var x = 5;
```

Let’s consider the following simple JavaScript program:

```
<html>
<body>
<Script>
for (i=0; i<10; i++)
  document.writeln(i + " " + i/3 + "<br>");
</script>
</body>
</html>
```

This simple web document is shown in Figure 24-1 and is on your example disk as math1.htm.

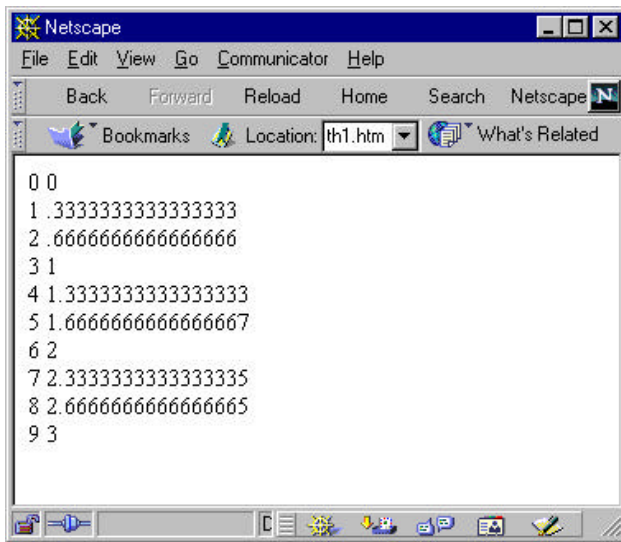


Figure 24-1: The numbers displayed by the math1.htm JavaScript program.

Note that the *for* loop has exactly the same syntax as it would in Java. It is at first tempting to equate the Java statement

```
System.out.println(i + " " + i/3);
```

with the JavaScript statement

```
document.writeln(i + " " + i/3 + "<br>");
```

However, the Java statement writes characters to the standard output channel, usually a DOS or terminal window, while the JavaScript statement writes HTML code to a web page. Thus, you are not just writing text to a screen, but are writing HTML code that will be displayed according to the rules of web pages. For example, if we just wrote

```
document.writeln(i + " " + i/3);
```

without including the `<br>` tag, all of the numbers would be run together on a single line. Another way to force separate lines on the screen is to enclose

the text within `<pre>` (presentation) tags, as is done in the example code `math2.htm`.

```
<Script>
document.writeln("<pre>");
for (i=0; i<10; i++)
  document.writeln(i + " " + i/3 );
document.writeln("</pre>");
</script>
```

This also forces the output to be in a Courier or typewriter-style font as shown in Figure 24-2.

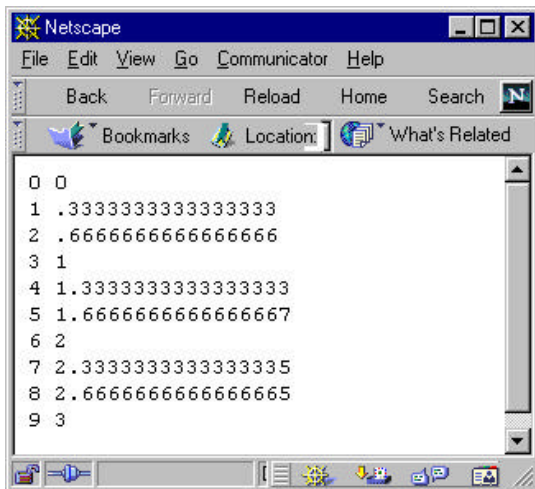


Figure 24-2: The numbers displayed by the `math2.htm` JavaScript program.

So to summarize, writing output to a web page is actually writing HTML markup to a web page and you can affect the font size, layout, color, and style just as you would if you created a static web page. In fact, it is this construction of dynamic web pages that is one of JavaScript's purposes.

## The JavaScript Objects on the Web Page

While JavaScript is not really an object-oriented language, it is sometimes called “object based” because a few of the parts of a web page can be treated as simple objects. The overall object structure of the web page consists of:

Window	The entire screen, including all frames
Document	One entire frame
Forms	One of the forms within the document
Elements	The visual controls within the form
Links	An array of the links within the document
Anchors	An array of all the <i>name=</i> tags within the document
Location	The current URL
History	An object containing the browser’s history list.

Each of these objects has a few methods and properties that you can use in your JavaScript programs.

## The Window Object

The Window object allows you to control the screen and pop up new windows. Most frequently, you use the *alert()* function, which is in fact a method of the window object. It pops up a window that says “JavaScript alert:” followed by whatever message you provide:

```
alert("You didn't enter a name");
```

The other common properties are shown in Table 24-1. You do not have to indicate the name of the window explicitly, it is always understood.

Table 24-1: Methods of the Window Object	
defaultStatus(text)	The text to appear in the status bar when not changed by an onMouseOver event for a particular

	control.
status(text)	The text that appears in the status bar.
frames()	An array of the frames that make up the window. The overall frame is the 0 <sup>th</sup> element of the array.
alert(text)	pops up an alert window
window.open(URL, "name", "features");	<p>opens a new window. The features, appearing as one long string in quotes may be</p> <pre> toolbar=yes,no status=yes,no menubar=yes,no scrollbars=yes,no resizeable=yes,no width=pixels height=pixels </pre>
close()	closes the current window. You should not close the main window.
confirm(text)	displays a message box followed by yes or now. Returns true or false.
prompt(text)	Displays a message and a text entry field. Returns a string.
SetTimeout("function", msec)	sets a time in milliseconds after which a function is called repeatedly. This allows effects such as scrolling.

## The History Object

The history object contains a reference to the browser's URL history list. You can perform the following three simple operations using the history object:

```
history.back();    //previous URL
history.next();    //forward to next URL
history.go(n);     //go to URL #n in list
```

## The Document Object

The most useful method of the document object is, of course, `document.write(text)` as we have already seen. The common use for this is to print out the last modified date at the bottom of a web page, so that it is always correct:

```
document.write("This page last modified: ");
document.writeln(document.lastmodified());
```

The complete list of document properties is shown in Table 24-2:

<code>alinkColor</code>	The color of an active link: the color it changes to when selected.
<code>linkColor</code>	The color of an unvisited link.
<code>vlinkColor</code>	The color of an already visited link.
<code>bgColor</code>	The color of the page's background.
<code>fgColor</code>	The foreground text color
<code>title</code>	The title in the title bar. You can only set this once. Once it is displayed, it cannot be changed.
<code>location</code>	The current URL

lastModified	The date the file was last modified.
--------------	--------------------------------------

## Using Forms in JavaScript

You can access each of the elements in a <Form> section of a web page in JavaScript if you give them names. If you give the form a name, you can access the state of the controls and use them to perform simple calculations. You also can respond to mouse and click events associated with each control, and use them to change the form or check the validity the data that the user entered.

The only significant method for the Form object is the *submit()* method, which causes it to be sent to the server.

## The Button Control

The button control generates a *click* event which you can respond to by declaring an *onClick* event handler as part of the button declaration on the form:

```
<input type="button" name="Compute"
      onClick="CalcTemp(this.form)">
```

The name of the function must be in quotes and since you will probably need to pass the name of a form to the function, we illustrate doing that here. The expression *this.form* refers to this form in the current document (*this*).

When the button is clicked, it calls the function *CalcTemp()*. You must declare this function in the <head> portion of the web page so that it is sure to be loaded when the button click calls it.

## A Simple Temperature Conversion Program

Let's consider the simple temperature conversion program again and see how we could write it in JavaScript. We will create a Form consisting of an

input text field, an output text field, two radio buttons and a Compute button.

```
<form name="tempcalc">
Enter temperature:<br>
<input type="text" name="Entryval" size=10><br>
Result:<br>
<input type="text" name="Results" size=10><br>
<input type="radio" name="Tempchoice">to Fahrenheit<br>
<input type="radio" name="Tempchoice">to Celsius<br>
<input type="button" value="Compute"
onClick="convert(this.form)">
</form>
```

The radio buttons must have the same *name* properties if you wish them to work together. You can determine which of them has been checked because the constitute and array of 2 elements. The first button declared is referred to as *Tempchoice[0]* and the second as *Tempchoice[1]*. In either case we refer to its *status* property which will be either true or false. The *convert* function then is

```
<html>

<head><title>Temperature Conversion</title></head>
<script>
function convert(form)
{
//This function is called when the
//Compute button is clicked
//-----
//read in entered value
temp = parseFloat(form.Entryval.value)
if (form.Tempchoice[0].status)
//check radio buttons
newtemp = temp*9/5+32; //to Fahrenheit
else
newtemp = (temp-32)/9*5; //to Celsius
form.Results.value = newtemp //display result
}
</script>
</head>
```

This program is in the example file *temper.htm* and is displayed in Figure 24-3.



Figure 24-3: The display of the `temper.htm` JavaScript program.

In this simple program, we examine the `status` property of the checkbox and then set the text value of the output text box called `Results`. Since the `convert` function could be called from more than one form, we pass the form object to the function.

## Properties of Form Controls in JavaScript

The major properties of the controls are listed in Table 24-3.

Button	value	the text on the button
checkbox	checked	true if checked
password	value	text of password-style text box
text	value	text of text box
radio	checked	true if selected
	length	number of radio buttons in group
select	length	number of lines in list box
	selectedIndex	index of selected line
	selected	indicates whether current line is selected
	options()	array of text lines in list box
textarea	value	all the text in the multiline text box

In addition, each of the controls has a number of methods associated with it and equivalent events as shown in Table 24-4. The methods are functions you can call to manipulate the controls. The events can be specified in the control definition, indicating what function is to be called.

<b>method</b>	<b>event</b>	
focus()	onFocus	set focus to this control
blur()	onBlur	move focus from this control
select()	onSelect	highlight text in text control
	onChange	called if user changes control
click()	onClick	clicks control but does not call onClick
	onSubmit	occurs when a submit button is clicked. Return true to go on with submit, false to abort.

## Validating User Input in JavaScript

If you create a web page with an HTML form in it, you might find it advantageous to check the validity of the data you enter before posting the form to the server. This is clearly faster than waiting for a server-side CGI script to return the same information. In the case of the dinner order form we wrote in Chapter 23, we ought to check that

1. The user entered his name.
2. He selected a main course.
3. He selected a type of wine.

4. He selected a soup, a salad or both.

We can accomplish this validation by either

- intercepting on onSubmit event and returning true or false, or
- changing the submit button to an ordinary button, executing an onClick function and calling the *submit()* method for the form only if the data is valid.

We choose the second method here. The form is defined as:

```
<form method="post" action="/cgi32/java.exe?cgiserver"
name="dinner">
<!-- ----Group the order using a table---- -->
<TABLE>
<TR><TD WIDTH=295>
Name: <input type="text" width=25 name="Name"><br>
</TD>
<TD WIDTH=295>
<input type="radio" name="wine" value="redwine" onClick=0>Red
wine<br>
<input type="radio" name="wine" value="whitewine"
onClick=0>White wine<br>
</TD>
</TR>
<TR>
<TD WIDTH=295 align=center>
<select name="meats" size=5>
<option>Roast chicken
<option>Steak bearnaise
<option>Duck ala orange
<option>Dull pasta salad
</select>
</TD>
<TD WIDTH=295>
<input type="checkbox" name="soup">Soup<br>
<input type="checkbox" name="salad">Salad<br>
</TD>
</TR>
</TABLE>
<P>
<!-- Put submit centered outside table -->
<center>
<input type="button" value="Submit order" name="Submit_order"
onClick="checkForm(this.form)">
<input type="reset" name="Clear entries">
```

```
</center>
</form>
```

When the “Submit order” button is clicked, the button calls the *checkForm()* method which is shown below:

```
<script language="JAVASCRIPT">
function checkForm(form) {
var ok_to_submit = true;           //flag to allow submit
var error="";                     //init error messages
if(form.Name.value == "") {
    error+="No name was entered\n";
    ok_to_submit=false;
}
if(form.meats.selectedIndex<0) {
    error+="No main course selected\n";
    ok_to_submit=false;
}
if(!(form.wine[1].checked || form.wine[0].checked)) {
    error+="You didn't select any wine\n";
    ok_to_submit=false;
}
if (!(form.soup.checked || form.salad.checked)) {
    error+="Pick a soup, salad or both";
    ok_to_submit=false;
}
if (ok_to_submit)
    form.submit();                //submit the form
else
    alert(error);                 //or display error msg
}
</script>
```

## Summary

As you can see, JavaScript can provide a quick and easy way to add logic to simple web pages. Its primary use is for data validation before submitting a form to a CGI server, but you can also use the `document.lastupmodified()` function to keep the message on the bottom of your web page up to date. JavaScript has some other annoying uses, such as putting rotating banners in the status bar at the bottom of your web page, but we'll leave programming them to you. In the next chapter, we'll look at how we can establish more robust client-server communication in Java directly.

## 24. Interacting with Web Page Forms

The basic HTML language provides some simple methods for adding some interactive controls to web pages inside *<Form>* tags. You can put text fields, check boxes, radio buttons, list boxes and drop-down list boxes on the form for user to fill in. This approach is frequently used in today's web pages, to gather data, order merchandise and add yourself to mailing lists. The disadvantage of this web form approach is that the only way to process the filled out HTML is to send it to the server, where a program you have written there processes the data. In this chapter, we'll look at the basic syntax of HTML forms and then discuss how you can use Java to process them on the server.

### The HTML Form

All of the interactive elements on an HTML must be enclosed inside the *<Form>* tag. For example, the following simple form puts two text fields on the screen:

```
<html>
<head><title>Forms Example</title></head>
<body>
<form action="/cgi/getnames" method="post">
Please enter your name:<br>
First name:
<input type="text" size=20 maxlength=25 name="f_name">
Last name:
<input type="text" size=20 maxlength=25 name="l_name">
</form>
</body>
</html>
```

This is illustrated in Fig 23-1 is can be found on your example disk as `form1.html`.

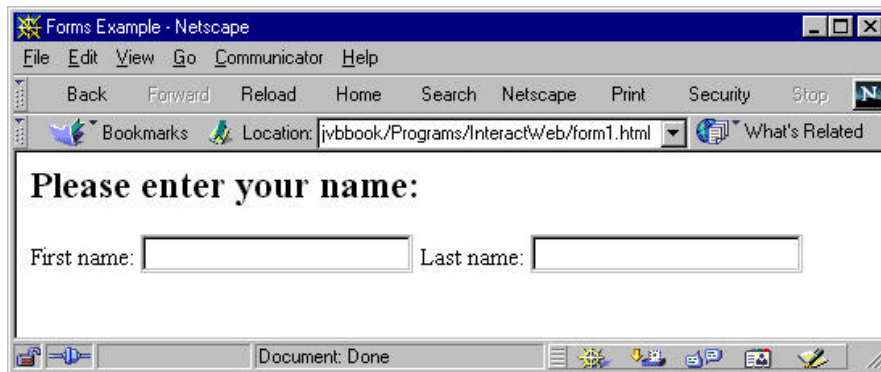


Figure 23-1: Two simple text entry fields on an HTML form.

## Sending the Data to the Server

You can load this little web page into your browser and it will display the fields you see in the figure. However, they aren't very useful unless some program somewhere can act on them. This is what is specified in the *Form* tag. The keyword phrase

```
action="/cgi/getnames"
```

indicates that a program called `getnames` in the `/cgi` directory is to be executed. We use Unix-like forward slashes to separate the directory path even if the server is a PC. The other keywords

```
method="post"
```

tells the Form how to send the data. This is far and away the most common of two possible transmission methods. The other possible tag value is

```
method="get"
```

but you will probably never need to use it.

OK, well enough, but how is the data sent? All of the data is sent when you click on the Submit button. This button may have any sort of label, but has the type "submit."

```
<input type="submit" value="Send Data">
```

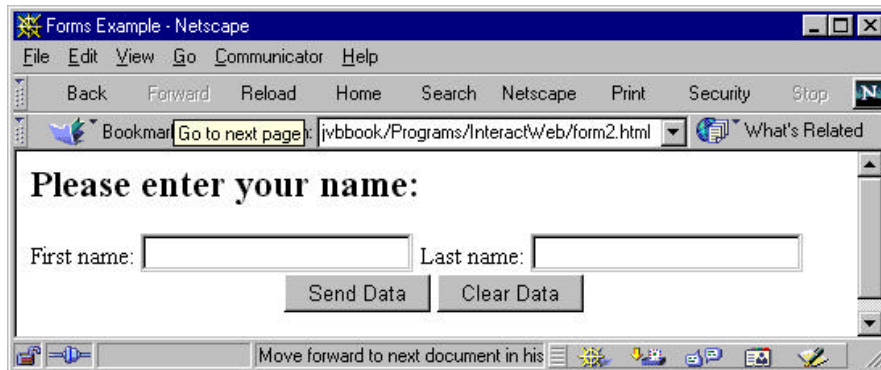
You can also include a Clear button which clears all of the entry fields:

```
<input type="reset" value="Clear Data">
```

This packages up all the fields and sends them to the server, where the program `/cgi/getnames` is executed and fed that data. We'll look into the format of that data and what the server program does with it shortly.

This example includes the two buttons, and it is illustrated in Fig 23-2, and is `form2.html` on your example disk.

```
<html>
<head><title>Forms Example</title></head>
<body>
<form action="/cgi/getnames" method="post">
<h2>Please enter your name:</H2>
First name:
<input type="text" size=20 maxlength=25 name="f_name">
Last name:
<input type="text" size=20 maxlength=25 name="l_name">
<center>
<input type="submit" value="Send Data">
<input type="reset" value="Clear Data">
</center>
</form>
</body>
```



```
</html>
```

Figure 23-2: A simple HTML form with a submit and clear button.

## Form Fields

There are only six possible controls that you can place on a form, and they are summarized in Table 23-1.

Input type=	Value=	Description
Text	any text	A text entry field. You can specify size= to specify the width of the field.
Textarea	any text	A multi line text entry field. You can specify rows= and cols= to define its size.
Checkbox	yes no	A single check box. The label is just text outside the checkbox input type.
Radio	any  CHECKED	A radio button. The value may be any text, but you may include the CHECKED keyword on one of the buttons. The label for the button is just text outside the radio input type. The <i>names</i> of all the grouped radio buttons must be the same. To distinguish them, give them different <i>value</i> properties.
Submit	any text	The button that sends the form contents to the server
Reset	any text	The button that clears all the entry fields

## Selection Lists

In addition to the simple inputs we show above, you can also include regular and drop down list boxes, which are called “Selection Lists.” Since these lists can be of any length, they are bracketed with the `<select>` and `</select>` tags.

```
<Select name="fruits" size=5>
<option>Apples
<option>Avocados
<option>Cherries
<option>Pears
<option>Pomegranates
```



```
</select>
```

The *size* keyword indicates the number of lines to be displayed. There can be many more entries than lines.

You can also specify that one line is to be selected by default by added the *Selected* keyword:

```
<option Selected>Pomegranates
```

If you want more than one listbox element to be selected at once you can mark this list a multi-select by including the *multiple* tag.

```
<select name="fruits" Multiple size=5>
```

Unfortunately, multi-select lists are not very easy to recognize or use. If you just click on a line, it will be selected by itself. To select more than one you must

1. Drag the mouse over the lines to be selected,
2. Hold down Shift and select contiguous lines, or
3. Hold down Control and select non-contiguous lines.

You should probably use a series of check boxes instead if you want to allow multiple selection.

## Drop-down Lists

The drop-down list is just a special case of the Selection list box, where you set “size=1” or omit *size* altogether. In either case a single line list is displayed which drops down to reveal the list selections. While these dropdown lists can also be multiply selected, they aren’t very easy to recognize or use, and a series of checkboxes is again easier for your user to understand.

## Making an Order Form

So now, let's use this form technology to order our evening meal. The simple HTML shown below will produce the order form shown in Fig. 23-3. This is the file order.html on your example disk.

```
<html>
<body>
<center>
<h1>Order your meal here</h1>
</center>
<form method="post" action="cgi/order.exe">

Name: <input type="text" width=25><br>
<input type="radio" name="wine">Red wine<br>
<input type="radio" name="wine">White wine<br>
<select name="meats" size=5>
<option>Roast chicken
<option>Steak bearnaise
<option>Duck ala orange
<option>Dull pasta salad
</select>
<p>
<input type="checkbox" name="soup">Soup<br>
<input type="checkbox" name="salad">Salad<br>
<p>
<center>
<input type="submit" name="Submit order">
<input type="reset" name="Clear entries">
</center>
</form>
</body>
</html>
```

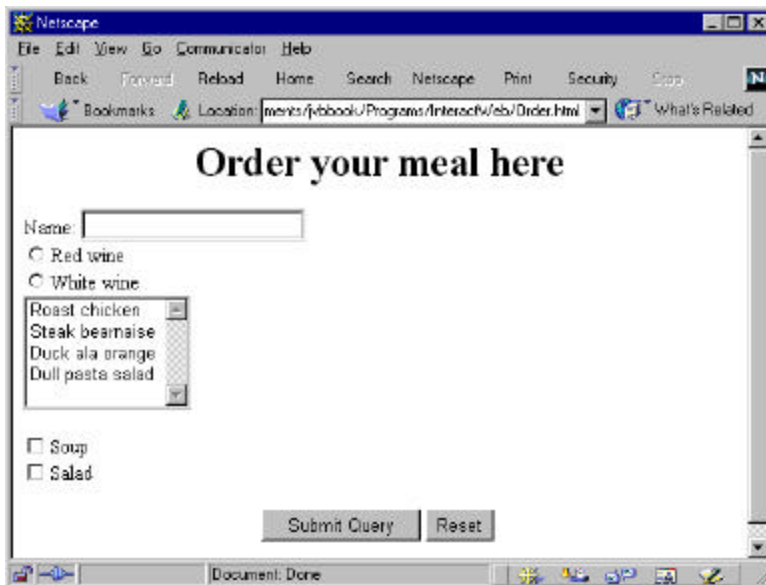


Figure 23-3: A simple HTML form for ordering your dinner.

Now let's improve on this slightly by grouping these controls using a table. We'll put the name in one cell, the wine order in another, the meat order in a third and the soup and salad in a fourth. We've also improved the contrast by giving the page a light gray background. The HTML which is given in the file `ordert.html` looks like this:

```
<HTML>
<HEAD>
<TITLE>Tabular meal order</TITLE>
</HEAD>
<BODY BGCOLOR="lightgray">
<h1>Order your meal here</h1>
</center>
<form method="post" action="cgi/order.exe">

<!-- ----Group the order using a table---- -->
<TABLE>
<TR><TD WIDTH=295>
Name: <input type="text" width=25 name="Name"><br>
</TD>
<TD WIDTH=295>
<input type="radio" name="wine" value="redwine">
  Red wine<br>
```

```

<input type="radio" name="wine" value="whitewine">
  White wine<br>
</TD>
</TR>
<TR>
<TD WIDTH=295 align=center>
<select name="meats" size=5>
<option>Roast chicken
<option>Steak bearnaise
<option>Duck ala orange
<option>Dull pasta salad
</select>
</TD>
<TD WIDTH=295>
<input type="checkbox" name="soup">Soup<br>
<input type="checkbox" name="salad">Salad<br>
</TD>
</TR>
</TABLE>
<P>

<!-- Put submit centered outside table -->
<center>
<input type="submit" name="Submit order">
<input type="reset" name="Clear entries">
</center>
</form>

</BODY>
</HTML>

```

Note the use of the HTML comment, starting with “<!--” and ending with “-->”. The resulting order page is shown in Fig. 22-4.

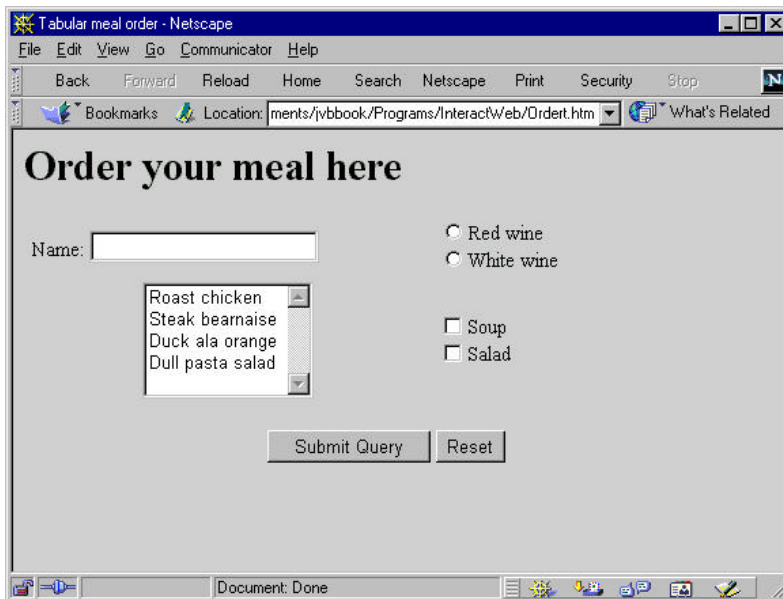


Figure 22-4: The HTML order form, grouped using table tags.

## Submitting Your Form

Now that we've clicked on the Submit button we need to understand what happens next. The submit operation sends a command string to the HTML server in a format called a CGI or Common Gateway Interface script. The data in this script launches the program you specify as part of the action tag, and feeds the remainder of the script to its standard input. Standard input is the same path as if you were typing at the keyboard. The program then digests that data and usually constructs a new HTML page and sends it out its standard out channel. This output is then ingested by the web server and sent back to your web browser.

The format of the input to the program executed on the server is a single long line of text, with arguments separated by the ampersand (&) sign and spaces replaced by the +sign. In addition, special characters such as quotes, apostrophes, ampersands and +signs are replaced by a %-sign followed by their hexadecimal equivalent. Thus, if the diner's name as Fred O'Farkle, the CGI string would begin as:

```
name="Fred+0%27Farkle"&wine="redwine"&
```

As you can see, each control on the HTML form must have a name. This name is sent followed by a string containing its value. A field which has not been filled in will have a zero-length string sent:

```
name=""
```

We'll discover that both Java servlets and JavaServer pages can parse this stream completely invisibly to you as the programmer.

## Summary

In this chapter, we've learned about forms on web pages and how they send data to the server, and how to parse and respond to that data. We briefly mentioned error checking as part of your server program, and in the next chapter, we'll look at how to check these errors using JavaScript, and then in the following chapter, how to use Java Servlets and JavaServer pages for the server end.



## 25. Using Servlets

In the previous chapter we discussed creating HTML forms that sent data to a server using the CGI interface. We avoided writing any programs for the server end there, because we can do a lot better using Java servlets instead of executable C or perl programs.

Let's suppose we have a typical application for an HTML form page, where we want to look up a list of people in an organization, or find out someone's phone number or address. We type the name into a field, check off some buttons restricting the search and click on the "submit" button. As you probably know, this sends a data stream back to the server in the form of a CGI (Common Gateway Interface) request. Usually, that request includes a stream of these selectable parameters from the web page, and the name of an executable program that the server is to run. That program then will parse the rest of the stream, act on it, look up the answer in a table or database, and generate an HTML output page to send back to the client.

As usual, the devil is in the details. Parsing that convoluted stream is a bit of work, although there are lots of C libraries and perl scripts that can do it. There is also a package of Java routines that can handle these CGI data streams very efficiently. These are available as a set of jar files in the Java Servlet Development Kit (JSDK). A Java servlet is a small Java program that can extend the HTML request and response service and parse that CGI stream using convenient high-level classes and methods. Then it can call any other Java methods or use any other available data to produce an output HTML page.

When you write a CGI-script in other languages, you point the browser to the actual program that will parse that form's data stream:

<http://someserver/cgi-bin/getData.pl>

or

<http://someserver/cgi-bin/getData.exe>

and the web server knows that it is to execute these programs. Since Java programs are executed through a Java interpreter program, you can't just point to a class file and have it executed. Instead, most web servers can be configured to launch Java servlets.



## Why Bother with Servlets?

If you've ever written a C program or perl script for handling a CGI request, the reasons for using a servlet are obvious: it's in Java. Even if you are expert in such programs, you'll quickly realize that you really don't want to revisit all of the inelegancies of those two languages.

There are some more substantive reasons for using servlets, however. Since the servlet running program keeps the Java servlet loaded once it is requested, future requests from that or from other clients will run much faster than C or perl programs which must reload the entire program infrastructure once they are called. Finally, writing servlets in Java is so simple, it is silly to use more complicated approaches.

## A Simple Servlet

When you send data from a form to a web server, you use either the GET or the POST method. The GET method attaches all of the contents of the form you send to the server to the end of the URL. The POST method sends the form as part of the data stream. The POST method is somewhat preferred nowadays, but servlets can handle either one. The beginning of the Form section of the web page specifies the posting method and the machine to send the requests to:

```
<form action=http://blahblah.com:8080/servlet/CTSwim
method=POST>
```

This line sends the contents of the form to port 8080 (the default servlet port) of the server *blahblah.com*, specifies the CTSwim servlet and uses the POST method to send the form data to the servlet. The web server tells the servlet runner system to load this CTSwim program, initialize it and then calls its *doPost* method.

The simplest servlet is one that just says Hi when we call it. This is very simple to set up. First we write the submitting form:

```
<html>
<title>Simple servlet test</title>
<body>
<h1>Simple servlet test</h1>
<form action="../servlet/Hi" method="POST">
<input type="submit" value="Submit">
</form>
</body></html>
```

which just shows a submit button when displayed, as we see in Figure 1.

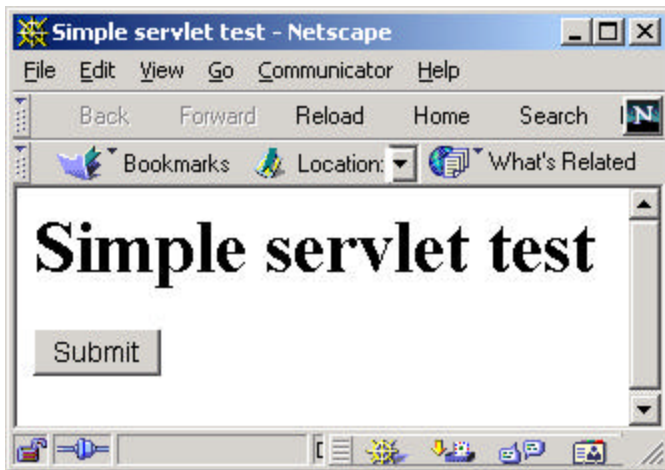


Figure 1 – A simple servlet.

The form's submit button calls the action routine “../servlet/Hi” using the POST method, although this simplest client does not send any parameters.

Then we write the simple servlet itself as Hi.java. It inherits from the HttpServlet class and has an init method that we simply pass on to the parent class. The only important part is the doPost method, which is called when you submit a POST request from the client. In this doPost method, you must generate the output data stream containing the HTML you want to send back to the server.

```
import javax.servlet.*;
import javax.servlet.http.*;

import java.io.*;
import java.util.*;
// Hi servlet derived from standard servlet example

public class Hi extends HttpServlet
    implements SingleThreadModel {

    //-----
    public void init(ServletConfig svg) throws
ServletException{
        super.init(svg);
    }
    //-----
}
```

```

    public void doPost (HttpServletRequest req,
        HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<center><h2>Hi</h2></center>");

        out.println("</body></html>");

    }
//-----
    public void destroy() {
        super.destroy();
    }
}

```

In this example, that data stream amounts to

```

<html>
<body>
<center><h2>Hi</h2></center>
</body></html>

```

Note that we also have to include the `setContentType` method call to specify that the text stream is HTML. The result is the display of a cherry “Hi” as shown in Figure 2.

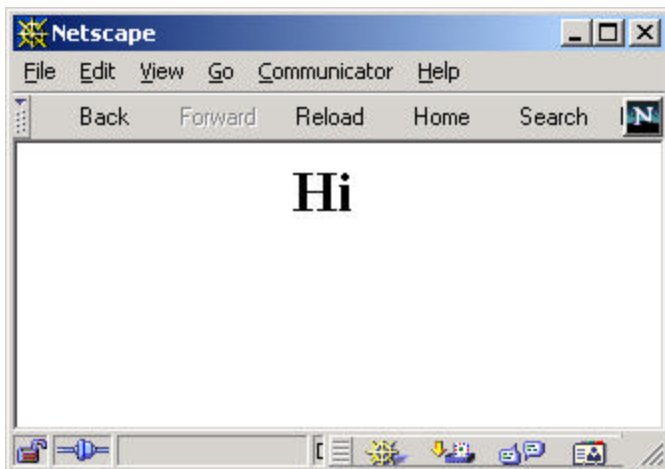


Figure 2 – Response to our first Hi servlet call.

The important part of the doPost method is that it is called by the system with the input and output streams already specified:

```
public void doPost (HttpServletRequest req,
HttpServletRequest response)
```

The input fields can be obtained from the HttpServletRequest object, and the output stream can be constructed from the HttpServletResponse object by making the call

```
PrintWriter out = response.getWriter();
```

Then, we use this PrintWriter object to construct the HTML output stream.

## Response to Fields in a Submitted Form

A more useful program is one in which we submit some information, and have the servlet return the results back to us:

```
<html>
<title>Simple Hello There Servlet</title>
<body>
<h1>Simple servlet test</h1>
<form action="../servlet/HiYou" method="POST">
<input type="text" name="name" size=20>
<input type="submit" value="Submit">
</form>
</body></html>
```

This is shown in Figure 3.



Figure 3 - A simple form we submit to a servlet with a name entered.

The output is nearly the same, but we get the input field from the `HttpServletRequest` object and print it out:

```
response.setContentType("text/html");
String name=req.getParameter("name");
PrintWriter out = response.getWriter();
out.println("<html>");
out.println("<body>");
out.println("<center><h2>Hi ");

out.println(name);

out.println("</h2></center>");
out.println("</body></html>");
```

## Ordering Dinner

In the previous chapter we developed a JavaScript form to order dinner. The form sends the data to the server using the POST method, and awaits a response. Now, lets finish that program by getting all the entered data from the input stream and printing it out for a thank you note



Figure 4- Order screen



and response screen.

The entire output routine in the servlet to generate the response is simply a matter of fetching the named fields from the input stream and generating the HTML formatting information:

```

public void doPost (HttpServletRequest req,
HttpServletResponse response)
    throws ServletException, IOException {

    String name = req.getParameter("Name");
    String wine = req.getParameter("wine");
    String meats = req.getParameter("meats");
    String soup = req.getParameter ("soup");
    String salad = req.getParameter ("salad");

    response.setContentType("text/html");

    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<body>");
    out.println("<center><h1>Thank you for your
order</h1>");
    out.println("<h2>" + name + "</center></h2>");
    out.println("<p>You ordered:");
    out.println("<ul>");
    out.println("<li>" + meats+"</li>");
    out.println("<li>" + wine + "</li>");
    if(soup != null)
        out.println("<li>soup</li>");
    if(salad != null)
        out.println("<li>salad</li>");
    out.println("</ul>");
    out.println("We'll send it right up!");
    out.println("</body></html>");
}

```

## Querying a Database

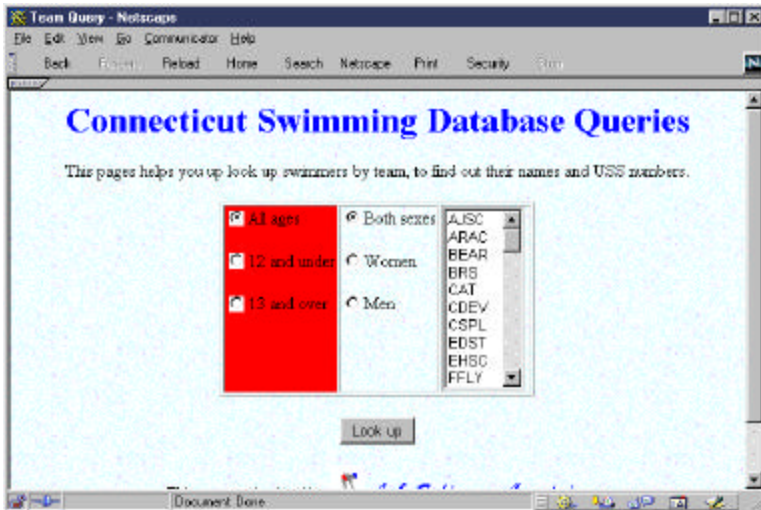
Let's consider a simple problem of providing a web site for a swimming organization, where users could look up the correct name and membership ID number of all the swimmers on each team. This is not just a static list problem, because both the teams and their members change quite often.

This seemed like an ideal case for a simple CGI query and servlet connection to a database of all the swimmers in the state. This system provides two screens:

1. Show a list of teams to select from.
2. Show a list of swimmers on the selected team.

Both of these screens require database queries. For the first, we need to obtain a current list of teams, and for the second a list of swimmers for the

selected team. So both screens must be generated dynamically by servlets. The team selection screen is shown below:



The form components in this table are defined for the age buttons as:

```
<input type="radio" name="Ages" value="All" >All ages
<input type="radio" name="Ages" value="12U"> 12 and under
<input type="radio" name="Ages" value="13O"> 13 and over
```

and for the sex selection buttons as

```
<input type="radio" name="Sex" value="Both">Both sexes
<input type="radio" name="Sex" value="F"> Women
<input type="radio" name="Sex" value="M"> Men
```

Note that all 3 radio buttons of the first group are named *Ages* and all three of the second group are named *Sex*. It is these group names that we use to find out which button has been selected.

## Writing the Servlet

The actual servlet code is very simple, and you can easily learn to write one from the examples supplied with the JSDK.

```
public class CTSwimServlet extends HttpServlet
    implements SingleThreadModel {
```

```

        protected Database db;          //database we talk to
    public void init(ServletConfig svg)
        throws ServletException{
        //open database during initialization
        db = new Database("sun.jdbc.odbc.JdbcOdbcDriver");
        db.Open ("jdbc:odbc:CTSwim99",null);
    }
    //-----
    public void doPost (HttpServletRequest req,
    HttpServletResponse res)
        throws ServletException, IOException {
    }
}

```

Then, inside the *doPost* method we get the parameters by name to decide the nature of the query to send to the database. Each of the radio buttons has the same *name* parameter, but a different *value* parameter. The three values for the *Ages* buttons are *All*, *12U* and *13O*. For example we can set the age range as follows:

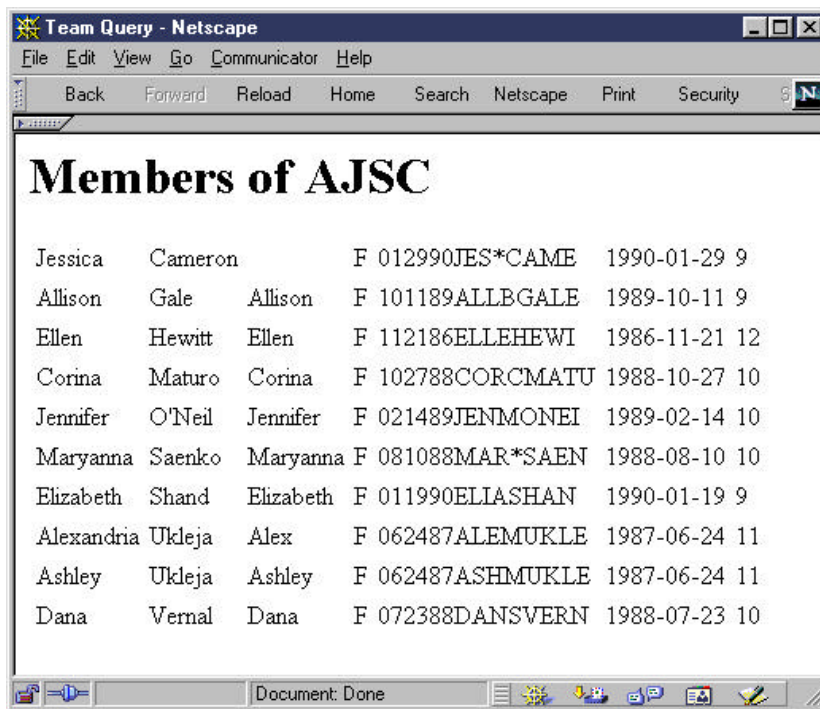
```

//12 and under
if (req.getParameter("Ages").equals("12U")) {
    maxage = 12;
    minage = 0;
}
//13 and over
if (req.getParameter("Ages").equals("13O")) {
    maxage = 100;
    minage = 13;
}

```

Then we can construct the query to the database and generate the output HTML file as shown below:





Sending the output just amounts to writing data to the output channel we can obtain from the *HTTPServletResponse* object.

```

PrintWriter out = res.getWriter();
res.setContentType("text/html");

out.println("<html>");
out.println("<head><title>Team Query</title></head>");
out.println("<body>");

String team = req.getParameter("teams"); //get
team name
out.println("<h1>"+ "Members of "+team+"</h1>");

```

Then we query the database and write out a table a line at a time with the various database fields we need as table cells.

## Are There Really Two Servlets?

The first screen we showed above shows a list box of team initials and radio buttons for age and sex. Since the teams change frequently, that list must be generated from the database each time a query begins. Thus, it would seem

that we need to create that page with the current team list using a second servlet.

On the other hand, these two pages share a common database and really amount to separate queries against that database. So, how can we easily decide what do if there is only one servlet? The simplest way is to use a hidden parameter on the querying web page

```
<input type=hidden name=servletType value=MakeTeams>
```

and have a hidden parameter having the *servletType* name but a different *value* on each page that calls your servlet. Then the servlet can check to see which methods are to be called and dispatch them appropriately. This leads to some simplifications in our servlet structure. Both of these servlet calls use many of the same methods, and we could simply check the value of the hidden *servletType* parameter and call the right method in a single servlet class, but it is cleaner and more scalable to make each of these methods a separate class.

Let's consider a base class called *ServletProcessor* which contains at least the following:

```
public abstract class ServletProcessor {
    protected HttpServletRequest req;
    protected HttpServletResponse res;
    protected Results rs;
    protected Database db;
    protected PrintWriter out;

    public void setHttp(HttpServletRequest rqst,
        HttpServletResponse rsp, Database dbase) {
        req = rqst;
        res = rsp;
        db = dbase;
    }
    //-----
    public abstract void Execute(PrintWriter outPrint);
    //-----
    protected void print (PrintWriter out,
        String name, String value) {
        out.print(" " + name + ": ");
        out.println(value == null ? "&lt;none&gt;" : value);
    }
    //-----
    protected void print (PrintWriter out,
        String name, int value) {
```

```

        out.print(" " + name + ": ");
        if (value == -1) {
            out.println("&lt;none&gt;");
        } else {
            out.println(value);
        }
    }
}

```

Then for each actual query, you only need to implement the `Execute` method. For creating the web page containing the list of teams, this `Execute` method is just:

```

public void Execute(PrintWriter out) {

    String query =
        "SELECT Clubs.ClubCode, Clubs.ClubName"+
        " FROM Clubs ORDER BY Clubs.ClubCode;";
    Results rs = db.Execute (query);

    res.setContentType("text/html");
    out.println("<html>");
    out.println("<head><title>Team Query</title></head>");
    out.println("<body>");
    //read in template html file and fill in team list
    InputFile fl = new InputFile("CTTQuery.htm");    //get
template file
    String s = fl.readLine();
    while (s.indexOf ("<option>")<=0) {
        out.println (s);
        s = fl.readLine();
    }
    while (rs.hasMoreElements ()) {
        out.println ("<option>" + rs.getColumnValue
("ClubCode"));
        //rs.nextElement ();
    }
    s = fl.readLine();
    while (s != null) {
        out.println (s);
        s=fl.readLine();
    }
}
}

```

The *Results* and *Database* classes are the ones we developed earlier.

The basic program is the `CTSwimServlet` and it instantiates instances of the abstract `ServletProcessor` class. The two instances in this simplified example are *TeamQuery* and *MakeTeamList*. As we noted above, the `CTSwimServlet` program can decide which class to instantiate based on the hidden form parameter *servletType*. One clever way to select the correct class is using a

Hashtable where instance of the classes are stored with the servlet type strings used as keys. Using this simple approach our entire main servlet program becomes just:

```
public class CTSwimServlet extends HttpServlet
    implements SingleThreadModel {

    protected Database db;          //database we talk to
    private Hashtable servList;     //list of servlet proc
    classes
    private ServletProcessor proc = null;

    public void init(ServletConfig svg)
        throws ServletException{
    super.init(svg);
    db = new Database("sun.jdbc.odbc.JdbcOdbcDriver");
    db.Open ("jdbc:odbc:CTSwim99",null);
    servList = new Hashtable();

    servList.put ("MakeTeams", new MakeTeamList());
    servList.put ("CTSwimTeams", new TeamQuery());
    }
    //-----
    public void doPost (HttpServletRequest req,
    HttpServletResponse res)
        throws ServletException, IOException {

        String stype = req.getParameter("servletType");
        proc = (ServletProcessor)servList.get(stype);
        proc.setHttp ( req, res, db);
        proc.Execute(res.getWriter());
    }
}
```

The nice thing about the simple approach is how easily it scales. As your needs for new queries grow, you just create new subclasses of ServletProcessor and add them to the hash table on startup. Since I began this article, I've add 3 more queries to the site, each requiring only a few minutes programming of a new subclass of ServletProcessor.

## Multiple User Accesses to Your Servlet

In a real-world system, many users might make requests to your servlet at once, and each will be launched as a separate thread. This means that your servlet application must not assume that the contents of class-level variables are invariant. You can do a lot to handle this if you are careful with the *synchronized* keyword, but for the simple example here, we simply had our

base server class implement the *SingleThreadModel* interface. This assures that each user access your servlet serially and will prevent thread confusion. Of course, if you have a high number of users this is not a sufficiently elegant solution. In this case you should make sure that there are few if any class-level variables that might get stomped on and that the rest are accessed only in synchronized processes.

## Running Servlets on Your System

The JavaServer Pages development kit provides a low traffic test server that you can use to test servlets and the JavaServer pages we will discuss in the following chapter. This product was developed by Sun, but donated to the Apache project, and is available as the Tomcat server. After you install Tomcat, you will find a `startserver.bat` file for Windows or `startserver` script for Unix which will launch the servlet engine. While it is running, you can address the server through port 8080 on your machine. If you start at <http://localhost:8080/> you will see a display of a page with servlet and JSP examples you can run.

To write your own servlet most simply, you place the Java class code in the

```
jakarta-tomcat\webapps\examples\WEB-INF\classes
```

directory and the web page file in the

```
jakarta-tomcat\webapps\examples\servlets
```

directory. You then address the servlet in the `action=` field as

```
<form action="../../servlet/javaclassname" method="POST">
```

To place the servlets in other directories of your own creation, you need to declare these directories in the file `web.xml` following the style shown in that file. This allows you to create new directories under

```
jakarta-tomcat\webapps\examples\WEB-INF\
```

## Summary

Servlets are small Java programs that run on a web server, that you can use to process requests from client web pages. They make receiving data easy,

but you still have to generate the output pages yourself. We'll see a more complete solution, when we discuss JavaServer pages in the next chapter.

## 26. Using Java Server Pages

As we have seen, servlets are a Java technology to replace cgi-bin client-server programming with a more efficient model. Ordinary cgi-bin programs are launched by the web server when HTML form pages are sent to a particular directory. These cgi-bin programs parse the form information and create a new dynamic HTML response page which they return to the client. Java servlets improve on this by keeping a servlet engine running which catches cgi-bin programs and parses them in Java. The servlet then computes the desired response page and returns it to the client in much the same way.

The problem with both of these approaches is that there is no really good way to construct complex output HTML pages. Usually, programmers rely on some kind of template pages which they copy and fill in with data from a particular cgi-bin query. This is one reason why most response pages are simple looking, and many are downright ugly.

Java Server Pages are one solution to remedy this problem. A JSP contains both the HTML to create the final output page and the Java to fill in the parts that must be computed. It also does the form parsing directly and almost invisibly for you. Java Server Pages are supported by a class of programs called Application Servers. There are quite a list of such packages, some of which add on to existing web servers and others of which perform both the web server and the JSP functions directly. You will find a complete list of current products at [java.sun.com/products/jsp](http://java.sun.com/products/jsp). Some of the common application servers include IBM's Websphere, Sun/Netscape's IPlanet, and BEA Weblogic.

### **Installing the JSP Development Kit**

You will also find pointers to the complete JSP development kit there, under the name Tomcat. If you download this package and install it, you will have a complete working JSP system where you can develop and test JSPs as well as servlets and then deploy them to your actual site. The JSP engine provided in this kit is perfectly adequate for rather small sites, but for larger sites you really need one of the application servers. It supports both JSPs and ordinary servlets.

To try out the servlet engine on Windows NT, go to the tomcat directory and run startup.bat. On Unix machines, run the startup shell script that is also

provided. To see the test programs that come with the JSWDK, start your web browser, and enter the URL

<http://localhost:8080>

This brings up a page describing the available examples and the servlet API. If you click on the JSP Samples link you will get the page shown in Figure 1. You can learn a lot from these examples, by watching them run and by examining the code. We'll start at the beginning with a "Hi" program and work up to some of the concepts they illustrate.

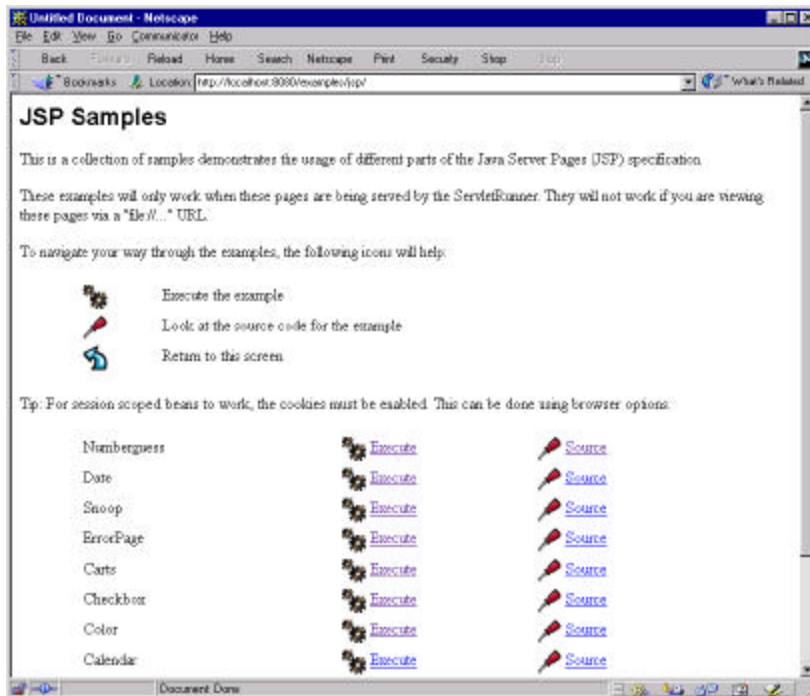


Figure 1 – The JSP samples page.

## A Simple Hello World JSP program

A JavaServer Page consists of HTML, of JSP directives and of actual Java code. The JSP engine compiles this code and caches it when you first access the page. This makes further accesses to the page much faster. The JSP page is a server-side concept: the actual code that is sent back to the client is pure HTML generated by the code on that page. This makes the client



requirements much less demanding than ones that require particular levels of Java support and the like.

Our first simple program, hi.jsp consists of a simple form, some output and communication with a back end Java bean. All JSP pages must start even before the <html> tag with declarations defining the fact that it is a JSP and the Java bean it will communicate with:

```
<%@ page language="java" import="mine.JTest" %>
OR
<jsp:useBean id="mine" scope = "page" class="mine.JTest" />
```

The first line is a jsp directive, describing the language (only Java is currently supported) and the bean the page will communicate with. The second line is the same jsp directive in an alternate format. You only need one or the other, although both do no harm. Note that the longer form follows the XML custom, terminating the tag with a /> to indicate that nothing more follows that relates to that tag.

## The Java Bean and the JSP Page

Most JSP pages connect with simple Java beans which contain and return the contents of the fields on the page. The beans that JSPs use are invisible server-side beans which just have get and set methods. They have no visual properties.

For example, you might have a <form> section on your page and inside it a text entry field.

```
<input type="text" name="entryText" size="25">
```

The associated Java bean then has get and set methods for this field:

```
package mine;
//A simple Java bean for the Hi demonstration programs
public class JTest {
    private String text; //text stored here

    public JTest() {
        text = null;
    }
    //set text here
    public void setEntryText(String t) {
        text = t;
    }
}
```

```

    }
    //get it back here
    public String getEntryText() {
        return text;
    }
}

```

Note that the case is odd but required. The name of the field must start in lower case, but the get and set methods convert it to upper case. When you begin using this bean on a JSP, you must initialize all of its properties in order to access them.

```
<jsp:setProperty name="mine" property="*" />
```

This line initializes all the bean's properties at once. This statement is required, or your bean properties won't work.

## Communicating with the Java Bean

The complete form section of our JSP page looks like this;

```

<form method="post">
<input type="text" name="entryText" size="25">
<br>
<input type="submit" value="submit">
</form>

```

Note that while there is a post method specified, there is no *action* specified which tells the page where to send the form. Instead, submit just sends all of the form data to the Java bean, executing the set method for each form field. In this case, the JSP calls the bean's *setEntryText* method with the string from the entry field as its argument. Thus, following the submit, the Bean contains all the data in the form's entry fields.

You can access that data using the get methods, and can print it out on the web page using JSP statements:

```

<%
out.println(mine.getEntryText());
%>

```

Note that you can actually embed real Java statements on a JSP page as long as then are enclosed in the <% and %>. You can also use a shorter form, where the equals sign replaces the "out.println" method call:

```
<%= mine.getEntryText() %>
```

This latter form is not a complete Java statement and does not need a terminating semicolon. Since it is so compact, you see it frequently.

## Our First Complete JSP

Combining the above fragments, we have a complete page as follows:

```
<jsp:useBean id="mine" scope = "page" class="mine.JTest" />
<jsp:setProperty name="mine" property="*" />

<html>
<head><title>Hi</title></head>
<body>

<form method="post">
<input type="text" name="entryText" size="25">
<br>
<input type="submit" value="submit">
</form>

Hello there: <%= mine.getEntryText() %>

</body>
</html>
```

This program initializes the *mine* Bean and uses the submit button to load it. Then it prints out the result.

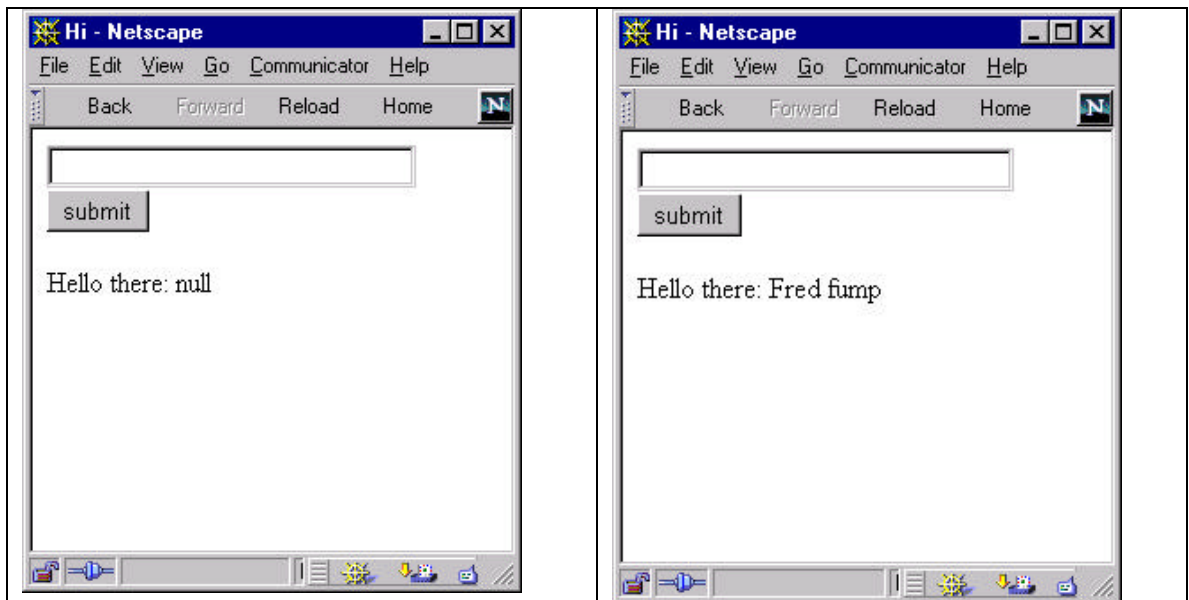


Figure 2 – The simple hi1.jsp program before and after entering a name and pressing submit.

The displays in Figure 2 are not very satisfactory, because we display “Hello there: null” before any name has been submitted. It would be better to display this only when there is real data. The second JSP example accomplishes this by testing for whether the name is non-null before displaying it:

```
<jsp:useBean id="mine" scope = "page" class="mine.JTest" />
<jsp:setProperty name="mine" property="*" />

<html>
<head><title>Hi</title></head>
<body>

<form method="post">
<input type="text" name="entryText" size="25">
<br>
<input type="submit" value="submit">
</form>

<% if (mine.getEntryText() != null) { %>
    Hello there:
    <%= mine.getEntryText() %>
<% } %>

</body>
</html>
```

In this example we do not see any “Hello there” text or null name displayed. Note how we can interweave Java and HTML in the same page to make decisions as to what to display. This is both the greatest strength and weakness of JSPs. You can write Java code to display data, but interleaving it makes the page look unstructured and confusing.

Now the purpose of JSPs is to keep the HTML and the server side logic together but logically separated to diminish clutter. The final example hi3.jsp just moves the test for printing Hello there and a name to another file which is included.

```
<jsp:useBean id="mine" scope = "page" class="mine.JTest" />
<jsp:setProperty name="mine" property="*" />

<html>
<head><title>Hi</title></head>
<body>

<form method="post">
```

```

<input type="text" name="entryText" size="25">
<br>
<input type="submit" value="submit">
</form>

<%@ include file="myResponse.jsp" %>

</body>
</html>

```

This gives the web designer control over the base jsp file and the server programmer control over the output in the myResponse.jsp, which is much the same code as in hi2.jsp:

```

<% if (mine.getEntryText() != null) { %>
    Hello there:
    <%= mine.getEntryText() %>
<% } %>

```

## Where to put all these classes?

If you install the Tomcat server as we described earlier, you will have a directory tree which contains

```
jakarta-tomcat\webapps\examples\jsp
```

along with a whole lot of little directories underneath for each of the provided examples. Create a directory under this for your jsp files.

The Java bean goes into the hierarchy

```
jakarta-tomcat\webapps\examples\WEB-INF\classes
```

Create a folder having the package name of your bean and place the class or jar files there.

## Summary

You can see the power of JSPs compared to other approaches. However, you can also see the clutter and confusion they can generate. Newest versions of several of the Java IDEs contain methods for building JSPs a little more automatically. You can use Borland's JBuilder, Visual Café, and indirectly can use IBM's Visual Age for Java to generate these JSPs.



## 27. More Sophisticated JavaServer Pages

As we discussed in the previous chapter, JSPs operate in conjunction with a non-visual JavaBean, which is simply a class with a bunch of get and set methods for various parameters on your input form. For example, if there is an input field called “name”

```
<input type="text" name="name" size="25">
```

you need to have setName and getName methods within the JavaBean to receive that data. There must be a setXxx method for every named field in the form. There need not be getXxx methods unless you need to retrieve the value of that field back from the bean. Since this is Java, you can actually use several derived classes inside the bean and decide which class to return using some simple software factory.

Let’s consider the case of a simple input form where you can enter your name either as *firstname lastname* or as *lastname,firstname*. We want to write a server process that can take these two cases apart and return the first and last names reliably. Of course, this is a ridiculously simple program that you could write in JavaScript on the client input page just as well, but we’ll use it as a model for more complex code you might want to write in real life.

### A Names Factory

We’ll start by considering the Bean we have to write. It will decide which order the name is entered in and select one of two classes. We’ll start by defining a base Namer class which has getFirst and getLast methods, but makes no decisions on name order:

```
package Names;
//A simple base class for both orders of names
public class Namer {
    //store the names here
    protected String first, last;

    //return the first name
    public String getFirst() {
        return first;
    }
    //return the last name
    public String getLast() {
        return last;
    }
}
```

Then, we derive the FirstFirst and LastFirst classes from it: each having a different constructor. Here is the first name class.

```
package Names;
public class FirstFirst extends Namer {

    //separates the leading first name
    public FirstFirst(String s) {
        int i = s.lastIndexOf(" ");
        if(i > 0) {
            first = s.substring(0, i).trim();
            last = s.substring(i + 1).trim();
        }
        else {
            first = "";
            last = s;
        }
    }
}
```

and here is the analogous lastname class:

```
package Names;
public class LastFirst extends Namer {
    //makes ht last name all to the left of the first comma
    public LastFirst(String s) {
        int i = s.lastIndexOf(",");
        if(i > 0) {
            last = s.substring(0, i).trim();
            first = s.substring(i + 1).trim();
        }
        else {
            first = "";
            last = s;
        }
    }
}
```

Now the class that decides which of these classes to invoke is the NamingBean class. Since it chooses one of several related classes, we refer to it as a Factory class:

```
package Names;
public class NamingBean {
    private Namer namer = null;

    //This selects one of two Namer subclasses
    public void setName(String nm) {
        int i = nm.indexOf(",");
        if( i > 0 )
            namer = new LastFirst(nm);
    }
}
```



```

        else
            namer = new FirstFirst(nm);
    }
    //return the last name
    public String getLast() {
        return namer.getLast ();
    }
    //return the first name
    public String getFirst() {
        return namer.getFirst ();
    }
    //get the instance of the Namer class
    public Namer getNamer() {
        return namer;
    }
}

```

The critical part of the NamingBean class is that it creates an instance either of FirstFirst or of LastFirst and stored that instance in the variable *namer*. Then the getFirst and getLast methods just return the value computed by whichever instance of the two classes currently occupies the *namer* variable. With that in mind, we can create the JSP itself.

## The JavaServer Page

This page declares that it uses the NamingBean class and sets up a form to submit with a text field where you can enter a name. Note that there is no *action=* modifier to the form. Instead, it just sends the form values to the bean we specify.

```

<%@ page language="java" import="Names.NamingBean" %>
<jsp:setProperty name="NameBean" property="*" />

<html>
<body bgcolor="c0c0ff">
<center>
<h2>Enter your name below</h2>

<form method="post">
<input type="text" name="name" size="25">
<br>
<input type="submit" value="Submit">
</form>

<% if (NameBean.getNamer() != null) { %>

<br><b>First name:</b> <%= NameBean.getFirst()%>
<br><b>Last name:</b> <%= NameBean.getLast()%>

```

```
<% } %>

</body>
</html>
```

The bottom half of the JSP consists of an if test to see if the *getNamer* method returns null or not. If it is null, no value has been entered yet for a name. If it is not null, the values for the first and last name are printed as part of the form. You can see the form displayed before and after the Submit button is clicked in Figure 1 and 2.



Figure 1 – The JSP before submitting it.



Figure 2 – The JSP after submitting it.

## More Class and Less Containment

In the previous example, we used the Factory to generate an instance of one of the two Namer classes and store it inside the bean. This is OK, but it does mean that the bean has to contain the same *getLast* and *getFirst* methods that the Namer class does and simply pass them on. It would be better if we returned the actual instance of the Namer class and called its methods directly. This is what we do in the second version of the JSP shown below.

```
<%@ page language="java" import="Names.NamingBean" %>
<jsp:useBean id="NameBean" scope = "page"
class="Names.NamingBean" />
<jsp:setProperty name="NameBean" property="*" />
```

```

<html>
<body bgcolor="c0c0ff">
<center>
<h2>Enter your name below</h2>

<form method="post">
<input type="text" name="name" size="25">
<br>
<input type="submit" value="Submit">
</form>

<% if (NameBean.getNamer() != null) {
    Names.Namer nm = NameBean.getNamer();
    out.println("<br><b> First name:</b> " + nm.getFirst());
    out.println("<br><b> Last name: </b> " + nm.getLast());
}
%>

</body>
</html>

```

In this version, we use the `getNamer` method to return an instance of the `Namer` class and call its methods right in the JSP code. Note that we have to declare the class using the package name prefix and that the `Namer` class is separate from the `NameBean` class we used as our `Bean`. However, we have now used a `Factory` in the `Bean` to return one of several classes to our `JavaServer Page` and have thus developed an example of the flexibility of the JSP system we don't find in other scripting languages, such as ASP.

## Accessing Databases

Let's suppose we want to create a web page to look up the grocery store having the lowest prices for various foods. We'll assume that comparison shoppers update the database regularly so we can comparison shop using our web site. IN our simple example, we'll create an Access database of foods, stores and prices. Figure 3 shows the `Food` and `Stores` tables and Figure 4 shows the relational table between the food, the store and the prices.

The image shows two side-by-side database table windows. The left window is titled 'Stores : Table' and contains three records with columns 'StoreKey' and 'StoreName'. The right window is titled 'Food : Table' and contains seven records with columns 'FoodKey' and 'FoodName'. Both windows have a 'Record:' indicator at the bottom showing the current record number.

StoreKey	StoreName
1	Stop and Shop
2	Village Market
3	Waldbaum's

FoodKey	FoodName
1	Apples
2	Oranges
3	Hamburger
4	Butter
5	Milk
6	Cola
7	Green beans

Figure 3 – The Stores and Food database tables

FSKey	StoreKey	FoodKey	Price
1	1	1	\$0.27
2	2	1	\$0.29
3	3	1	\$0.33
4	1	2	\$0.36
5	2	2	\$0.29
6	3	2	\$0.47
7	1	3	\$1.98
8	2	3	\$2.45
9	3	3	\$2.29
10	1	4	\$2.39
11	2	4	\$2.99
12	3	4	\$3.29
13	1	5	\$1.98
14	2	5	\$1.79
15	3	5	\$1.89
16	1	6	\$2.65
17	2	6	\$3.79
18	3	6	\$2.99
19	1	7	\$2.29
20	2	7	\$2.19
21	3	7	\$1.99

\* (Number) 0 0 \$0.00

Record: 1 of 21

Figure 4 –The Food-Store-Price relations table.

We can easily write a database query in the SQL language to get the prices for any food in increasing order, but Access allows us to construct the query graphically as shown in Figure 5.

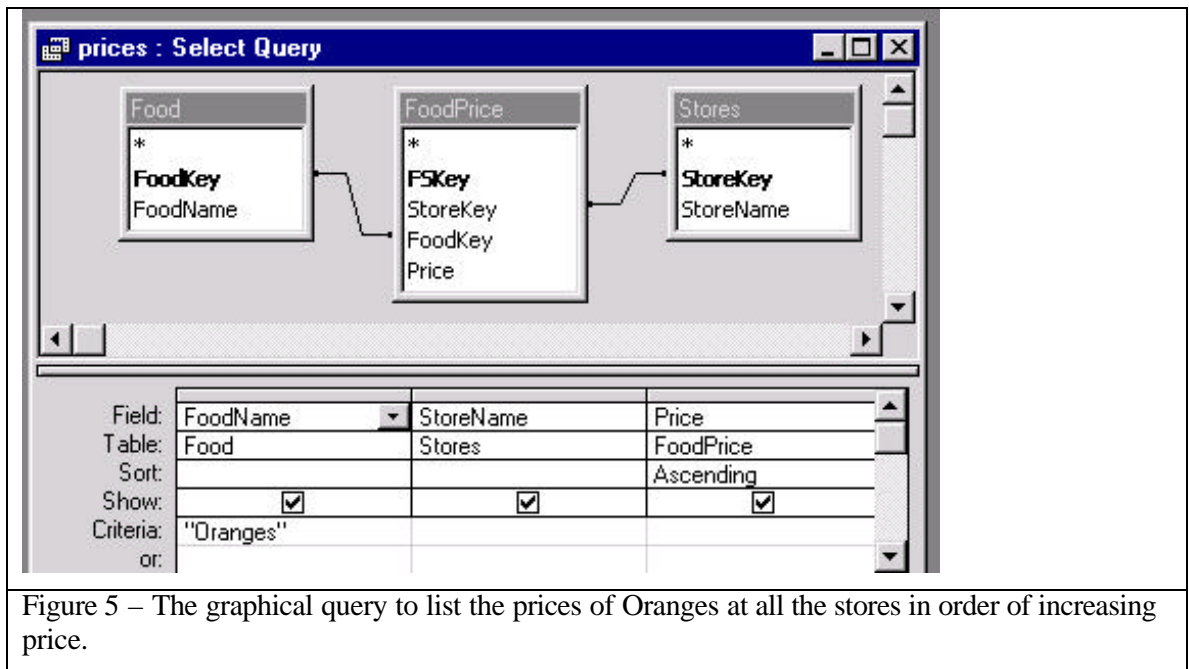


Figure 5 – The graphical query to list the prices of Oranges at all the stores in order of increasing price.

The actual SQL query it generate is just

```
SELECT DISTINCTROW Food.FoodName, Stores.StoreName,
FoodPrice.Price
FROM (Food INNER JOIN FoodPrice ON Food.FoodKey =
FoodPrice.FoodKey) INNER JOIN Stores ON FoodPrice.StoreKey =
Stores.StoreKey
WHERE (((Food.FoodName)="Oranges"))
ORDER BY FoodPrice.Price;
```

## The Database Bean

Our database bean is fairly simple. We just provide a setFood method and a method to get an Enumeration of the foods in the database.

```
package Groceries;
import java.net.URL;
import java.sql.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;

public class grocBean {
    Database db;           //database
    Results rs, fdrs;     //result sets
```

```

Vector foods;           //foods listed here
Enumeration eFood;     //an enumeration of the foods
boolean queryDone;    //true if query done
String food;          //current food

public grocBean() {

    db = new Database("sun.jdbc.odbc.JdbcOdbcDriver");
    db.Open("jdbc:odbc:Grocery prices", null);
    queryDone = false;

}
//-----
public void getFoods() {
    System.out.println("getting foods");
    String query = "Select FoodName from Food Order by
FoodName";
    fdrs = db.Execute (query);

    Vector foods = new Vector();
    while (fdrs.hasMoreElements ()) {
        foods.add((String)fdrs.getColumnValue
("FoodName"));
    }
    eFood= foods.elements();
}
public boolean hasMoreFoods() {
    boolean hasMore = eFood.hasMoreElements ();
    if(! hasMore)
        fdrs.close ();
    return hasMore;
}

public String nextFood() {
    return (String)eFood.nextElement ();
}
//-----
public void setFood(String myfood) {
    food = myfood;
    String queryText ="SELECT DISTINCTROW FoodName, StoreName,
Price"+
"FROM (Food INNER JOIN FoodPrice ON Food.FoodKey =
FoodPrice.FoodKey) "+ "INNER JOIN Stores ON FoodPrice.StoreKey
= Stores.StoreKey "+
"WHERE (((Food.FoodName)='\'' + food + '\'')) ORDER BY
FoodPrice.Price;";

    rs = db.Execute(queryText);
    queryDone = true;
}
}

```

```

//-----
public String getFood() {
    return food;
}
public boolean hasData() {
    return queryDone;
}
public String getPrice() {
    return rs.getColumnValue("Price");
}
public String getStore() {
    return rs.getColumnValue ("StoreName");
}
public void nextElement () {
    rs.nextElement ();
}
public boolean hasMoreElements() {
    boolean hasMore = false;
    if(rs != null) {
        hasMore = rs.hasMoreElements ();
        if(! hasMore)
            rs.close ();
    }
    else
        hasMore = false;
    return hasMore;
}
}

```

Note that in this example we are using the Database and Results classes we developed earlier in this book.

The JSP for this query is just as simple as you might think. We enumerate the foods in the database and store them in a list box. When we submit the query, we detect which food is selected and return a table of the prices at various local stores.

```

<%@ page language="java" import="Groceries.grocBean" %>
<jsp:useBean id="grocBean" scope = "page"
class="Groceries.grocBean" />
<jsp:setProperty name="grocBean" property="*" />

<html>
<head><title>Grocery Prices</title></head>
<body bgcolor = "80ffff">
<!-- ----- -->
<center>
<form method = "post">

```



```

<select name = "food" size = "10">
  <% grocBean.getFoods();
  while (grocBean.hasMoreFoods()) { %>
    <option> <%= grocBean.nextFood() %>
  <% } %>

</select>
<br>
<input type=submit value="Submit">
</form>

<% if (grocBean.hasData()) { %>
  <h2> Prices of <%= grocBean.getFood() %> </h2>
  <table>
bbbbbbbbbb<% } %>

<% while(grocBean.hasMoreElements()) { %>
  <tr>
    <td> <%= grocBean.getStore() %></td><td> <%=
grocBean.getPrice() %> </td>
  </tr>

<%}%>
  </table>
</center>

</body>
</html>

```

We show the results of a simple query in Figure 6.

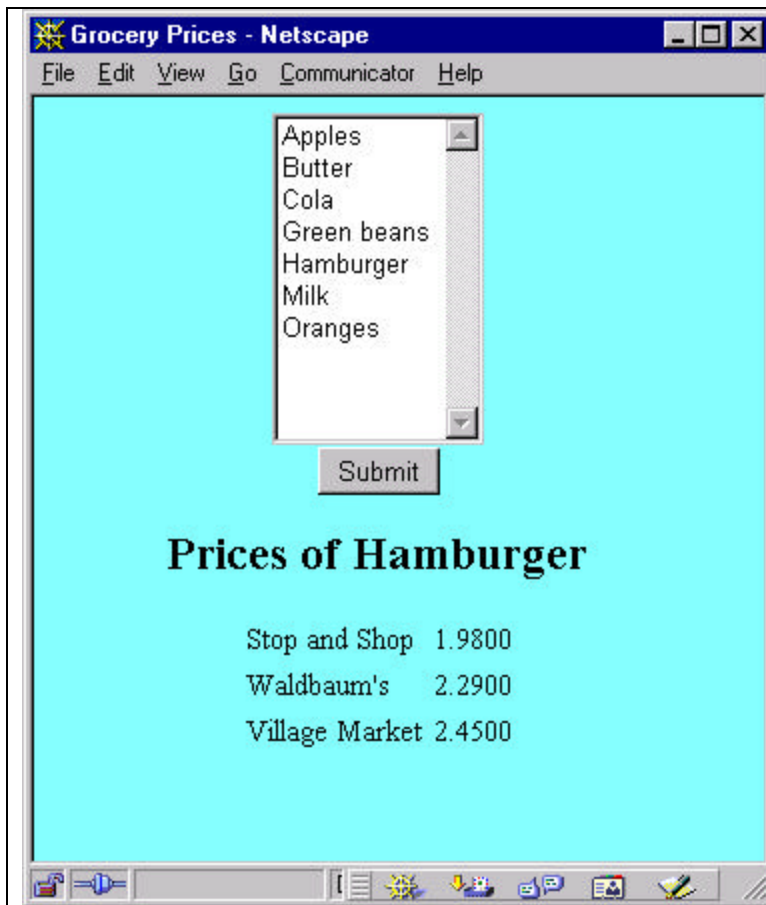


Figure 6 – The results of querying the database for Hamburger prices.

## Conclusions

In this chapter, we've seen some of the really powerful techniques that JavaServer pages make easy for you. Now you've seen the entire sweep of Java from simple clients, to Swing classes to JDBC and database to client-server programming, and you are ready to tackle most real world problems with some assurance.

We've shown some more powerful uses of JavaServer Pages in this column. We showedn how you can use a Factory to return a class to the clent page