10. Introduction to the Swing Classes

The Java Foundation Classes (JFC or “Swing”) are a complete set of light-weight user interface components that enhance, extend and to a large degree replace the AWT components such as buttons, panels, listboxes and checkboxes. In addition to the buttons, lists, tables and trees in the JFC, you will also find a pluggable look-and-feel that allows the components to take on the appearance of several popular windowing systems, as well as its own look and feel. The JFC actually uses a few common design patterns, and we will be using the JFC for most of the examples in this book. Thus, in this appendix, we review how to use the JFC and point out a number of the patterns it encompasses. If you haven’t used these powerful classes yet, here is a simple introduction. If you have, read through this chapter to see the number of patterns you can discover in them.

We should note at the outset, that this package was called “Swing” during development and it was intended that it be referred to as “JFC” upon release. However, the nickname has stuck, and this has led to the Java programmer’s explanation: “it’s spelled JFC, but it’s pronounced Swing.”

Installing and Using the JFC

All programs which are to make use of the JFC, must import

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

and may require one or more of the following depending on the components you use.

```java
import javax.swing.event.*;
import javax.swing.border.*;
import javax.swing.text.*;
```

Ideas Behind Swing

The Swing components are referred to as “lightweight” components, because they don’t rely on native user-interface components. They are, in fact, 100% pure Java. Thus, a Swing JButton does not rely on a Windows button or a Motif button or a Macintosh button to implement its functionality. They also use fewer classes to achieve this interface than the
previous heavier-weight AWT classes. In addition, there are many more Swing user-interface components than there were AWT components. Swing gives us image buttons, hover buttons, tooltips, tables, trees, splitter panels, customizable dialog boxes and quite a few other components.

Since Swing components create their look and feel completely within the Swing class hierarchy, you can have a pluggable look and feel to emulate Windows, Motif, Macintosh or the native Swing (metal) look.

Many Swing components make use of an architecture derived from the model-view-controller design pattern. The idea of this MVC pattern, is to keep the data in a *model* class, display the data in a *view* class and vary the data and view using a *controller* class. We’ll see in later chapters that this is exactly how the JList and JTable handle their data.

### When to Use Swing Classes
You should use the Swing GUI classes whenever you want to have a professional looking user interface. The classes provide beveled borders, buttons and menus with images, tooltips, and more sophisticated lists, trees and tables. You should only avoid Swing classes in applets when some of your users may be using back-level browsers and don’t want to load the Java plug-in.

### The Swing Class Hierarchy
All Swing components inherit from the JComponent class. While JComponent is much like the AWT’s Component in its position in the hierarchy, JComponent provides the pluggable look and feel. It also provides

- Keystroke handling that works with nested components.
- A border property that defines both the border and the component’s insets.
- Tooltips that pop up when the mouse hovers over the component.
- Automatic scrolling of any component when placed in a scroller container.

Because of this interaction with the user interface environment, Swing’s JComponent is actually more like the AWT’s Canvas than its Component...
class. You also should avoid mixing AWT and Swing components in the same program, since they do not interact well.

Writing a Simple JFC Program

Getting started using the Swing classes is pretty simple. Application windows inherit from JFrame and applets inherit from JApplet. The only difference between Frame and JFrame is that you cannot add components or set the layout directly for JFrame. Instead, you must use the getContentPane method to obtain the container where you can add components and vary the layout.

```java
g getContentPane().setLayout(new BorderLayout());
JButton b = new JButton("Hi");
g getContentPane().add(b); //add button to layout
```

This is sometimes a bit tedious to type each time, so we recommend creating a JPanel, adding it to the content pane and then adding all the components to that panel.

```java
JPanel jp = getContentPane();
JButton b = new JButton("Hi");
jp.add(b);
```

JPanels are containers much like the AWT Panel object, except that they are automatically double buffered and repaint more quickly and smoothly.

Setting the Look and Feel

If you do not select a look and feel, Swing programs will start up in their own native look and feel (called Metal) rather than the Windows, Motif or Mac look. You must specifically set the look and feel in each program, using a simple method like the following:

```java
private void setLF() {
    // Force SwingApp to come up in the System L&F
    String laf = UIManager.getSystemLookAndFeelClassName();
    try {
        UIManager.setLookAndFeel(laf);
    } catch (UnsupportedLookAndFeelException exc) {
        System.err.println("UnsupportedLookAndFeel: "+laf);
    }
}```
System.err.println("Error " + laf + ": " + exc);

Setting the Window Close Box

Like the Frame component, the system exit procedure is not called automatically when a user clicks on the close box. In order to enable that behavior, you must add a WindowListener to the frame and catch the WindowClosing event. This can be done most effectively by subclassing the WindowAdapter class:

private void setCloseClick() {
    // create window listener to respond to window close click
    addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
}

Making a JxFrame Class

Since we must always set the look and feel and must always create a WindowAdapter to close the JFrame, we have created a JxFrame class which contains those two functions, and which calls them as part of initialization:

public class JxFrame extends JFrame {
    public JxFrame(String title) {
        super(title);
        setCloseClick();
        setLF();
    }
}

The setLF and setCloseClick methods are included as well. It is this JxFrame class that we use in virtually all of our examples in this book, to avoid continually retyping the same code.

A Simple Two Button Program

Now with these fundamentals taken care of, we can write a simple program with two buttons in a frame like the one in Figure 10-1.
Figure 10-1– A simple two button Swing program.

One button switches the color of the background and the other causes the program to exit. We start by initializing our GUI and catching both button clicks in an `actionPerformed` method:

```java
public class SimpleJFC extends JFrame
    implements ActionListener {
    private JButton OK, Quit;
    private Color   color;
    private JPanel  jp;

    public SimpleJFC() {
        super("Simple JFC Program");
        color = Color.yellow;
        setGUI();
    }

    //-------------------------------
    private void setGUI() {
        jp = new JPanel();
        getContentPane().add(jp);

        //create and add buttons
        OK =   new JButton("OK");
        Quit = new JButton("Quit");
        OK.addActionListener(this);
        Quit.addActionListener(this);
        jp.add(OK);
        jp.add(Quit);

        setSize(new Dimension(250,100));
        setVisible(true);
    }

    //-------------------------------
    public void actionPerformed(ActionEvent e) {
        Object obj = e.getSource();
        if (obj == OK)
            switchColors();
        if (obj == Quit)
            System.exit(0);
    }

```
The only remaining part is the code that switches the background colors. This is, of course, extremely simple as well:

```java
private void switchColors() {
    if (color == Color.green)
        color = Color.yellow;
    else
        color = Color.green;
    jp.setBackground(color);
    repaint();
}
```

That’s all there is to writing a basic JFC application. JFC applets are identical except for the applet’s init routine replacing the constructor. Now let’s look at some of the power of the more common JFC components.

**JButtons**

The JButton has several constructors to specify text, an icon or both:

- JButton(String text);
- JButton(Icon icon);
- JButton(String text, Icon icon);

You can also set two other images to go with the button

- setSelectedIcon(Icon icon); //shown when clicked
- setRolloverIcon(Icon icon); //shown when mouse over

Finally, like all other JComponents, you can use setToolTipText to set the text of a ToolTip to be displayed when the mouse hovers over the button. The code for implementing these small improvements is simply

```java
OK = new JButton("OK", new ImageIcon("color.gif");
OK.setRolloverIcon(new ImageIcon("overColor.gif");
OK.setToolTipsText("Change background color");
Quit = new JButton("Quit", new ImageIcon("exit.gif");
Quit.setToolTipsText("Exit from program");
```

The resulting application window is shown in Figure 10-2 with the tooltip up.
RadioButtons and Toolbars

Swing provides separate implementations of both the JRadioButton and the JCheckBox. A checkbox has two states and within a group of checkboxes, any number can be selected or deselected. Radio buttons should be grouped into a ButtonGroup object so that only one radio button of a group can be selected at a time.

Radio Buttons

Both radio buttons and check boxes can be instantiated with an image as well as a title and both can have rollover icons. The JCheckBox component is derived from the simpler JToggleButton object. JToggleButton is a button that can be switched between two states by clicking, but which stays in that new state (up or down) like a 2-state check box does. Further the JToggleButton can take on the exclusive aspects of a radio button by adding it to a ButtonGroup.

```java
//create radio buttons in right panel
JRadioButton Rep, Dem, Flat;
right.add(Rep = new JRadioButton("Republicrat"));
right.add(Dem = new JRadioButton("Demmican"));
right.add(Flat = new JRadioButton("Flat Earth"));
ButtonGroup bgroup = new ButtonGroup();
bgroup.add(Rep);  //add to button group
bgroup.add(Dem);
bgroup.add(Flat);
```

If you neglect to add the radio buttons to a ButtonGroup, you can have several of them turned on at once. It is the ButtonGroup that assures that only one at a time can be turned on. The ButtonGroup object thus keeps track of the state of all the radio buttons in the group to enforce this only-
one-on protocol. This is a clear example of the Mediator pattern we’ll be discussing in the chapters ahead.

The JToolBar

The JToolBar is a container bar for tool buttons of the type you see in many programs. Normally, the JDK documentation recommends that you add the JToolBar as the only component on one side of a BorderLayout (typically the North side), and that you not add components to the other 3 sides. The buttons you add to the toolbar are just small JButtons with picture icons and without text. The JToolBar class has two important methods: add and addSeparator.

```java
JToolBar toolbar = new JToolBar();
JButton Open = new JButton("open.gif");
toolbar.add(Open);
toolbar.addSeparator();
```

By default, JButton has a rectangular shape. To make the usual square-looking buttons, you need to use square icons and set the insets of the button to zero. On most toolbars, the icons are 25 x 25 pixels. We thus develop the simple ToolButton class below, which handles both the insets and the size:

```java
public class ToolButton extends JButton {
    public ToolButton(Icon img) {
        super(img);
        setMargin(new Insets(0,0,0,0));
        setSize(25,25);
    }
}
```

The JToolBar also has the characteristic that you can detach it from its anchored position along the top side of the program and attach it to another side, or leave it floating. This allows some user customization of the running program, but is otherwise not terribly useful. It also is not particularly well implemented and can be confusing to the user. Thus, we recommend that you use the setFloatable(false) method to turn this feature off.

JToggleButton

The JToggleButton class is actually the parent class for check boxes and radio buttons. It is a two-state button that will stay in an up or down position when clicked, and you can use it just like a check box. While toggle buttons look sort of strange on most screens, they look very reasonable as part of toolbars. You can use individual toggle buttons to indicate the state of
actions the user might select. By themselves, toggle buttons behave like check boxes, so you can press as many as you want, and you can “uncheck” or raise toggle buttons by using the `setSelected(false)` method.

You can also add toggle buttons to a ButtonGroup so that they behave like radio buttons: only one at a time can be pressed down. However, once a ButtonGroup object is mediating them, you can’t raise the buttons using the `setSelected` method. If you want to be able to raise them, but still only allow one at a time to be pressed, you need to write your own Mediator class to replace the ButtonGroup object.

**Sample Code**

The simple program display in Figure 10-3 illustrates checkboxes, radio buttons, toolbar buttons and toggle buttons:

![Buttons and Checks](image)

*Figure 10-3 – A simple display of checkboxes, radio buttons, toolbar buttons and toggle buttons.*

Note the “b” JToggleButton stays depressed until another button is selected, as toggle buttons are supposed to do. While the user can select any number of organizations in which he holds memberships using the JCheckboxes, he can only select one political party from the JRadioButtons.

**Borders in Swing**

Swing provides a series of Border objects that are a kind of decorators. You can add a new Border object to any JComponent, and there also is a way to add several borders. Swing defines several standard border classes:
<table>
<thead>
<tr>
<th>Border Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BevelBorder((n))</td>
<td>Simple 2-line bevel, can be LOWERED or RAISED</td>
</tr>
<tr>
<td>CompoundBorder ((inner, outer))</td>
<td>Allows you to add 2 borders</td>
</tr>
<tr>
<td>EmptyBorder((top, left, bottom, right))</td>
<td>Blank border width specified on each side.</td>
</tr>
<tr>
<td>EtchedBorder</td>
<td>Creates etched border.</td>
</tr>
<tr>
<td>LineBorder((width, color))</td>
<td>Creates simple line border,</td>
</tr>
<tr>
<td>MatteBorder</td>
<td>Creates a matte border of a solid color or a tiled icon.</td>
</tr>
<tr>
<td>SoftBeveledBorder</td>
<td>Creates beveled border with rounded corners.</td>
</tr>
<tr>
<td>TitledBorder</td>
<td>Creates a border containing a title. Use this to surround and label a JPanel.</td>
</tr>
</tbody>
</table>

These borders are simple to use, in conjunction with the `setBorder` method of each JComponent. The illustration in Figure 10-4 shows a normal JButton with a 2-pixel solid line border, combined with a 4-pixel EmptyBorder and an EtchedBorder.

![Deco Button](image.png)

**Figure 10-4** – Dbutton has a 2-pixel solid line border and Quit has a 4 pixel EmptyBorder and an etched border.

This was created with the following simple code:

```java
JPanel jp = new JPanel();
getContentPane().add(jp);
jp.add( Cbutton = new JButton("Cbutton"));
jp.add( Dbutton = new JButton("Dbutton"));
EmptyBorder ep = new EmptyBorder(4,4,4,4);
```
LineBorder lb = new LineBorder(Color.black, 2);
Dbutton.setBorder(new CompoundBorder(lb, ep));
jp.add(Quit = new JButton("Quit"));
EtchedBorder eb = new EtchedBorder();
    Quit.addActionListener(this);
    Quit.setBorder(eb);

One drawback of these Border objects is that they replace the default Insets values that determine the spacing around the component. Note that we had to add a 4-pixel EmptyBorder to the Dbutton to make it similar in size to the CButton. We did not do this for the Quit button, and it is therefore substantially smaller than the others.

You can also affect the border around a JButton, rather like a CoolButton, using the `setBorderPainted(b)` method. If $b$ is false, the border is not drawn. You then would turn it back on and repaint when the button receives a mouseOver event.

The JList Class

The JList class is a more powerful replacement for the simple List class that is provided with the AWT. A JList can be instantiated using a Vector or array to represent its contents. The JList does not itself support scrolling and thus must be added to a JScrollPane to allow scrolling to take place.

In the simplest program you can write using a JList, you

1. add a JScrollPane to the Frame, and then
2. create a Vector of data
3. create a JList using that Vector
4. add the JList to the JScrollPane’s viewport

This is shown below

```java
JPanel jp = new JPanel();  //panel in Frame
getContentPane().add(jp);

//create scroll pane
JScrollPane sp = new JScrollPane();
    jp.add(sp);  //add to layout
Vector dlist = new Vector();  //create vector
dlist.addElement("Anchovies");  //and add data
```
dlist.addElement("Bananas");
dlist.addElement("Cilantro");
dlist.addElement("Doughnuts");
dlist.addElement("Escarrole");
JList list = new JList(dlist); //create list with data
sp.getViewport().add(list); //add list to scrollpane

This produces the display shown in Figure 10-5.

![JList demo](image)

**Figure 10-5 – A simple JList.**

You could just as easily use an array instead of a Vector and have the same result.

The JList is considerably more flexible than the AWT List class, but we will defer its sophisticated features to the advanced Swing chapter, where we will see that the JList, the JTable, and the JTree all support the Model-View-Controller pattern or Observer pattern, where the data are completely separate from the interface class.

**Summary**

We’ve taken an introductory look at the JFC or Swing classes in this chapter. They provide better looking more versatile components and do not depend on the underlying operating system to draw the components. We’ll look at some more of them in the advanced chapter.
11. Writing a Simple Visual Application in Java

Windows in Java

When you write visual applications in Java, you are writing programs that use and manipulate windows. All Java windows inherit from the Container class, but the inheritance tree beyond that has two branches. For applications, the hierarchy starts with the Window class, which is an undecorated square on the screen. The Frame class, derived from Window, is the one we usually see on the screen. It contains a title bar and may contain a menu.

Applets are also a kind of window derived from Panel. Panels are designed to be regions inside another container where you draw or arrange components in a particular way, and Applets are a kind of panel with specific methods that the web browser can call to initialize and start them running. The inheritance tree for Applets and Frames is shown in Figure 8-1.

Fig 8-1: The Inheritance Tree of Applets and Frames

Since both Applets and Frames are kinds of containers, it is not surprising that we add the visual controls that we wish to use into these containers. It is important to recognize that create instances of these graphical objects is only part of our job. We also have to tell the window that it is going to contain them, so it can display them by calling their paint methods. How and where it will display the controls is really dependent on the window’s layout manager, and we will consider several ways of laying out a window in this chapter.

Building a Temperature Conversion Applet

Now that we’ve outlined the common controls in a Java application, let’s build one. We are going to build a temperature conversion program that will allow us to convert temperatures between Fahrenheit and Celsius. This
program will consist of a text field, two labels, two radio buttons and a Compute button.

We are going to write this conversion program as an application. In this example, we are going to create a class called Tempcalc in a file called tempcalc.java. We will compile this into an applet called tempcalc.class which we will run from a HTML file called tempcalc.html. The structure of the applet consists of

1. The public class initializer routine where we initialize variables.
2. Layout of controls on the screen
3. Routines to process button click events and compute the result.

Laying out Controls on the Screen

Java layout managers are objects that arrange the positions of the controls dynamically so that they never overlap, based on rules embedded in the various layout manager classes. Further, layout manager-driven applications are independent of screen resolution and computer platform. They allow the layout to expand and stretch dynamically as a window is resized.

In this chapter we are first going to write the program while ignoring the details of layout managers and then we will revisit it using layout managers in the following chapter.

| Option button “Fahrenheit” | 125, 100 | 100 x 15 |

Applet Methods

All of the initialization we have performed in applications so far has been in the class’s constructor. Applets are launched by the web browser which displays the web page and the interface between the web browser and the applet is well defined.
The init() Method

After the applet is loaded and its constructor called, the browser calls its init method. This is the time at which connection is first made to the runtime environment and the earliest that making references to the layout of graphical objects is meaningful. So, for applets, rather than doing our initialization inside the constructor, we do it inside the init method.

The start() and stop() Methods

What happens if we display an applet on a web page and then scroll down or switch pages so it is no longer showing? Does it continue to run? Can it be restarted?

The web browser calls the init() method only once. Then it calls the applet’s start() method. The applet runs until that part of the web page is obscured either by scrolling away from it or by moving to another page. Then the browser calls the applet’s stop() method. If you scroll or switch back to the applet again, the browser again calls the start() method.

While you may not need to override these methods and write any actual code for many applets, they can be useful when you write animation programs.

Laying Out Our Applet

The private variables in our tempcalc applet consist of the graphical controls on the screen and the checkbox group object.

```java
private CheckboxGroup Temper;  //group radio buttons
private Label labell;
private TextField TempEntry;   //edit box
private Checkbox Fahr;         //radio button
private Checkbox Celsius;      //radio button
private Label result;          //result shown here
private Button Compute;        //compute when clicked
private Button Quit;           //exit when clicked
```
Creating Objects

Let’s consider that Font statement above for a minute.

```java
setFont(new Font("SavSerif", Font.PLAIN, 12));
```

What do we mean by this? Well, if you look at the constructor for the Font class you will find that it is

```java
public Font(String  name, int  style, int  size);
```

Then, if you look for the method `setFont` under Applet, you won’t find it. If you start going down the inheritance tree, you will pass through Panel and Container before you find the method in the base Component class:

```java
public void setFont(Font  f);
```

So in order to set the font to the new style or size, we have to create a Font object. We could do this by

```java
Font newfont = Font("Helvetica", Font.PLAIN,12);
setFont(newfont);
```

or we could do it by creating a new font object right in the `setFont` method call:

```java
setFont(new Font("Helvetica", Font.PLAIN, 12));
```

which is what we have done here.

Initializing the Controls

Then we begin initializing the controls. For each control, we create an instance of it using the `new` operator. Note that while we declared a variable `label1` of type `Label` above, we didn’t create an object to put in that variable. This is where objects differ from simple numeric types such as `float` or `int`. Objects are some sort of unspecified data structure containing data and pointers to methods. In order to use an object, you have to allocate space for it using `new`. 
So in the case of the `label1` object, we create a label using the label constructor which specifies the text of the label. Then we use the applet’s `add` method to add each control to the applet container.

```
label1 = new Label("Enter temperature:");
add(label1);
```

We do pretty much the same thing for each of the other controls. However, the Option (radio) buttons are a little difference since they are a special kind of Checkbox. First we need to create a CheckboxGroup object and then refer to it in the constructor of each of the option buttons that you want to have work together as a unit:

```
Temper = new CheckboxGroup();
Fahr = new Checkbox("to Fahrenheit",Temper, false);
add(Fahr);

Celsius = new Checkbox("to Celsius",Temper, false);
add(Celsius);
```

**Adding the Buttons and an ActionListener**

Now that we have initialized the controls and started the applet, we need to take action when the Compute or Quit buttons are clicked.

```
compute=new Button("Compute");
add(compute);
compute.addActionListener(this);

quit = new Button("Quit");
add(quit);
quit.addActionListener(this);
```

We declare the outside Applet to implement the ActionListener interface

```
public class tempcalc extends Applet
    implements ActionListener {
and include an actionPerfomed method to intercept the button presses:

public void actionPerformed(ActionEvent evt) {
    Object obj = evt.getSource ();
    if (obj == compute) {
        clickedCompute();   //compute temperature
    } else
        if (obj == quit) {

Copyright © 2000, by James W. Cooper
System.exit(0); //exit from application
}
}

Note that the Quit button calls the System.exit(0) method. This method only
applies to stand-alone applications, and has no effect in applets. This
program is designed to work in both modes, however.

The clickedCompute Method

Now to the crux of the program. We need to read the value in the TextField
box TempEntry and then see whether the Fahr button is selected and
perform one calculation or the other. We will need two local variables:

float temp, newtemp;

Then we convert the text in the text box into a float as follows. First we
convert the string into a Float object:

Float flt = new Float(TempEntry.getText());

Then we obtain the float numerical value from the Float object using that
class’s floatValue() method.

temp = flt.floatValue();

Or we can do the whole thing in one single statement:

temp = new Float(TempEntry.getText()).floatValue();

So our entire click method is as follows:

public void clickedCompute() {
    float temp, newtemp;
    temp = new Float(TempEntry.getText()).floatValue();
    if (Fahr.getState())
        newtemp = 9 * temp / 5 + 32;
    else
        newtemp = 5 * (temp - 32) / 9;
    result.setText(new String().valueOf(newtemp));
}

Copyright © 2000, by James W. Cooper
The computed result is converted to a String and shown in the result label text field. The running program for computing and displaying temperatures is illustrated in Figure 8-4.

![Figure 8-4: The tempcalc program running in the appletviewer.](image)

The listing is given below:

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

public class tempcalc extends Applet implements ActionListener
{
    private CheckboxGroup grpTemper;       //group radio
    private Label label1;
```

Copyright © 2000, by James W. Cooper
private TextField tempEntry;        //edit box
private Checkbox fahr;              //radio button
private Checkbox celsius;           //radio button
private Label result;               //result shown here
private Button compute;             //compute when clicked
private Button quit;                //exit when clicked

//-------------------------------------------------------
public void init() {
    //initialize applet
    setLayout(new GridLayout(6,1));
    setBackground(Color.lightGray);

    label1 = new Label("Enter temperature:");
    add(label1);

    tempEntry= new TextField(7);
    add(tempEntry);

    result= new Label(" ");
    add(result);

    grpTemper = new CheckboxGroup();
    fahr = new Checkbox("to Fahrenheit",grpTemper, false);
    add(fahr);

    celsius = new Checkbox("to Celsius",grpTemper, false);
    add(celsius);

    Panel p = new Panel();
    add(p);
    p.setLayout(new GridLayout(1,2));

    compute=new Button("Compute");
    Panel lp = new Panel();
    p.add(lp);
    lp.add(compute);
    compute.addActionListener(this);

    Panel rp = new Panel();
    p.add(rp);
    quit = new Button("Quit");
    rp.add(quit);
    quit.addActionListener(this);
    super.init();
}

//-----------------------------------------------------
public void actionPerformed(ActionEvent evt) {
    Object obj = evt.getSource();
    if (obj == compute) {

        Copyright © 2000, by James W. Cooper
Applets vs. Applications

An Applet is always a visual program. It lives inside a web page and come into existence when the web browser calls its init and start methods.

By contrast, applications do not need to have visual components. If you want to display something in a window, you must provide an initialized window. The basic displayable unit is the Window, but the Frame class, which extends Window, provides borders and a title bar. It is usually the Frame which we will use to put up windows in applications.

If we add a main routine to the tempcalc class we can create a program which will run either as an application or as an applet. We need only create a frame, which is just another kind of container, add the tempcalc class to it, resize it and display it.

```java
public static void main(String arg[]) {
    //create the frame
    Frame fr = new Frame("Temperature calculation");
    fr.setLayout(new BorderLayout());
    //create the applet
    Applet app = new tempcalc();
    fr.add("Center", app); //add into the frame
    fr.setSize(364, 225); //define its size

    fr.show(); //show the frame
    app.init(); //lay out the applet
    app.start(); //start it
}
fr.pack();  //arrange applet inside frame
}

If we compile and run the tempcalc class with this **main** method included, it will run as a stand-alone window. This main method is included in the tempcalc.java program in the \chapter8 directory of your example disk. The tempcalc program running as an application is shown in Figure 8-5.

![Figure 8-5: The same tempcalc program running as an application.](image)

**Stopping Your Application from the System Close Button**

If you tried to close the application we show above by clicking in the upper right corner system close button, it will not close. This is because, by default, the **windowClosing** event is not handled by the Frame object. If you do no specifically include a Quit button as we did in the above example, you must implement the **WindowListener** interface in your application and close the window in response to the windowClosing event.

To implement this interface, you must have instances of all 7 window events, even though all but one are empty methods:

```java
public void windowActivated(WindowEvent e){}
public void windowClosed(WindowEvent e){}
public void windowDeactivated(WindowEvent e){}
```
public void windowIconified(WindowEvent e) {}  
public void windowDeiconified(WindowEvent e) {}  
public void windowOpened(WindowEvent e) {}  
public void windowClosing(WindowEvent e) {  
    System.exit(0);  
}

This is the approach used in the example TempApp.java which runs only as an application. In this case, the main method becomes simply

public static void main(String arg[]) {  
    //create the frame  
    new TempApp("Temperature calculation");  
}

Making an Application/Applet Exit

We have to admit that having a program that will run both as an applet and as an application is more a conjuring trick than a really useful approach, it does illustrate some interesting problems.

There is no way to have an Applet implement the WindowListener interface and have it exit from the Close box when run as an application, because the applet does not receive that message. Conversely, there is no direct way to have a Frame class implement the WindowListener. However, it is perfectly possible for a subclass of Frame to implement that interface and exit properly. Thus, if we create an XFrame class we have the ability to have a closable window frame.

public class XFrame extends Frame implements WindowListener {  
    // a subclass of Frame that exits on the close box  
    public XFrame(String title) {  
        super(title); // put the title in the title bar  
        addWindowListener(this); //listen for close  
        setLayout(new BorderLayout());  
    }  
    //WindowListener interface classes  
    public void windowActivated(WindowEvent e) {}  
    public void windowClosed(WindowEvent e) {}  
    public void windowDeactivated(WindowEvent e) {}  
    public void windowIconified(WindowEvent e) {}  
    public void windowDeiconified(WindowEvent e) {}  
    public void windowOpened(WindowEvent e) {}  
    public void windowClosing(WindowEvent e) {  
        System.exit(0);  
    }  
}  

Copyright © 2000, by James W. Cooper
Then we modify the `main` method of the applet class to create this new type of frame rather than an ordinary Frame.

```java
public static void main(String arg[]) {
    //create the frame
    XFrame fr = new XFrame("Temperature calculation");

    //create the applet
    Applet app = new XTempCalc();
    fr.add("Center", app); //add into the frame
    fr.setSize(364, 225); //define its size

    fr.show(); //show the frame
    app.init(); //lay out the applet
    app.start(); //start it
    fr.pack(); //arrange applet inside frame
}
```

This modified version of the program is called `XTempCalc.java` on your example disk. We’ll be using this XFrame class throughout the rest of our examples using the AWT, and an analogous JFrame class for our Swing examples.

### Subclassing the Button Control

One of the great strengths of Java is that as an object-oriented language, you can extend nearly any class to add function to it. We just saw a useful reason to extend the Frame class. Now let’s consider the Button class.

It would be convenient if we could have each button control know what operation it is to perform and then go ahead and do it. Each button could be its own actionListener, or we could vector all actions through the same interface.

In Java, if we extend our button class, it would be a separate class which would not have direct access to the `clickedCompute` method of our applet class. In order to make it available, we need to make sure it is a public method and that we pass a reference this instance of the applet class into the button class. We do this using the special reserved word `this` which refers to the current instance of a class.
Our final drawback to this approach is that we have to create a separate subclass for each separate button on the form. Here, since there are only two, this is not particularly difficult. A good way to do this is to start with an abstract button class which contains a general Execute method, and then subclass that to create the actual buttons.

```java
// an abstract button class that executes the command
public abstract class CommandButton extends Button {
    public CommandButton(String label) {
        // pass the label to the base Button class
        super(label);
    }
    abstract public void Execute();
}
```

Note that the Execute method is abstract. This just means that we haven’t filled it in yet, and that we have to make sure that each derived class contains an implementation of that method.

For the Quit button we simply create a class that exits:

```java
public QuitButton(String caption) {
    super(caption);
}
public void Execute() {
    System.exit(0);
}
```

However, for the Compute button, we have to access the fields of the form, do the computation and put the result back in the result label. We could put all this in the button class, but since we already have a computeClick method in our main program class, we can just call it. In order to call it, we have to give the button an instance of the applet class to use in the call.

Our QuitButton class thus has an Execute method and a set method to pass in an instance of the applet class:

```java
public class ComputeButton extends CommandButton {
    private subButton sbt = null;
    public ComputeButton(String caption) {
        super(caption);
    }
    // get the calling applet
```
public void setApp(subButton app) {
    sbt = app;
}
//if the applet instance is set,
//call its compute method
public void Execute() {
    if(sbt !=null) {
        sbt.clickedCompute();
    }
}

The complete program is called subButton and is in the \chapter8 directory of your example disk.

Summary
Now that we have built some real visual applications in Java we have all the fundamentals in hand. We’ve learned about applets and applications, and we’ve seen how to derive new classes from the Frame and from the Button class and have begun to write some significant Java code.

In the next chapter we’ll look at how we lay out Java controls using layout managers. Then we’ll continue to build on our knowledge by taking up images and files and chapters to come.
12. Layout Managers

While you can position controls in a Java window can be done using absolute pixel positioning, it is more common to use a layout manager class. While many programmers may be used to visual GUI builders that generate absolute pixel positioning, Java started in the Unix world where specifying a program to do your layout is more common.

Before Java came along, the most popular high level language for creating GUI applications was TCL/TK, a language system developed almost entirely by John Osterhout. Since he had to deal with any number of different Unix platforms as well as screens varying from 640 x 480 to thousands of pixels across, he developed the concept of a layout manager, which would allow you to tell it something about the relative positions of the controls and let the layout manager figure out how to achieve it.

The great advantage of layout managers is that they are both screen size and operating system independent. Further, if you resize a window controlled by a layout manager, the components will move apart in a sensible fashion. The great disadvantage is that layout managers take a good deal of programming by the user to achieve this screen independence.

Java Layout Managers

There are 5 layout manager objects provided with Java:

- FlowLayout - controls flow from left to right and onto new lines as needed.
- BorderLayout - controls can be added to North, South, East and West. The remainder goes into the Center.
- CardLayout - provided to allow you to display one of several card areas a time, like Hypercard or a tabbed container does.
- GridLayout - a layout of n rows by m columns, where you specify m and n.
• GridBagLayout - a complex layout where each object can take up several rows or columns.

For the most part, you can lay out simple screens using the Flow, Grid and Border layouts. The others are for more involved cases.

In Java applications, a Frame is the top level window which you display on the screen. It has a title bar and may have a menu. Layouts apply to the Frame where you add controls in either an applet or an application.

A Panel is a logical, but unmarked region of a screen where you can group controls. You can add one or more Panels within the frame area each of which can then have its own layout. The default layout manager for a Frame is BorderLayout and the default for a Panel is FlowLayout. Rather than trying to remember this, we will always set the layout we need.

For simple programs, you can do a fairly good job using the GridLayout to divide your window into regions and then inserting panels in the grid cells, each having a FlowLayout so that the components within a cell are spaced apart and do not touch. You can use the BorderLayout if you want some component to be along one of the edges of the window.

**The Flow Layout**

A flow layout allows you to add components to a panel in a simple, left-to-right fashion using the following constructors:

```java
public FlowLayout();

public FlowLayout(int align);

public final static int CENTER;
public final static int LEFT;
public final static int RIGHT;
public FlowLayout(int align, int hgap, int vgap);
```

When one row is full, the next component starts a new row. When you set a FlowLayout you have the option of specifying the alignment and gap between components. However, the default alignment is CENTER, and the default gap is a reasonable looking 10-20 pixels.
The FlowLayout is primarily for laying out rows of buttons, although of course you can use it for check or text boxes or any other kind of component you wish. The following code produces the layout show in Figure 9-1.

import java.awt.*;

public class flow1 extends XFrame {
    public flow1() {
        super("Flow Layout 1");
        setLayout(new FlowLayout());

        add(new Button("Button 1"));
        add(new Button("Button 2"));
        add(new Button("Button 3"));
        add(new Button("Button 4"));
        add(new Button("Button 5"));

        setSize(200,100);
        setVisible(true);
    }

    public static void main(String arg[]) {
        new flow1();
    }
}

Figure 9-1: The FlowLayout of 5 buttons showing how 2 different screen widths affect the layout automatically.

If you want the components to be aligned along the left edge, use the third version of the constructor as shown below:

public class flow2 extends XFrame {
    public flow2() {
        super("Flow Layout 2");
        setLayout(new FlowLayout(FlowLayout.LEFT,30, 30));

        add(new Button("Button 1"));
        add(new Button("Button 2"));
        add(new Button("Button 3"));

        setSize(200,100);
        setVisible(true);
    }

    public static void main(String arg[]) {
        new flow2();
    }
}
add(new Button("Button 4"));
add(new Button("Button 5"));
setSize(200,100);
setVisible(true);
}

This program is flow2.java on your example disk. It is illustrated in Figure 9-2.

![Flow Layout 2](image)

**Figure 9-2: The FlowLayout using left alignment and a 30-pixel horizontal and vertical gap.**

Flow layouts can also produce fairly reasonable looking mixed control layouts. Each row must be the same height all the way across, but different rows may have different heights to accommodate different kinds of components. For example, the following code leads to the layout shown in Figure 9-3.

```java
public class flow3 extends XFrame {
    public flow3() {
        super("Flow Layout 3");
        setBackground(Color.lightGray);
        setLayout(new FlowLayout(FlowLayout.LEFT,30, 30));
        add(new Button("Button 1"));
        add(new Button("Button 2"));
        add(new Checkbox("Display spelling errors"));
        add(new List(8, false));
        add(new Button("Button 5"));
        setSize(200,350);
        setVisible(true);
    }
    public static void main(String arg[]) {
        new flow3();
    }
}
```

Copyright © 2000, by James W. Cooper
The Grid Layout

The GridLayout manager organizes the container into equal sized cells in a specified number of rows and columns. The constructors are:

```java
public GridLayout(int rows, int cols);
public GridLayout(int rows, int cols, int hgap,
                   int vgap);
```

So for example, if we want to lay out five controls in two columns you could use the Grid Layout:

```java
public class grid1 extends XFrame {
    public grid1() {
        super("Grid Layout 1");
        setLayout(new GridLayout(3, 3));

        add(new Button("Button 1"));
        add(new Button("Button 2"));
        add(new Button("Button 3"));
        add(new Button("Button 4"));
        add(new Button("Button 5"));
    }
}
```

Figure 9-3: A FlowLayout of a mixed group of components.
setBounds(100,100,200,100);
setVisible(true);
}
public static void main(String arg[]) {
    new grid1();
}

This program, grid1.java, produces the display shown in Figure 9-4. It is clearly not particularly elegant looking to have the buttons touching each other without any space between them. However, by simply changing the GridLayout constructor to

setLayout(new GridLayout(3, 3, 30, 5));

we can have the somewhat more pleasant spacing shown in Figure 9-5.

Figure 9-4: The GridLayout using a grid of 3 x 3 with 5 buttons. No spacing between buttons was specified.

Figure 9-5: The GridLayout using a grid of 3 x 3 with 5 buttons and a spacing of x = 30 and y = 5.
The BorderLayout Manager

The BorderLayout allows you to divide the screen into 5 regions named “North,” “South,” “East,” “West,” and “Center.”

![Diagram of BorderLayout]

Normally, you use the top and bottom border areas as narrow strips to hold buttons or toolbars, and the side borders for such things as scrollbars.

As with the other managers, it has two constructors:

```java
public BorderLayout();
public BorderLayout(int hgap, int vgap);
```

When you add a component to a border layout, you specify the edge name as part of the add method:

```java
add("North", Button1);
```

Each component is given the amount of space it naturally requires: there is no requirement that the layout manager distribute the space equally. For example, the following program produces the layout shown in Figure 9-6:

```java
public class border1 extends XFrame {
    public border1() {
        super("Border Layout 1");
        setLayout(new BorderLayout(5, 5));

        add("West", new Button("Button 1"));
        add("North", new Button("Button 2"));
        add("East", new Button("Btn 3"));
        add("South", new Button("Button 4"));
        add("Center", new Button("Button 5"));

        setBounds(100,100,300,200);
        setVisible(true);
    }
    public static void main(String arg[]) {
        new border1();
    }
}
```

Copyright © 2000, by James W. Cooper
The program is called border1.java in \chapter9 of your example disk.

Figure 9-6: A frame using the BorderLayout manager. Note that the right and left edges are unequal in width.

Since we named the West button as “Button 1,” but the East button as “Btn 3,” the left and right panels are not of the same width. Programmers generally use the BorderLayout manager to set aside part of the window at one side or another, and then put a panel containing another type of layout manager in the remaining space.

For example, we could divide the screen in half, putting a list box in one part and a series of buttons in the other. Since the center always takes up “what’s left,” we will put the list box in the center and the buttons on the right side. We will add the list box to the Center part of the border layout and then create a panel to add to the East side of the border layout. The program for laying out the controls in two columns is shown below:

```java
public class border2 extends XFrame {
    public border2() {
        super("Border Layout 2");
        setLayout(new BorderLayout(15, 1));
        setBackground(Color.lightGray);
        //create right panel
        Panel rpanel = new Panel();
        add("East", rpanel);
        //put List in center
        add("Center", new List(10, false));
        //put rest in right panel
        rpanel.setLayout(new GridLayout(3, 1, 30, 25));
        rpanel.add(new Button("Swimming"));
    }
}
```

Copyright © 2000, by James W. Cooper
rpanel.add(new Button("Diving"));
rpanel.add(new Button("Beach Soccer"));
setBounds(100,100,300,200);
setVisible(true);
}

and Figure 9-7 shows the window it generates. This program is called

![Figure 9-7: Using a BorderLayout to divide the screen in half. The “East” side is further divided using a 3 x 1 GridLayout.](image)

border2.java in `chapter9` on your example disk.

Padding a Layout Using Panels

One problem with the layout as shown, is that the buttons are too fat and take up to much of the right hand screen area. It would be better if there were more space above the top button and below the bottom button, but that the top-center-bottom button spacing was about the same as it is now.

The easiest way to accomplish this is to make the GridLayout 1 x 5 instead of 1 x 3. But how will we will adjust the middle three cells? Simple. We’ll simply insert a Panel component in the top and bottom cells:

```java
public class border3 extends XFrame {
    public border3() {

    }

Copyright © 2000, by James W. Cooper
super("Border Layout 3");
setBackground(Color.lightGray);
setLayout(new BorderLayout(15, 1));

Panel rpanel = new Panel();
add("East", rpanel);
add("Center", new List(10, false));
rpanel.setLayout(new GridLayout(5, 1, 30, 5));
rpanel.add(new Panel());
rpanel.add(new Button("Swimming"));
rpanel.add(new Button("Diving"));
rpanel.add(new Button("Beach Soccer"));
rpanel.add(new Panel());
setBounds(100,100,300,200);
setVisible(true);
}

This layout is shown in Figure 9-8 and the program is called border3.java on your example disk.

![Figure 9-8: A BorderLayout divided with a 1 x 5 GridLayout on the right side. Top and bottom grid cells contain an empty Panel.](image)

Laying Out the Temperature Conversion Program

In the previous chapter, we deferred discussing how we laid out the controls on the screen.
To lay out the controls for this program, we’ll need to put the controls into 6 rows and one column:

1. A label “Enter Temperature” and an entry field..
2. A label where the result is to be displayed.
3. The “to Fahrenheit” option button.
4. The “to Celsius” option button.
5. A Compute and Quit button on a single line together.

We create the layout by the following steps.

1. Create a 6 x 1 grid.
2. Add a label.
3. Add a TextField.
4. Add the Results label.
5. Add the “to Fahrenheit” radio button to it.
6. Add the “to Celsius” radio button to it.
7. Add a panel to the sixth line and add the two pushbuttons to that.

The code for accomplishing this is shown below.

```java
public class tempcalc extends Applet
    implements ActionListener {
    private CheckboxGroup grpTemper; //group radio
    private Label label1; //title label
    private TextField tempEntry; //edit box
    private Checkbox fahr; //radio button
    private Checkbox celsius; //radio button
    private Label result; //result shown here
    private Button compute; //compute on click
    private Button quit; //exit on click
```

Copyright © 2000, by James W. Cooper
public void init() {
    //initialize applet
    setLayout(new GridLayout(6,1));
    setBackground(Color.lightGray);

    label1 = new Label("Enter temperature:");
    add(label1);

    tempEntry = new TextField(7);
    add(tempEntry);

    result = new Label(" ");
    add(result);

    //put radio buttons in group
    grpTemper = new CheckboxGroup();
    fahr = new Checkbox("to Fahrenheit",
                       grpTemper, false);
    add(fahr);

    celsius = new Checkbox("to Celsius",
                           grpTemper, false);
    add(celsius);

    //two buttons in panel with grid layout
    Panel p = new Panel();
    add(p);
    p.setLayout(new GridLayout(1,2));

    compute = new Button("Compute");
    Panel lp = new Panel();
    p.add(lp);
    lp.add(compute);
    compute.addActionListener(this);

    Panel rp = new Panel();
    p.add(rp);
    quit = new Button("Quit");
    rp.add(quit);
    quit.addActionListener(this);
    super.init();
}

The program is called TempCalc.java on your example disk, and the displayed window is shown in Figure 9-9.
This window is adequate but not all that appealing. You can improve it somewhat by putting every component inside a Panel inside each line of the grid. This gives each grid line its own flow layout, and the labels, buttons and entry field do not try to fill the entire grid line.

Since every add method will have a new Panel associated with it, you can just extend the Frame’s add method to always create a panel first:

```java
public Component add(Component comp) {
    Panel p = new Panel();
    p.add(comp);
    super.add(p);
    return comp;
}
```

The resulting display is shown in Figure 9-10 below.
The CardLayout Manager

The card layout manager is designed to allow you to create a number of components, usually panels containing controls, but show only one of these components at a time. It can provide a basis for designing a tabbed dialog or a Hypercard-like program. It is now pretty much obsolete as more sophisticated controls provided by the Swing classes supercede it.

The GridBagLayout Manager

Of the layout managers provided with Java, the GridBagLayout manager is the most complex as well as the most versatile. The underlying assumption of this layout manager is that you can have controls that span more than one row or column.

```java
public void GridBagLayout();
public void GridBagConstraints();
```

Each element that you add to a container managed by a GridBagLayout has its drawing controlled by an instance of the GridBagConstraints class. Since the values of this class are copied into the GridBagLayout along with
a reference to the control, you only need one instance of the GridBagConstraints class as well.

The GridBagConstraints class has a number of publicly accessible variables that tell the GridBagLayout class how to position that component. You can change these fields before adding each new component to the layout.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gridx</td>
<td>x grid position of the component.</td>
</tr>
<tr>
<td>Gridy</td>
<td>y grid position of the component.</td>
</tr>
<tr>
<td>Gridwidth</td>
<td>the number of x cells the component spans.</td>
</tr>
<tr>
<td>Gridheight</td>
<td>the number of y cells the component spans.</td>
</tr>
<tr>
<td>Fill</td>
<td>which dimensions should grow if the space is larger than the component: NONE, BOTH, HORIZONTAL or VERTICAL.</td>
</tr>
<tr>
<td>Ipadx</td>
<td>extra space in pixels added to the width of the component.</td>
</tr>
<tr>
<td>Ipady</td>
<td>extra space in pixels added to the height of the component.</td>
</tr>
<tr>
<td>Insets</td>
<td>margins to appear around component: part of inset class.</td>
</tr>
<tr>
<td>Anchor</td>
<td>how the component should be aligned within the grid cells: CENTER, EAST, NORTH, SOUTH, WEST, NORTHEAST, NORTHWEST, SOUTHEAST, SOUTHWEST</td>
</tr>
<tr>
<td>weightx</td>
<td>how much weight a given cell should be given relative to others if window is wider than needed. Default is 0.</td>
</tr>
<tr>
<td>weighty</td>
<td>how much weight a given cell should be given relative to others if window is higher than needed.</td>
</tr>
</tbody>
</table>
To position a component using the GridBagLayout, we have to create an instance of both the GridBagLayout and GridBagConstraints, add the component and then set the constraints for that component:

```java
    gbl = new GridBagLayout(); //create layout mgr
    gbc = new GridBagConstraints(); //and constraints
    setLayout(gbl); //set layout

    Button b1 = new Button("One"); //create button
    gbc.gridx = 4; //fourth row
    gbc.gridy = 0; //first column
    gbc.gridwidth = 2; //two cells wide
    gbc.gridheight =1; //one cell high
    add(b1); //add button into layout
    gbl.setConstraints(b1, gbc); //set constraints for b1
```

Since most controls will have just their positions set as they are added, we can put most of the above into a private method which operates on `gbl` and `gbc`.

```java
private void add_component(Component c, int x, int y, int w, int h)
{
    gbc.gridx = x; //set x and y positions
    gbc.gridy = y;
    gbc.gridwidth = w; //and sizes
    gbc.gridheight =h;
    add(c); //add component
    gbl.setConstraints(c, gbc); //set constraints
}
```

**A Simple GridBagLayout Example**

Now let's write a simple program using a list box and three buttons. The list box will occupy 3 horizontal cells and 4 vertical ones. The push buttons will be one cell each: two in row 0 and one in row 3. The complete program for laying out these buttons is:

```java
public class gblay extends XFrame
    implements ActionListener {

    public class gblay extends XFrame
        implements ActionListener {

    Button b1 = new Button("One"); //create button
```
private GridBagLayout gbl;
private GridBagConstraints gbc;
private List lb;

public gblay(String caption) {
    super(caption);
    setBackground(Color.lightGray);

    gbl = new GridBagLayout();
gbc = new GridBagConstraints();
setLayout(gbl);

    lb = new List(5,false);
    add_component(lb, 0, 0, 3, 4);

    Button b1 = new Button("One");
b1.addActionListener(this);
    add_component(b1, 4, 0, 2, 1);

    Button b2 = new Button("Two");
b2.addActionListener(this);
    add_component(b2, 6, 0, 2, 1);

    Button b3 = new Button("Three");
b3.addActionListener(this);
    add_component(b3, 5, 3, 2, 1);

    setBounds(20, 20, 250, 150);
    setVisible(true);
}

private void add_component(Component c, int x, int y, int w, int h) {
    gbc.gridx = x;
gbc.gridy = y;
gbc.gridwidth = w;
gbc.gridheight =h;
    add(c);
gbl.setConstraints(c, gbc);
}

public void actionPerformed(ActionEvent evt) {
    Button but = (Button)evt.getSource();
    lb.add(but.getLabel());
}

public static void main(String arg[]) {
    new gblay("Simple GridBagLayout");
}

Copyright © 2000, by James W. Cooper
The resulting layout is illustrated in Figure 9-12. The program is called gblay.java on your example disk.

![Simple GridBagLayout](image)

**Figure 9-12: Simple GridBagLayout of list box and three buttons.**

**Improving on Our Simple GridBagLayout**

If you stretch the window in the above gblay.java program, the buttons and list box will stay the same size and stay clustered in the center of the window. Further, the buttons seem a bit crowded. Let’s see what we can do to unclutter the layout slightly.

First, we can make the buttons a bit bigger on either side of their captions by making ipadx and ipady larger than their default, 0.

```java
gbc.ipadx = 5;        //make buttons wider
gbc.ipady = 3;        //make buttons higher
```

Then we can add a little more space between the buttons by adding a few pixels to the **insets** class. This class has four direction names and we add 4 pixels to each:

```java
gbc.insets.left =   4;
gbc.insets.right =  4;
gbc.insets.bottom = 4;
gbc.insets.top =    4;
```

Finally, let’s experiment with the **weightx** variable to see what it does. It should give that column more space as the window expands. We’ll use it on Button b2.
Button b2 = new Button(" Two ");
gbc.weightx = 1;      //this column gets more space
add_component(b2, 6, 0, 2, 1);

The complete program gblay2.java is on your example disk. It contains only
the changes we have just discussed. Figure 9-13 shows the window
displayed by this program at its normal width and Figure 9-14 shows the
same window stretched horizontally. As you can see, buttons 1 and 3 stay
near the list box, but button 2 moves outward as the window is expanded.

![Figure 9-13: GridBagLayout gblay2.java, showing the effects of
increasing ipadx and ipady and adding values for insets.](image1)

![Figure 9-14: GridBagLayout gblay2.java, stretched horizontally.
This illustrates the affect of setting weightx to 1 for button 2.](image2)
A GridBagLayout for the TempCalc Program

When we looked at the layout of the TempCalc program in Figure 9-9 that we constructed using flow and grid layouts, it look adequate but not that professional. Let’s see how we can do using the GridBagLayout manager to arrange the controls. We’ll arrange them using the grid layout illustrated in Figure 9-15.

<table>
<thead>
<tr>
<th>(blank)</th>
<th>(blank)</th>
<th>(blank)</th>
<th>(blank)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter temperature:</td>
<td>TempEntry TextField</td>
<td>Result label</td>
<td></td>
</tr>
<tr>
<td>O - To Fahrenheit</td>
<td>O - to Celsius</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(blank)</td>
<td>(blank)</td>
<td>(blank)</td>
<td>(blank)</td>
</tr>
<tr>
<td>Compute</td>
<td>Quit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9-15: Layout of cells to be used for the final layout of the TempCalc program.

The complete program is show below and is named gbTemp.java on your example disk. The window it displays is shown in Figure 9-16.

```java
public class gbTemp extends XFrame implements ActionListener {
    private GridBagLayout gbl; //layout mgr
    private GridBagConstraints gbc; //and constraints
    private Label label1; //label
    private TextField tempEntry; //entry field
    private Label result; //result label
    private CheckboxGroup cbg; //radio buttons
    private Checkbox fahr; //radio buttons
    private Checkbox celsius;
    private Button compute; //compute button
    private Button quit; //quit button

    public gbTemp(String caption) {
        super(caption);
        setBackground(Color.lightGray);
    }
}
```

Copyright © 2000, by James W. Cooper
```java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TemperatureConverter extends Frame {
    public TemperatureConverter() {
        GridBagLayout gbl = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        setLayout(gbl);
        gbc.ipadx = 5;        //make buttons wider
        gbc.ipady = 5;        //make buttons higher
        label1 = new Label("Enter temperature:");
        add_component(label1,0,1,2,1);

        tempEntry = new TextField(5);
        add_component(tempEntry,2,1,1,1);

        result = new Label(" result ");
        add_component(result,1,2,2,1);

        cbg = new CheckboxGroup();
        //left align check boxes in cell 1
        gbc.anchor = GridBagConstraints.WEST;
        fahr = new Checkbox("to Fahrenheit", cbg, true);
        add_component(fahr,1,3,2,1);

        celsius = new Checkbox("to Celsius", cbg, false);
        add_component(celsius,1,4,2,1);

        //return to center alignment
        gbc.anchor = GridBagConstraints.CENTER;
        add_component(new Label(""),1,5,1,1); //leave space
        compute = new Button("Compute");
        compute.addActionListener (this);
        add_component(compute, 0,6,1,1);

        quit = new Button("  Quit   ");
        quit.addActionListener (this);
        add_component(quit, 2,6,1,1);

        setBounds(20, 20, 250, 300);
        setVisible(true);
    }

    private void add_component(Component c, int x, int y, int w, int h) {
        gbc.gridx = x;
        gbc.gridy = y;
        gbc.gridwidth = w;
        gbc.gridheight =h;
        add(c);
        gbl.setConstraints(c, gbc);
    }

    private void weight_cell(Component c, int weightx, int weighty) {
        gbc.gridwidth = weightx;
        gbc.gridheight = weighty;
        add(c);
        gbl.setConstraints(c, gbc);
    }
}
```
int weighty) {
    gbc.weightx = weightx;
    gbc.weighty = weighty;
    gbl.setConstraints(c, gbc);
}

//-------------------------------
public void actionPerformed(ActionEvent evt) {
    Object obj = evt.getSource();
    if (obj == compute) {
        clickedCompute();
    }

    if (obj == quit) {
        System.exit(0);
    }
}

//-------------------------------
private void clickedCompute() {
    float temp, newtemp;

    temp = new Float(tempEntry.getText()).floatValue();
    if (fahr.getState())
        newtemp = 9 * temp / 5 + 32;
    else
        newtemp = 5 * (temp - 32) / 9;
    result.setText(" " +
        new String().valueOf(newtemp)+" ");
}

//-------------------------------
public static void main(String arg[]) {
    new gbTemp("GridBag Temperature");
}

Figure 9-16: The TempCalc program as laid out using the GridBagLayout manager.

Summary

In this chapter, we’ve looked at all the ways Java provides for laying out components in windows. We’ve looked at the FlowLayout, the GridLayout, the BorderLayout, the CardLayout and the venerable GridBagLayout. We’ve seen how you can use the GridBagConstraints class to position almost anything anywhere. In the next chapter we’re going to talk about graphics and then about how Java uses files.
13. Writing a Program With Two Windows

We often need to write programs that contain a number of windows which need to communicate data entered between them. You can do this in Java, but in an even more rigorously object-oriented fashion than in other languages.

A Temperature Plotting Program

In Chapter 8, we wrote a simple program for converting temperatures between Fahrenheit and Celsius, but with no memory of past data points. In this chapter, we’ll write a program which keeps an array of data points and then allows you to display a plot of them on the screen.

While an array would be a likely candidate to store a list of data points, one drawback is that we won’t know ahead of time how many data points the user might enter. Rather than reserving some large, unnecessary amount of memory, we have a somewhat simpler way of doing the same thing, using the Vector class. An instance of a Vector class can hold any number of objects (although not primitive data types) and expands as needed. While there is no requirement that the objects all be of the same type, if you intermingle objects it is up to you to make sure you obtain objects of the correct type, or a run-time error will occur. The major Vector methods are

Vector v = new Vector(); //pick default size
Vector v = new Vector(n); //estimate size
v.addElement(obj); //any object type
obj = v.elementAt(i); //return an element
v.removeElement(i); //remove an element
v.insertElementAt(obj,i) //insert element
n = v.size(); //return number in use

Since Vectors always return objects of type Object, you must cast them to the correct type of object on return:

v.addElement(Float(x)); //add Float object
Float Fx = (Float)v.elementAt(I); //retrieve one

Copyright © 2000, by James W. Cooper
In this plotting program, we will accumulate the data to be plotted in a Vector and then pass a reference to the entire vector to the plotting window when we want to plot it. Our main entry screen is shown in Figure 11-1. The complete program is on your example disk as TempPlot.java in the \chapter11 directory.

![Temperature plotter](image)

**Figure 11-1:** The TempPlot entry screen.

The program screen is laid out using the Border and Grid layout managers as follows:

```java
public class TempPlot extends XFrame implements ActionListener {
    private Vector plotPoints;       //data stored here
    private TextField numEdit;
    private Label label1;
    private Label counter;
    private Button btStore;
    private Button btPlot;
    //---
    public TempPlot(String caption) {
        super(caption);
        setBackground(Color.lightGray);
        //layout using border layout for top line
        //and Grid Layout for panel inserted in center
        setLayout(new BorderLayout());
        label1 = new Label("Enter temperature");
        add("North", label1);
        Panel p0 = new Panel(); //put panel in center
        add("Center", p0);
        //create 2 x 2 Grid with 20 pixel gaps
        p0.setLayout(new GridLayout(1,3, 20, 20));
    }
}
```

Copyright © 2000, by James W. Cooper
Note that if we want to set the focus to the text entry field `numEdit` using the `requestFocus` method, we must do this after the window’s `setVisible` method has been executed.

We can enter any number of data points by typing them in and clicking on the Store button. Each time we store data, we clear the text box and place the cursor back there for the next data point.

```java
public void clickedStore() {   //create Float object from fp data
    Float temp = new Float(numEdit.getText());
    plotPoints.addElement(temp);
    //change label to show new number of points
    counter.setText(new String().valueOf(plotPoints.size()) + " points");
}
```
numEdit.setText(" "); //clear entry field
numEdit.requestFocus(); //reset focus here

It would be convenient to be able to enter data using the Enter key rather than having to click on the Store button each time. In Visual Basic, we would have done this by making the Store button the Default button. In Java, we can do something analogous, because pressing the Enter key generates an **actionPerformed** event. Then we need only test for where it came from and call the clickedStore procedure:

```java
public void actionPerformed(ActionEvent evt) {
    Object obj = evt.getSource();
    if (obj == btPlot)
        clickedPlot();  //plot button starts plot
    else
        clickedStore();  //store button adds point
}
```

Then when we have accumulated enough data points, we click on the Plot button and it creates an instance of the Plotit class.

```java
public void clickedPlot() {
    //create instance of plot form
    Plotit pl = new Plotit(plotPoints);
    //and display it
    pl.setVisible(true);
}
```

The Plotit display is shown in Figure 11-2.

![Plot of data points](image)

Figure 11-2: A plot of the data entered using the TempPlot class.
The Plotit Display

This window contains a Close button, four label fields and a Panel control that we have subclassed to plot x-y data. When we create an instance of the Plotit class we pass it a reference to the vector of x,y points. Then, when we define the size of the panel using the reshape method, the plotPanel extends this method to calculate the x- and y-scaling factors. Once the class has determined the maximum and minimum y-values as part of this scaling process, we can use its public Ymax and Ymin methods to obtain the values we will put in the ymin and ymax axis labels. We can determine the x axis labels from the number of points in the vector. The Plotit class constructor is shown below:

The PlotPanel class does most of the work in this display. It receives a copy of the vector of plot points in its constructor.

```java
public class PlotPanel extends Panel {
    //this panel class plots an array within its boundaries
    //and draws an outline in black
    private Vector plotpnts;
    private float xmin, xmax, ymin, ymax;
    private int pwidth, pheight;
    private int pleft, ptop;
    private float xscale, yscale;
    //-------------------------------------------------------
    public PlotPanel (Vector pl) {
        plotpnts = pl; //save vector inside object
    }
}
```

During its setBounds method, it calculates the scale for the plot based on the size of the window. The method calculates the maximum and minimum y values and from the number of y-pixels computes the y scale. Since (0,0) is in the upper left hand corner in Java graphics, the yscale is computed as a negative, and during plotting this value is added to the height to determine the absolute y-pixel position.

```java
//-------------------------------------------------------
    public void setBounds(int xpos, int ypos,
                          int width, int height) {
        Dimension sz;
        setBackground(Color.white);
        xmin = 1;
        xmax = plotpnts.size();
        ymin = 1e35f;
    }
```

Copyright © 2000, by James W. Cooper
ymax = -1e35f;
//find max and min to scale plot
for (int i = 0; i < plotpnts.size(); i++) {
    float y = ((Float)
        plotpnts.elementAt(i)).floatValue();
    if (y > ymax)
        ymax = y;
    if (y < ymin)
        ymin = y;
}
//compute scale for plots
xscale = width/((xmax-1));
yscale = height/(-(ymax - ymin));
super.setBounds(xpos, ypos, width, height);

//-------------------------------------------------
public float getYmin() {
    return ymin;
}
//-------------------------------------------------
public float getYmax() {
    return ymax;
}

The paint routine is where the plotting actually takes place. The plotPanel determines its size and draws a border around that region. Then it computes the x and y pixel position of each data point and draws a line from the last point to it.

//-------------------------------------------------
public void paint(Graphics g) {
    float x, y;
    int xpix, ypix, xold=0, yold=0;
    //draw black rectangle for border
    g.setColor(Color.black);
    Dimension sz = getSize(); //get window size
    g.drawRect(0,0,sz.width-1,sz.height-1);
    //plot data in blue
    g.setColor(Color.blue);
    for (int i=0; i< plotpnts.size(); i++) {
        x = i + 1;
        //get each floating value from Vector
        y = ((Float)
            plotpnts.elementAt(i)).floatValue();
        //convert to pixel values
        xpix = (int)((x -xmin) * xscale ) ;
        ypix = (int)((y- ymin) * yscale ) +
            sz.height ;
        if (i>0)
g.drawLine(xold, yold, xpix, ypix);
xold = xpix;  // save for next line
yold = ypix;
}

Summary
In this brief chapter, we’ve tackled a real world problem: entering an array of data and plotting on the screen using Java. We saw most significantly, how we could use a public method to pass a reference to the entire vector to the plotting window. In addition, we saw how we could subclass Panel to have it plot x-y data and how subclassing the reshape method allowed us to compute the plotting scale.

In the next chapter we’ll take up how to read and write files and make use of exceptions to handle I/O error conditions.
14. Files and Exceptions

Java applications have the ability to handle files in a large number of ways that make them as powerful as applications written in any other language. By contrast, Java applets, of course, cannot read or write files on any of your computer’s disk drives. They do have the ability to read files from the web server machine as we will see later.

Java has a large number of classes in the java.io package which can be useful in reading and writing files. The complete list is shown below, but we will be discussing only a few of them in this chapter. You can read about the capabilities of each of these classes in the reference documentation.

BufferedInputStream
BufferedOutputStream
BufferedReader
ByteArrayInputStream
ByteArrayOutputStream
DataInputStream
DataOutputStream
File
FileDescriptor
FileInputStream
FileOutputStream
FileReader
FilterInputStream
FilterOutputStream
InputStream
LineNumberInputStream
OutputStream
PipedInputStream
PipedOutputStream
PrintStream
PushbackInputStream
RandomAccessFile
SequenceInputStream
StreamTokenizer
StringBufferInputStream

The File Class

The File class provides you with a convenient system independent way to manipulate files and directories. It has the two constructors:

public void File(String filename);

Copyright © 2000, by James W. Cooper
public void File(String path, String filename);

as well as one using the File class itself:

public void File(File dir, String name);

The methods in the file class provide most of the convenient ways of checking for a file’s existence, finding its directory, creating directories, renaming, finding its length and listing the contents of the directory.

However, the File class has no open, read or write methods. In fact, all of the classes which do actual input or output make use of the File class as an underlying object. Before we can actually open a file and write to it, we need to understand how Exceptions work.

Exceptions

An Exception is a class of objects which represent various kinds of fatal and near-fatal errors that require special handing in your program. When you perform an illegal operation, such as trying to read a file that doesn’t exist or trying to write a file to a write-protected device or volume, Java throws an exception.

Most operations surrounding file handling throw exceptions and since the fact that they throw exceptions is included in the method definition, the compiler issues errors if you do not test for exceptions when using these methods. For example, the RandomAccessFile constructor is declared like this:

```java
public RandomAccessFile(String name, String mode)
    throws IOException
```

To catch such exceptions, you must enclose them in a `try block`:

```java
RandomAccessFile f = null;
try {
    f = new RandomAccessFile("people.add", "r");
} catch (IOException e) {
    System.out.println("no file found");
} finally {
```

Copyright © 2000, by James W. Cooper
Try blocks may be followed by zero or more `catch` blocks, each catching a
different exception, and may be further followed by one `finally` block.

The program proceeds as follows. The statements within the try block are
executed sequentially and if no error occurs, the program exits from the try
block and executes any statements in the finally block. If an error occurs, the
Java system looks for the first catch block whose arguments match or are
derived from the class of exception specified in the argument list. It executes
that catch block and then, if a finally block exists, goes on to execute that
code as well.

Note the unusual syntax of the catch block:

```java
catch (IOException e) { statements; }
```

The exception `e` is, of course, an object, with methods you can use to obtain
more information. The two most useful methods are:

```java
String e.getMessage();  // obtain a descriptive message
e.printStackTrace();   // print stack trace to output
```

### Kinds of Exceptions

Java I/O classes throw exceptions for most of the kinds of errors you might
expect for files: file not found, read errors and write errors:

- `EOFException`
- `FileNotFoundException`
- `IOException`
- `InterruptedIOException`
- `UTFDataFormatException`

Another class of exceptions are thrown by the runtime system, but the
compiler does not require you to catch them, since they are usually fatal to
the program in any case:

- `ArithmeticException`
- `ArrayIndexOutOfBoundsException`
- `ArrayStoreException`
ClassCastException
ClassNotFoundException
CloneNotSupportedException
IllegalAccessException
IllegalArgumentException
IllegalMonitorStateException
IllegalThreadStateException
IndexOutOfBoundsException
InstantiationException
InterruptedException
NegativeArraySizeException
NoSuchMethodException
NullPointerException
NumberFormatException
RuntimeException
SecurityException
StringIndexOutOfBoundsException

However, you can wrap any sequence of statements you like in a try block and catch any of these exceptions as well, if you want your program to deal with them.

Reading Text Files

Now that we’ve discussed exceptions, let’s see how we use them in reading files. We’ll start by considering the RandomAccessFile class. Now this class is quite powerful and versatile and allows you to read and write data from and to any position in a file. It allows you to read binary integers and floating point numbers as well as text. But most useful to us is the method:

```java
public final String readLine();
```

which allows us to read lines of text from a text file. This method returns the entire line, including the newline character, and in some cases you may want to remove this while processing the file. The `readLine` method does not throw an EOFException if you try to read past the end of the file, since you will almost always read past the end, not knowing the file length in advance. Instead, it returns a string value of `null` when no more lines can be read.

Let’s assume that we have a series of comma-separated lines of text representing a series of customers:

Fred, Fump, 437 Old Farkle Way, Neenah, WI, 65432, 517-443-5555
Janey, Antwerp, 22 Dilly Drive, Darlington, MO, 66789, 217-445-6789
Franz, Bloviate, 1456-23 Bafflegab Pl, Bilgewater, CT, 06401, 203-222-3333

Copyright © 2000, by James W. Cooper
We can read these in as lines of text and parse them later using the following program:

```java
public class Ftest {
    public Ftest() {
        String s = "";        //initialize s
        RandomAccessFile f = null;     //and file
        try {
            //open file
            f = new RandomAccessFile("people.add", "r");
        }
        catch (IOException e) {
            System.out.println("no file found");
        }
        do {
            try {
                s = f.readLine();
            }
            catch (IOException e) {
                System.out.println("File read error");
            }
            if (s != null)
                System.out.println(s);
        }
        while (s != null);
    }

    //---------------------------------------------------------
    public static void main(String argv[]) {
        new Ftest();
    }
}
```

Note that both the instantiation of the file class and the readLine method require that you surround them with try blocks. However, you don’t ever have to worry about which lines to surround with try blocks: the compiler will tell you if they are required.

However, we see here some unusual code initializing both the string and the file object to some value before we enter the try block:

```java
String s = "";        //initialize string
RandomAccessFile f = null;     //and file
```

Copyright © 2000, by James W. Cooper
Why do we do this? If we simple declared the variables without initializing them:

```java
String s;
RandomAccessFile f;
try {
    f = new RandomAccessFile("people.add", "r");
}
try {
    s = f.readLine();
}
catch (IOException e)
    {System.out.println("File read error");}
System.out.println(s);
```

the compiler would detect that the statements inside the try block may not be executed and thus that `f` and `s` had not been initialized.

```
ftest.java(23): Variable f may not have been initialized
ftest.java(27): Variable s may not have been initialized
2 error(s), 0 warning(s)
```

Then the statements:

```java
s = f.readLine();
```

and

```java
System.out.println(s);
```

would thus be illegal since neither `s` nor `f` have been initialized to any value. Therefore by simply initializing both to any value, including `null`, you can avoid this compiler error message.

This program is called `ftest.java` and is in the `\Files` directory of your example disk.

**Multiple Tests in a Single Try Block**

Obviously, the above code is somewhat confusing and cluttered to read. However, there is no reason why you can’t enclose all of the statements that
might throw exceptions in a single try block and make the program easier to code and easier to read. We revise this program to do this in the program Ftest2.java:

```java
public class Ftest2 {
    public Ftest2() {
        String s = "";
        RandomAccessFile f = null;
        try {
            f = new RandomAccessFile("people.add", "r");
            do {
                s = f.readLine();
                if (s != null)
                    System.out.println(s);
            }
            while (s != null);
        }
        catch (IOException e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
        }
    }
    public static void main(String argv[]) {
        new Ftest2();
    }
}
```

### Catching More Than One Exception

Since we have enclosed more than one method that can throw an exception in our try block, we might expect to need to catch more than one type of exception. It turns out that most of the exceptions you will need to catch are of the class IOException, so this doesn’t happen that often. However, let’s modify the program so that it generates a null pointer exception.

As you may recall, the readLine method RandomAcessFile class does not throw an EOFException, but instead simply returns a null for any string once the end of file is reached. Now the output routine System.out.println() handles virtually all data types correctly and converts the reference null to the string “null.” This does not cause an error, then.

But suppose we modify the loop to print out the length of each line as well as its contents:

```java
String s = "";
RandomAccessFile f = null;
```
do
{
    s = f.readLine();
    System.out.println(s.length() + " " + s);
}
while (s != null);

Then, when the f.readLine() method returns a null, the s.length() method will fail on a null pointer. We can trap this by catching both kinds of exceptions in a loop as follows:

```java
public class Ftest3 {
  public Ftest3() {
    String s = "";
    RandomAccessFile f = null;
    try {
      f = new RandomAccessFile("people.add", "r");
      do {
        s = f.readLine();
        System.out.println(s.length() + " "
          + s.trim());
      } while (s != null);
    }
    catch (IOException e) {
      System.out.println(e.getMessage());
      e.printStackTrace();
    }
    catch (NullPointerException e) {
      System.out.println("exception: " +
        e.getMessage());
      e.printStackTrace();
    }
  }
  public static void main(String argv[]) {
    new Ftest3();
  }
}
```

This final version of the program is called Ftest3.java and when run, it prints out the error:

```
exception: null
java.lang.NullPointerException
```

Copyright © 2000, by James W. Cooper
Building an InputFile Class

Now even though the RandomAccessFile class is fairly powerful, it still doesn’t give us the flexibility we’d like. It is fairly simple, however, to encapsulate all of the functions of this class inside one with the usual file reading capabilities. We also find that we can build a more efficient file reader by creating a BufferedReader which takes a FileReader as its constructor argument. This is just complicated enough that we might want to create a class to handle it.

Let’s first try to simplify the number of exceptions we have to catch. We might at first think that burying these exceptions and setting an error flag would be more efficient. Let’s try it and see. We’ll throw an exception if the file cannot be opened, but set an error flag on a read failure. This version of our InputFile class is

```java
import java.io.*;
//Input file class that encapsulates the FileReader
//and throws exceptions
public class InputFile {
    private BufferedReader f = null;
    private boolean errFlag;
    private String s = null;
    private String fileName;

    public InputFile(String fname) throws IOException {
        errFlag = false;
        //open file
        f = new BufferedReader(new FileReader(fname));
        fileName = fname;
    }

    public String readLine() {
        //read in a line from the file
        s = null;
        try {
            s = f.readLine(); //could throw error
        } catch (IOException e) {
            errFlag = true;
            System.out.println("File read error");
        }
        return s;
    }

    public boolean checkErr() {
        return errFlag;
    }
}
```

Copyright © 2000, by James W. Cooper
public void close() {
    try {
        f.close();     //close file
    } catch (IOException e) {
        System.out.println("File close error");
        errFlag = true;
    }
}
}

and our corresponding file reading program is

import java.io.*;
public class Input1 {
    public Input1() {
        String s = ";"
        InputFile f = null;
        try {
            f = new InputFile("people.add");
        }
        catch (IOException e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
        }
        s = f.readLine ();  //read a line
        while (s != null) {
            if (! f.checkErr ())   //check error
                System.out.println(s);
            s = f.readLine();   //Read a line
        }
    }
    public static void main(String argv[]) {
        new Input1();
    }
}

Note that we still have to catch the exception on file opening and we also have to check the errors after reading each line. This isn’t such a terrific improvement.

Instead, we’ll write another version of the InputFile class which throws all the exceptions. It is short and easy to read:

public class InputFile {
    private BufferedReader f = null;
    private String s = null;
}
public InputFile(String fname) throws IOException {
    //open file
    f = new BufferedReader(new FileReader(fname));
}

public String readLine() throws IOException {
    //read in a line from the file
    s = null;
    s = f.readLine(); //could throw error
    return s;
}

public void close() throws IOException {
    f.close(); //close file
}
}

and our file reading main program is equally simple with only one exception to catch.

public class Input2 {
    public Input2() {
        String s = "";
        InputFile f = null;
        try {
            f = new InputFile("people.add");
            s = f.readLine();
            while (s != null) {
                System.out.println(s);
                s = f.readLine();
            }
            f.close();
        }
        catch (IOException e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
        }
    }
    public static void main(String argv[]) {
        new Input2();
    }
}

If we want to read in data a field at a time, where the fields on a single line are separated by commas, we simple read in a line at a time as before, and then search for commas, returning the text between commas until none remain, and then returning the remainder of the line:
private void readNewLine() {
    try {
        line = readLine();
        if(line != null)
            tok = new StringTokenizer(line, ",");
    } catch (IOException e) {
        s = null; //none left
    }
}

public String read() {
    //reads comma-separated tokens
    //returns null when done
    if (line == null)  {
        readNewLine();
    }
    if(! tok.hasMoreTokens ()) {
        readNewLine();
    }
    if(tok.hasMoreTokens ())
        s = tok.nextToken ();
    else
        s= null;
    return s;
}

Now we can rewrite our calling program only slightly, and we can print out
the fields in the file one by one:

public class InputToken {
    public InputToken() {
        String s = "";
        InputFile f = null;
        try {
            f = new InputFile("people.add");
            s = f.read ();
            while (s != null) {
                System.out.println(s);
                s = f.read();
            }
            f.close();
        } catch (IOException e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
        }
    }
    public static void main(String argv[]) {

Building an OutputFile Class

Just as we built an InputFile class to provide simple line input functionality, we can build an OutputFile class which allows us to print numeric and string data to a file. We will include the ability to space data apart in columns. Since, many word processors prefer tab-separated columns when you want to import data into tables, we will put in the ability to write an actual tab character to a file.

Similarly to the InputFile class, we can extend a buffer file class to do output file writing. The PrintWriter class, however, contains print methods for all of the standard data types and it is quite simple to enclose this class inside a new object which will have all of these output methods.

However, if you look at the constructors for the PrintWriter class, you discover that it requires an instance of the FileWriter class:

```java
public PrintWriter(FileWriter  out);
```

Thus, the constructor for our OutputFile class will thus create an instance of the BufferedWriter class which will be used to create an instance of the FileWriter class which we will use within the class whenever one of our methods generates output:

```java
public class OutputFile {
    private BufferedWriter f;
    private PrintWriter p;
    private boolean errflag;
    private int tabcolumn;
    private int width;
    //---------------
    public OutputFile(String filename)
        throws IOException {
        errflag = false;
        tabcolumn = 0;
        width = 0;
        f = new BufferedWriter(
            new FileWriter(filename));
        p = new PrintWriter(f);
    }
}
```
PrintStream Methods

The methods shown below are the PrintWriter methods we will be wrapping. Each delivers data to the output stream.

```java
public void flush();
public void print(boolean b);
public void print(char c);
public void print(char s[]);
public void print(double d);
public void print(float f);
public void print(int i);
public void print(long l);
public void print(Object obj);
public void print(String s);
public void println();
public void println(boolean b);
public void println(char c);
public void println(char s[]);
public void println(double d);
public void println(float f);
public void println(int i);
public void println(long l);
public void println(Object obj);
public void println(String s);
```

Again, note that we will have to explicitly name all the methods we wish to use in our OutputFile class since we aren’t extending PrintWriter but wrapping another class around it. However, this is not an onerous task, because we need to make our own versions of each of these methods (effectively extending each of them) so we can count the number of characters we’ve printed in order to keep our columns lined up.

In this class, we will convert each kind of variable to a string directly and then use our own print method for strings to keep count of the number of characters to the next tab column:

```java
public void print(String s) {
    p.print(s);
    tabcolumn += s.length();
}
```

Then, when we need to print spaces up to a specified column, we know how many there need to be:

```java
public void tab(int tb) {
    if (tb > tabcolumn) {
```
The space method returns a string containing the number of spaces you specify. In Java, the String class produces constant strings: you can’t change them after they are directed. However, you can create strings whose contents you can change using the StringBuffer class which has an insert method. Our space method is:

public String space(int n) {
    StringBuffer sb = new StringBuffer(n);
    //put spaces into string buffer
    for (int i=0; i < n; i++) {
        sb.insert(i, ' ');
    }
    return sb.toString();
}

The program outclass.java in the \Files directory of your example disk prints a few lines to a file. It is merely:

public class Outclass {
    OutputFile outf;
    public Outclass() {
        try {
            System.out.println("creating file");
            outf = new OutputFile("foo.txt");
        }
        catch (IOException e) {
            System.out.println (e.getMessage ());
        }
        System.out.println("writing to file");
        outf.print("fooie");
        outf.tab(12);
        outf.println(123.45);
        outf.close();
    }
    //-----------------------------
    public static void main(String argv[]) {
        new Outclass();
    }
}

The actual OutputFile class is in the separate file OutputFile.java in the same directory. Since it is in your classpath and referred to by the outclass program, it is automatically compiled when you compile outclass and loaded
when you run the java interpreter. The complete OutFile class is shown below:

```java
import java.io.*;
//Output file class
public class OutputFile {
    private BufferedWriter f;
    private PrintWriter p;
    private boolean errflag;
    private int tabcolumn;
    private int width;
    public OutputFile(String filename) throws IOException {
        errflag = false;
        tabcolumn = 0;
        width = 0;
        f = new BufferedWriter(new FileWriter(filename));
        p = new PrintWriter(f);
    }
    public void tab() {
        p.print("\t");
    }
    public String space(int n) {
        StringBuffer sb = new StringBuffer(n);
        for (int i=0; i < n; i++) {
            sb.insert(i, ' ');
        }
        return sb.toString();
    }
    public void tab(int tb) {
        if (tb > tabcolumn) {
            print(space(tb - tabcolumn));
        } else
            println("\n");
    }
    public void println(String s) {
        p.println(s);
        tabcolumn = 0;
    }
    public void println(int i) {
        p.println(i);
        tabcolumn = 0;
    }
}
```

Copyright © 2000, by James W. Cooper
public void println(double d) {
    tabcolumn = 0;
    p.println(d);
}

public void print(String s) {
    p.print(s);
    tabcolumn += s.length();
}

public void print(int i) {
    String s=new Integer(i).toString();
    if (s.length() < width)
        print((space(width-s.length()) ) );
    print(s);
}

public void print(float f) {
    String s=new Float(f).toString();
    print(s);
}

public void print(double d) {
    String s=new Double(d).toString();
    print(s);
}

public void close() {
    p.close();
}

public void finalize() {
    close();
}

Using the File Class

The File class allows us to manipulate files and directories on any of your disk drives. Using the File class, you can move through directories, create new directories and delete and rename files. While you can determine the “current directory” that the program was launched from, Java does not have the concept of changing to a new current directory. You can move to new directories and determine their absolute path to use in reading or writing files.
File Methods

While some operating systems allow you to use either the “\” or the “/" character for separating directory and subdirectory names, others are less forgiving. Fortunately the separator string for your system is stored in the static string File.separator and you can use it completely independent of the actual platform your program is running on.

String fl = f.getAbsolutePath() + File.separator + st;

Probably the most useful method in the File class is the exists() method. You can create a file object and find out if it exists without causing an error or exception:

File f = new File(“nail.txt”);
if (f.exists())
    System.out.println(“The file exists”);

You can also delete or rename such a file just as easily:

if (f.exists())
    {  
f.renameTo(“file.txt”); //either rename
f.delete();           //or delete
}

Copyright © 2000, by James W. Cooper
Moving Through Directories

You must start by finding what directory your program is running in. This information is available as ‘user.dir,’ one of 15 named system properties that Java saves when the interpreter starts.

```java
String thisdir = System.getProperty("user.dir");
```

Once you have the name of any directory, you can obtain either

- a list of files (and subdirectories) in that directory, or
- the name of the parent directory.

The list of files in that directory is available using the File object’s `list()` method. You can examine each name and determine whether it is a file or a subdirectory using the `isDirectory()` method. The `isFile()` method is not an exact complement. System files and hidden files will return `false` from the `isFile()` method.

If you want to open a new directory, just create a file object having that path and reload the list boxes using the `list()` method. Figure 12-1 shows a simple, visual program for exploring files and directories. The code for this program is shown below and is on your example disk as Filedir.java.

```java
import java.awt.*;
import java.awt.event.*;
import java.io.*;
//displays a file and directory list box
//and an "up directory" button
//clicking on a directory entry causes the
//program to open that directory
public class Filedir extends XFrame implements ActionListener, ItemListener {
    List fDir;       //directory list
    List fList;      //filename list
    Button upDir;    //up directory button
    File f;
    String[] filelist;   //list of files
    //-------------------------------------------
    public Filedir() {
        super("File and directory list");   //title
        setBackground(Color.lightGray);
        setLayout(new BorderLayout());
        upDir = new Button("Up Dir");   //button
```
```java
upDir.addActionListener(this);
Panel p0 = new Panel();
add("North", p0);
p0.add(upDir); //panel for button
Panel p1 = new Panel(); //panel for lists
add("Center", p1);

fDir = new List(10, false);
fDir.setBackground(Color.white);
p1.add(fDir);
fList = new List(10, false);
fList.setBackground(Color.white);
fDir.addItemListener(this);
p1.add(fList);

// start with current directory
loadlists(System.getProperty("user.dir"));
setBounds(100, 100, 400, 300);
setVisible(true);
}

public void actionPerformed (ActionEvent evt) {
// if button is clicked move up one directory
String p = f.getParent();
if (p == null)
p = f.pathSeparator;
loadlists(p); // and reload list boxes
}

private void loadlists(String p) {
f = new File(p); // create new obj each time
setTitle(f.getAbsolutePath());
filelist = f.list();
// clear out lists
fDir.removeAll();
fList.removeAll();

for (int i = 0; i < filelist.length; i++) {
File file = new File(f.getAbsolutePath() + "\" +
    filelist[i]);
    // put in one list or the other
    if (file.isDirectory())
fDir.add(filelist[i]);
else
fList.add(filelist[i]);
}
}

public void itemStateChanged(ItemEvent evt) {
// open directory that was clicked on
```
String st = fDir.getSelectedItem();
loadlists(f.getAbsolutePath() +
    File.separator + st);
}
 public static void main(String argv[]) {
    new Filedir();
}

Figure 12-1: The display of files and directories from the
filedir.java program.

The FilenameFilter Interface

The FilenameFilter is an interface that you can use to create a class to filter
filenames, returning only those of interest to your application. You can write
any kind of class you want as long as it implements the FilenameFilter
interface. This interface amounts to only one method:

    public boolean accept(File dir, String name)
You write any sort of method you want to examine the directory and filename and return either true or false. Then you tell the list method of the File class the name of the class you created. It passes every filename through the accept method of your class and receives either true or false back.

To use a FilenameFilter, we create a class which implements that interface:

```java
public class FileFilter implements FilenameFilter {
    Vector extnlist;
    public FileFilter(String s) {
        extnlist = new Vector();
        extnlist.addElement(s);
    }
    //-------------------------------
    public boolean accept(File dir, String name) {
        boolean found = false;
        for (int i = 0; i < extnlist.size(); i++) {
            found = found ||
                    (name.endsWith((String)
                        extnlist.elementAt(i)));
            found = found ||
                    new File(dir +
                        File.separator+name).isDirectory();
        }
        return found;
    }
    //-------------------------------
    public void addExtension(String s) {
        extnlist.addElement(s);
    }
}
```

Our FileFilter class has a constructor which saves one type of file extension. We also include an addExtension method to add additional ones. Then, in the accept method we filter each filename against a list of possibilities. Here we allow any file extension in the vector extnlist and also allow any name which is a directory.

We modify our calling program to make use of this filter as follows:

```java
private void loadlists(String p) {
    FileFilter filter;
    filter = new FileFilter("class");
    filter.addExtension("java");
}
```
f = new File(p);     //create new obj each time
setTitle(f.getAbsolutePath());
//create filtered list
filelist = f.list(filter);     //clear out lists
fDir.removeAll ();
fList.removeAll();

The remainder of the program is the same as in the previous example. The complete program is filedir2.java on your example disk.

File Selection

The above programs work on a single disk drive. Java does not currently provide a way to obtain the names of other drives on your computer or attached over the network. If you need to move between drives, however, Java provides a method to call the operating system’s file dialog box. The FileDialog object allows you to bring up a File/Open or a File/Save As dialog using the following simple constructor:

```java
public FileDialog(Frame  parent, String  title,
        int  mode);
```

where `mode` can be either FileDialog.LOAD or FileDialog.SAVE. You simple create the object and execute it’s show method. The file dialog is shown as a modal dialog, meaning that no other windows can get the focus until it is closed.

```java
fdlg = new FileDialog(this, "Open a File", FileDialog.LOAD);
fdlg.show();
System.out.println(fdlg.getDirectory());
System.out.println(fdlg.getFile());
```

The `getDirectory` and `getFile` methods return the directory and filename you selected. The FileDialog is supposed to support use of a FilenameFilter, but it has not been implemented in Java 1.0 in either Windows or in OS/2. If you bring up the file dialog in Save mode, and select an existing filename, it asks if you want to replace that file and only exits if you answer by clicking on Yes. The program filedlg.java on your example disk illustrates the use of the file dialog. This program is illustrated in Figures 12-2 and 12-3.
Summary

We’ve covered quite a number of ways to use files in Java in this chapter. We created an InputFile and OutputFile class to make their use easier, basing them on the RandomAccessFile and PrintStream classes, respectively. We also learned how to navigate through directories in Java and how to use the FilenameFilter class. Finally, we learned how to bring up the standard File/Open and File/Save dialogs.

Copyright © 2000, by James W. Cooper
Next we’re going to take up displaying images and after that using threads in Java. Then we’ll look into some more elaborate applications and applets.
15. Using Threads in Java

One of the most powerful features of Java is its ability to execute several threads within the same program. When you think of a multitasking system like Windows or Unix, you recognize that you can start several programs at the same time and have them all appear to be running “at once.” Of course, what is really happening is that each of the programs takes a turn receiving a fraction or *slice* of the available time from the underlying operating system’s scheduler. Thus, it is not unusual for Windows 95 users to keep a mail program, a web browser and a word processor all running at once. Each program or *process* gets time successively if it has work to do.

In the same manner, you can subdivide a program into little processes called *threads* and have them execute simultaneously, each getting successive time slices from your operating system. The difference is that within your program, you must somehow tell the operating system how you want to break up your program into threads and when you want them to start executing.

One of the great strengths of Java over most other languages is that you can write a multi-threaded program very simply by making just a few changes to a single-threaded program. In other words, Java makes this kind of multitasking trivially simple compared to almost any other language.

The Thread Object

Of course, in Java a thread is an object with a set of methods associated with it.

```java
public void checkAccess();
public void destroy();
public static int enumerate(Thread tarray[]);
public final String getName();
public final int getPriority();
public final ThreadGroup getThreadGroup();
public void interrupt();
public static boolean interrupted();
public final boolean isAlive();
public final boolean isDaemon();
public boolean isInterrupted();
public final void join();
```

Copyright © 2000, by James W. Cooper
public final void join(long millis);
public final void resume();
public void run();
public final void setDaemon(boolean on);
public final void setName(String name);
public final void setPriority(int newPriority);
public static void sleep(long millis);
public void start();
public final void stop();
public final void suspend();
public static void yield();

As you can see, you can start, stop, suspend and resume threads as well as creating and destroying them. The most important of these methods is the run method, although you never call it yourself. Instead, the operating system calls it for you when it is that thread’s turn to receive a time slice.

Making an Object Into a Thread

There are two ways to make a class into a thread. You can either extend the Thread class

class Timer extends Thread

or, if your class is already the descendent of another class, you can simply indicate that it implements the Runnable interface:

class ThreadList extends Frame implements Runnable

Now the Runnable interface simply indicates that that class has a run method.

public interface java.lang.Runnable {
    public abstract void run();
}

and in fact all you need to do is to include a run method and that method will execute in a separate thread.
Reasons for Using Threads

The trouble with most explanations of threads is that they never tell you what they’re for, unless you want to write little animation programs. While such animations are diverting, they are by no means the most significant purpose of threads. Threaded programs allow you to take time consuming parts of programs and execute them as background processes while the user interface remains lively and interactive.

To illustrate this point, let’s consider a program which needs to load a large number of names into a list box. This program will essentially freeze until the data are completely read from the input file. We’ll first write this program using a single process without threads so you can see how easy it is to change and so you can try both versions on the example disk. We are going to read in a list of names from the file spnames.txt and add them to the listbox. We’ll use the InputFile class we developed in chapter 12 to save time:

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

public class LoadList extends XFrame implements ActionListener
{
    private List namelist;          //listbox
    private Button openit, closit;  //and two buttons
    //-------------------------------
    public LoadList() {
        super ("Normal List loader"); //title bar
        setBackground(Color.lightGray);
        setLayout(new BorderLayout());
        namelist = new List(15, false); //create list
        namelist.setBackground(Color.white);
        openit = new Button("Open");  //and buttons
        closit = new Button("Close");
        openit.addActionListener (this);
        closit.addActionListener (this);
        Panel p1 = new Panel();
        add("Center", namelist);      //add list to middle
        add("South", p1);             //and panel to bottom
        p1.add(openit);
        p1.add(closit);               //add buttons to panel
        setBounds (100, 100, 250,300);
        setVisible(true);
    }
    //-------------------------------
}
```

Copyright © 2000, by James W. Cooper
public void actionPerformed (ActionEvent evt) {
    Object obj = evt.getSource ();
    if (obj == openit) {
        clickedOpenit();       //open and load file
    }
    else
        if (obj == closit) {
            clickedClosit();      //exit on close
        }
}

private void clickedClosit() {
    System.exit(0);
}

public void clickedOpenit() {
    //read in data file
    try {
        FileInputStream f = new FileInputStream("spnames.txt");
        String s = "";
        while (s != null) {
            s = f.readLine();   //read a line at a time
            if ((s != null) && (s.length ()>0))
                namelist.add(s);//and add to list
        }
        f.close();
    } catch(IOException e) {
        System.out.println(e.getMessage());
    }
}

public static void main(String argv[]) {
    new LoadList();
}

The significant part of this program is the clickedOpenit method, which
is executed when the user clicks on the Open button. It opens the file and
reads it a line at a time loading it into the listbox. The display it produces is
shown in Figure 13-1 and the program is called LoadList.java and it is in the
\chapter13 directory of your example disk.

During the time the file is being read in, you can click on the Close
button and it will appear to press in, but nothing will happen: the program
will not exit. This is because while the graphics methods run in their own

Copyright © 2000, by James W. Cooper
thread in Java, they have no effect on the program execution thread, which is busy reading the file.

Figure 13-1: The single-threaded version of the LoadList program.

Adding Threads to this Program

Now let's consider how we could make a small change in this program which will allow it to react to other user commands while it is reading in the list of names. In the program above, all the time consuming activity is localized in the clickedOpenit method. If we could make that method run in another thread we could make the program respond at once to other button clicks.

In order to convert this program we need to do only three things:

1. Add implements Runnable to the class definition.

2. Put the time consuming code in a run method.
3. Create an instance of the Thread class within this class, and call its **start** method.

So this simple set of changes gives us the program below:

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

public class ThreadList extends XFrame implements ActionListener, Runnable {
    private List namelist;          //listbox
    private Button openit, closit;  //and two buttons
    private Thread filler;          //-------------

    public ThreadList() { 
        super ("Normal List loader"); //title bar
        setBackground(Color.lightGray); 
        setLayout(new BorderLayout()); 
        namelist = new List(15, false); //create list 
        namelist.setBackground(Color.white); 
        openit = new Button("Open");    //and buttons 
        closit = new Button("Close"); 
        openit.addActionListener(this); 
        closit.addActionListener(this); 
        Panel p1 = new Panel(); 
        add("Center", namelist); //add list to middle 
        add("South", p1);         //and panel to bottom 
        p1.add(openit); 
        p1.add(closit);          //add buttons to panel 
        setBounds (100, 100, 250,300); 
        setVisible(true); 
    }

    public void actionPerformed (ActionEvent evt) { 
        Object obj = evt.getSource(); 
        if (obj == openit) { 
            filler = new Thread(this); 
            filler.start();
        } else 
        if (obj == closit) { 
            clickedClosit(); //exit on close 
        }
    }

    private void clickedClosit() {
        System.exit(0);
    }
}
```

Copyright © 2000, by James W. Cooper
public void run() {
//read in data file
    try {
        InputFile f = new InputFile("spnames.txt");
        String s = ";
        while (s != null) {
            s = f.readLine();   //read a line at a time
            if ((s != null) && (s.length ()>0))
                namelist.add(s);//and add to list
        }
        f.close();
    }
    catch(IOException e) {
        System.out.println(e.getMessage());
    }
//------------------------------
    public static void main(String argv[]) {
        new ThreadList();
    }

Let’s consider how these changes work. By including implements Runnable, we are simply saying that the class will have a run method. Then, when someone clicks on the Open button, we create an instance of a thread with

filler = new Thread(this);  //instance of thread

This creates a thread which can be started within the current instance of this class. Note that we use the reserved word this to indicate that we mean the current instance of ThreadList, not some new instance. Then, to launch code that will run in a separate thread, all we have to do is start the thread:

filler.start();

You might find these last two operations confusing. The only instance of the ThreadList class is running as a main thread. How can we also have it running in another thread? We don’t, of course. The code that runs in the new thread is contained in this class’s run method. The rest of the code continues to run in the main thread. Again, note that we never call the run method. We launch the thread by calling start: the Java thread system calls the run method when that thread is to be given a time slice.
This version of the program is called ThreadList in the \Threads directory of your example disk. When running the display is shown in Figure 13-2. The only difference is that if you click on the Close button while the list is loading, the program will exit immediately.

Figure 13-2: The ThreadList program.

Another Syntax for Starting a Thread

The Java documentation suggests that you create a thread for the current instance of the class using the code:

```java
new Thread(this).start();       //start the thread
```

This is really the same thing as the code above in more compressed to a less readable form. We create a thread and start it in one statement instead of two. This is just an example of how to write unclear code, like the ternary operator we discussed earlier. The compiler will generate the same code for either of them. A program using this terser method is provided as ThreadList2.java.
Several Threads in the Same PROGRAM

You can also write programs where a single class is used for several threads at once. Most of the examples of this come from visual effects such as animation. Let’s suppose we wanted to write a program where two squares blink on and off at different rates.

In the Java Blinker program below, we create a panel class with the Runnable interface. Then the instances of this class are run as separate threads to allow them to blink between two colors at different rates, in this case 0.5 sec and 0.3 sec.

We use the thread’s sleep method to delay execution within that thread for a specified interval. The run method is just a continuous loop with the sleep method delaying its execution each time:

```java
//almost all of the run method
while (true) {
    if (thiscolor == color1)
        thiscolor = color2;
    else
        thiscolor = color1;
    setBackground(thiscolor);
    repaint(); //redraw in thiscolor
    Thread.sleep(interval); //almost all
}
```

You will note that we inserted a couple of caveats in the comments for this routine. Since it is possible that the sleep method could be interrupted, the Java compiler requires that we catch exceptions surrounding the sleep method:

```java
try {
    Thread.sleep(interval);
} catch (InterruptedException e) {};
```

The complete Java version of the blinker program is on your example disk as Blinker.java and is show below. The running program is shown in Fig 13-5.
Fig 13-5: The Java Blinker program

import java.awt.*;
import java.awt.event.*;
import java.util.*;
//Blinks two squares at different rates
public class Blinker extends XFrame
    implements ActionListener {
    Button start; //start pushbutton
    Blinky b1, b2; //timed panels
    //------------------------------------
    public Blinker() {
        super("Two blinkers"); //title bar
        setBackground(Color.lightGray);
        setLayout(new BorderLayout());
        Panel cPanel = new Panel();
        add("Center", cPanel);
        cPanel.setLayout(new GridLayout(1, 2));
        //create two instances of panel class that
        //is a runnable thread
        b1 = new Blinky(Color.red, Color.green, 500);
        b2 = new Blinky(Color.blue, Color.yellow, 300);
        cPanel.add(b1);
        cPanel.add(b2);
        //add start button
        start = new Button("Start");
        Panel bPanel = new Panel();
        add("South", bPanel);
        bPanel.add(start);
start.addActionListener (this);
setBounds(100,100,300,250);
setVisible(true);
}
//-------------------------------
public void actionPerformed(ActionEvent evt) {
ob1.start();
ob2.start();
}
//-------------------------------
public static void main(String argv[]) {
    new Blinky();
}

and the Blinky class is a derived class of Panel which implements the Runnable interface.

import java.awt.*;
public class Blinky extends Panel implements Runnable {
    Color color1, color2, thiscolor;
    int interval;
    Thread blinkit;

    public Blinky(Color c1, Color c2, int intval) {
        color1 = c1;           //store colors
        color2 = c2;
        interval = intval;     //store interval
        thiscolor= color1;
        setBackground(color1);
    }
    //-------------------------------
    public void start() {
        blinkit = new Thread(this);    //create a thread
        blinkit.start();               //and start it
    }
    //-------------------------------
    public void run() {
        while (true) {
            if (thiscolor == color1)
                thiscolor = color2;
            else
                thiscolor = color1;
            setBackground(thiscolor);
            repaint();
            try {
                Thread.sleep(interval);
            }
        }
    }
}
catch (InterruptedException e) {
    
}
}

Building a Timer Class

You can also build classes which are derived directly from the Thread class. Even in these classes you also must override the run method and include the code you want to have executed in a separate thread. Let’s build a class like the Visual Basic Timer control, which you can set to “tick” at a specified rate.

This class will delay and then execute a tick at a specified interval. The question is, how will it tell the parent class that it has ticked, and how can we make this a general feature. This is where the idea of an interface becomes extremely useful. We’ll define an interface Ticker as follows:

interface Ticker {
    public abstract void tick();
}

Now all this says is that whatever class implements the Ticker interface must have a method called tick(). This is very useful because we now can have a way for the parent class of our timer to tell the timer class where to notify that the time for a tick occurs. The Timer class itself is just the following:

public class Timer extends Thread {
    private int interval;
    private Ticker parent = null;
    private boolean runFlag;
    //------------------------------------
    public Timer(Ticker obj, int intval) {
        interval = intval;
        parent = obj;
        runFlag = true;     //can run
    }
    //----------------------------------------
    public void stopTimer () {
        runFlag = false;
    }
    //----------------------------------------
    public void run() {
        while (runFlag ) {

    Copyright © 2000, by James W. Cooper
parent.tick();  //call parent tick routine
try {
    sleep(interval);     //wait a while
} 
catch (InterruptedException e) {
};
}
Note that this Timer class is derived from Thread and therefore has all of its
methods without any special interface or reference to the Thread class. Thus
we can call the sleep() method directly. The stopTimer method sets a flag
which causes the run() method to exit on the next loop. This is safer than the
previously deinfed stop() method which is no longer recommended for
Thread classes.

The class constructor passes in a reference to the class which creates Timer
and the value of the timing interval. Note that the type of the parent is
declared to be Ticker, meaning that the parent class has a tick() method. It
doesn’t matter that the parent class is really of type TimerBox and that it
only implements the interface Ticker. The advantage of this approach is that
a parent class of any type at all can use the Timer class, as long as it
implements the Ticker interface.

Then the timer’s run method simply loops until the runFlag variable
becomes false, calling tick in the parent class and then sleeping for the
specified interval. Now lets discuss how we can use this Timer class in a
real program.

The TimerBox Program

The TimerBox program, on your example disk as TimerBox.java, is another
simple example program which causes a square to change color. You can
start and stop the flashing using two pushbuttons as show in Figure 13-6.
import java.awt.*;
import java.awt.event.*;

public class TimerBox extends XFrame implements Ticker, ActionListener {
    private Button start, stop, close;  //three buttons
    private Panel blinkPanel;
    private Timer timer;
    private Color color;
    //------------------------------------
    public TimerBox() {
        super("Timed Display Blinker"); //title bar
        setBackground(Color.lightGray);
        setLayout(new BorderLayout());
        //create buttons
        start = new Button("Start");    //and buttons
        stop = new Button("Stop");
        close = new Button("Close");
        start.addActionListener (this);
        stop.addActionListener (this);
        close.addActionListener (this);
        blinkPanel = new Panel();     //create blinking
        add("Center", blinkPanel);
        Panel botPanel = new Panel();
        add("South", botPanel);                    //panel
        botPanel.add(start);
    }
}

Fig 13-6: The Timerbox program.

Copyright © 2000, by James W. Cooper
botPanel.add(stop);     //add buttons to panel
botPanel.add(close);

setBounds (100, 100, 250,200);
color = Color.red;              //initial color
blinkPanel.setBackground(color);
setVisible(true);
}

public void tick() {
//called by Timer class
    if (color == Color.blue)
        color = Color.red;
    else
        color = Color.blue;
    blinkPanel.setBackground(color);
    blinkPanel.repaint();
}

public void actionPerformed (ActionEvent evt) {
    Object obj = evt.getSource ();
    if (obj == start) {        //start button
        timer = new Timer(this, 200);       //create timer
        timer.start();        //begin timer ticking
        start.setEnabled(false);              //turn off start button
    }
    else
        if (obj == stop) {         //stop button
            timer.stopTimer();      //stop timer
            start.setEnabled(true);               //turn on start button
        }
        else
            if (obj == close) {       //close button
                System.exit(0);           //program exit
            }

}

public static void main(String argv[]) {
    new TimerBox();
}

Note that our TimerBox class extends the Frame class and implements the Ticker interface. Then because it has an interface of type Ticker, the compiler will allow it to create an instance of Timer and pass in the current instance of TimerBox using this. When the Timer class ticks, it calls this
class’s tick method which then switches the color of the panel and calls the `repaint()` method to change the color on the screen.

Since the Timer class is derived directly from Thread, it already has a `start` method, and we use it with the push buttons on the user interface to start the timer. We create the timer when we click on the start button, and we destroy the timer when we click on the stop button, and set the runFlag to false.

**Synchronized Methods**

A **synchronized** method can be called by only one thread at a time. If several threads try to call it at once, all but one are put on hold until the first one has left that method. This sort of method is needed when you need to make sure that two threads don’t attempt to change data at the same time.

For example, whenever you are keeping numerical totals of any kind and a thread reads a value, increments it and puts it back, you want to make sure that other threads aren’t doing this at the same time. This sort of access protection is clearly needed in any sort of financial or accounting program, where two threads should not attempt to modify a bank balance at the same time and write it back to the total.

**The Model Town Bank**

Let’s consider a simple model of a town bank. To keep this really simple, we’ll assume that there are 5 depositors who transact business only with each other. As the curtain rises, each depositor happens to have exactly $1000 in his account. We employ one teller to fetch each persons records, withdraw from one person’s account and deposit to another person’s account.

Now let’s assume that business picks up suddenly: a casino opens or a Ponzi scheme begins to flourish and the depositors transact so much business in a day that the bank has to hire 2 more tellers to keep up. (Never mind where they keep their money.) Now it becomes more difficult: more than one teller may need access to a person’s account at a time in order to withdraw or deposit money.
Let’s represent our tellers by 3 threads and our accounts by an array of integers, to avoid accidental roundoff error. If we write a program to illustrate this behavior, we simply use a random number generator to decide which two customers are transacting business, and in the interests of ever increasing job specialization, we assign each teller a single transaction amount: one will do all one dollar transactions, one will do all five dollar transactions and one will do only seven dollar transactions.

We give each teller access to the array of depositor’s balances and start them working, moving money around like mad. The Balances.java program is on your example disk and is shown below.

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

public class Balances extends XFrame implements ActionListener {
    private Label names[]; //array of customer name labels
    private TextField sbalances[]; //displayed array of balances
    private int balances[]; //integer array of balances
    private Button start, stop; //buttons to start and stop business
    private int sum;
    private Teller teller1, teller2, teller3; //3 Tellers
    //-----------------------------
    public Balances() {
        super("Balance sheet"); //title bar
        setBackground(Color.lightGray);
        setLayout(new BorderLayout());

        Panel p1 = new Panel(); //holds names & balances
        Panel p2 = new Panel(); //holds buttons
        add("South", p2);
        add("Center", p1);
        p1.setLayout(new GridLayout(6, 2)); //names and balances
        balances = new int[5]; //array of balances
        sbalances = new TextField[6]; //displayed balances

        sum = 0;
        names = new Label[6];
    }
}
```

Copyright © 2000, by James W. Cooper
names[0] = new Label("Alfie");
names[1] = new Label("Bonnie");
names[3] = new Label("Daphne");
names[4] = new Label("Essie");
names[5] = new Label("TOTAL");
for (int i = 0; i < 5; i++) {
    balances[i] = 1000;                //initialize
    sum += balances[i];
    sbalances[i] = new TextField(5);
    sbalances[i].setText(new Integer(balances[i]).toString());
    Panel p = new Panel();
    p1.add(p);
    p1.add(names[i]);
    p.add(sbalances[i]);
    p1.add(p);
}
sbalances[5] = new TextField(5);
sbalances[5].setText(new Float(sum).toString());
Panel p = new Panel();
p1.add(names[5]);            //add in TOTAL and
p1.add(p);
p.add(sbalances[5]);         //display sum
start = new Button("Start"); //add in buttons
stop = new Button("Stop");
start.addActionListener (this);
stop.addActionListener (this);

p2.add(start);
p2.add(stop);
setBounds(100,100,300,250);
setVisible(true);
}
//-------------------------------
public void paint(Graphics g) {
    //repaint values each time
    float sum = 0;
    for (int i = 0; i < 5; i++) {
        sbalances[i].setText( new Integer(balances[i]).toString());
        sum += balances[i];
    }
sbalances[5].setText( new Float(sum).toString());
}
//-------------------------------
public void clickedStart() {
    //start tellers running about
teller1 = new Teller(balances, 1);
teller2 = new Teller(balances, 5);
teller3 = new Teller(balances, 7);
//start your tellers
    teller1.start();
    teller2.start();
    teller3.start();
    start.setEnabled(false); //only once
}

//-------------------------------
public void clickedStop() {
    teller1.stopTimer(); //stop tellers
    teller2.stopTimer();
    teller3.stopTimer();
    start.setEnabled(true); //allow restarting
    repaint();
}
//-------------------------------
public void actionPerformed(ActionEvent evt) {
    Object obj = evt.getSource();
    if (obj == start) {
        clickedStart();
    }
    else
        if (obj == stop) {
            clickedStop();
        }
}
//-------------------------------
public static void main(String argv[]) {
    new Balances();
}

and the Teller classes actually move the money:

public class Teller extends Thread {
    int count;
    float amount;
    int balances[];
    boolean runFlag;
    //-------------------------------
    public Teller(int accounts[], float amt) {
        count = accounts.length; //total number of customers
        amount = amt; //save amount to move
        balances = accounts;
        runFlag = true;
    }
    //-------------------------------
public void stopTimer() {
    runFlag = false;
}

//--------------------------------------
public void run() {
    while (runFlag) {
        //compute customer to move money from and to
        int from =
            (int)Math.round(Math.random() *(count-1));
        int to =
            (int)Math.round(Math.random() *(count-1));
        if (from != to) {
            int fbalance = balances[from];
            int tbalance = balances[to];
            fbalance -= amount;
            tbalance += amount;
            balances[from] = fbalance;
            balances[to] = tbalance;
        }
    }
}

Now, if we run this program, click on the Start button, and let it execute for
a minute or so and then click on Stop, we find that the total is no longer
$5000! This is illustrated in Figure 13-7.

![Balance sheet](image)

Fig 13-7: The Balances program, showing how parallel actions
can result in out of balance accounts.

Copyright © 2000, by James W. Cooper
What happened? Is one of our tellers light fingered? Well, if you look at the code in the Teller class, it laboriously fetches the balances, adds and subtracts the amount specified and puts it back:

```java
//move money
int fbalance = balances[from];
int tbalance = balances[to];
fbalance -= amount;
tbalance += amount;
balances[from] = fbalance;
balances[to] = tbalance;
```

If we consider what each teller is doing, we realize that it is perfectly possible for two tellers to be handing the same account at once and both subtracting or adding money at the same time. Then the accounts become out of balance as shown in the figure.

A Bureaucratic Solution

The obvious bureaucratic solution is to prevent more than one teller from accessing the balances at a time. We could do this by creating a synchronized moveMoney method in the main program:

```java
public synchronized void moveMoney(int from,
                                        int to, float amt)
    //this routine moves money from customer "from"
    //to customer number "to"
{
    if (from != to)               //but not if same person
    {
        int frombalance = balances[from]; //get each balance
        int tobalance = balances[to];
        frombalance -= amt;            //withdraw from one
        tobalance += amt;              //and deposit to other
        balances[from] = frombalance;  //put back
        balances[to] = tobalance;
    }
}
```

Then each teller receives a reference to the calling class and calls that moveMoney method as needed:

```java
class Teller extends Thread
{
    int count;
}
```

Copyright © 2000, by James W. Cooper
Balances1 bank;       //calling class ref
float amount;
                          //-------------------------------
public Teller(Balances1 fr, int cnt, float amt)
{
    count = cnt;   //total number of customers
    bank = fr;     //reference to parent class
    amount = amt;  //save amount to move
}
                          //-------------------------------
public void run()
{
    while (true)
    {
        //compute customer to move money from and to
        int from = (int)Math.round(Math.random() *(count-1));
        int to = (int)Math.round(Math.random() *(count-1));

        //and ask bank to do it
        bank.moveMoney(from, to, amount);
    }
}
}

This program is available on your example disk as Balances1.java.

Now the discerning reader will observe, unlike the average bureaucrat, that using this approach means that only one teller can move money at once. Logically, this is as if they had to stand in line to move funds one at a time, thus augmenting customer lines with teller lines. Clearly this is not very satisfactory.

**Objects in the Bank**

A better solution to this teller overlap problem is to prevent two tellers from accessing the same *account* at once rather than the whole bank. We can do this by redesigning each customer account as an object.

    public class Customer {
        int balance;
                            //-------------------------------
        public Customer(int money) {
            balance = money;
        }
    }                        //-------------------------------
synchronized public void changeLedger(int value) {
    balance += value;
} //-----------------------------------
public int getBalance() {
    return balance;
}

Then we can create an array of these objects for the tellers to work on:

Customer customers[] = new Customer[5];
for (int i = 0; i < 5; i++) {
    customers[i] = new Customer(1000);
}

band within the Teller objects, we adjust the balances:

Customer custf, custto;

//compute two customers
int from = (int)Math.round(Math.random() * (count-1));
int to = (int)Math.round(Math.random() * (count-1));
if (from != to) {
    custf = (Customer)cust[from];    //get customers
    custto = (Customer)cust[to];
    custf.changeLedger(-amount);     //move money
    custto.changeLedger(amount);
} //end if

Since the changeLedger method is declared synchronized, the balances will never get out of alignment. You can try this in the program ObjBalances.java and if you remove the synchronized modifier, you will see that the teller threads again can interfere with each other. This program is on your sample disk as ObjBalances.java, and the resulting display is shown in Figure 13-8.
Synchronizing Classes You Can’t Change

If you don’t have the source and for some reason can’t derive a new class from such a class, you can synchronize instances of a class using the `synchronized` modifier as an operator:

```java
public void run() {
    Customer custf, custto;
    while (! stop_flag) {
        int from = (int)Math.round(Math.random() *(count-1));
        int to = (int)Math.round(Math.random() *(count-1));
        if (from != to) {
            custf = (Customer)cust[from];
            custto = (Customer)cust[to];
            synchronized(custf) {
                custf.changeLedger(-amount);
            }
            synchronized(custto) {
                custto.changeLedger(amount);
            }
        } //end if
    } //end while
}
```

In the code above we apply the synchronized operator to the class instances `custf` and `custto`. Then no other can access these objects while this thread is
using them. This version of the program is on your example disk as ObjBalances1.java.

Other Thread Properties
We’ve marched through most of the important features of threads in this chapter. However, to finish up we need to discuss daemon threads, thread priorities, and thread groups.

Daemon Threads
Daemon threads are threads that continue to run in the background throughout a program. However, if they are marked as daemon threads, you don’t have to specifically shut them down for the program to terminate. If the only threads left running are daemon threads the Java Virtual Machine will exit automatically.

To define a thread as a daemon thread, you must use the `setDaemon` method before you start the thread.

```java
Thread t = new Thread();
t.setDaemon(true);
t.start();
```

Thread Priorities
In addition to starting threads, you can define the priority at which they will run, where higher priority threads get proportionately more time than lower priority threads. The `setPriority` method is used to define a thread’s priority. The priorities values currently can be set between 1 and 10 with 5 being the default setting. However, the numerical values are not guaranteed to remain constant between Java releases, and we recommend that you use the following constants instead.

```java
t.setPriority(Thread.NORM_PRIORITY);
t.setPriority(Thread.MAX_PRIORITY);
t.setPriority(Thread.MIN_PRIORITY);
```
Thread Groups

Java allows you to group threads into named groups rather like the
CheckboxGroups we discussed earlier. The purpose of these groups is to
allow you to destroy an entire group of threads at once. Since a thread group
can itself contain thread groups, it is possible to build up a tree of threads
which you can suspend, resume, stop or destroy all at once. Typical code for
creating and using thread groups is:

```java
ThreadGroup grp = new ThreadGroup();
Thread th1 = new Thread(grp, "tom"); // put in group
Thread th2 = new Thread(grp, "disk"); // put in group
Thread th3 = new Thread(grp, "hery"); // put in group
grp.suspend(); // suspend all threads
```

Summary

Now that we’ve looked into threads we really can see the power of Java and
how it can be used. We’ve looked at threads used for timing and threads
used for delays. We’ve also looked at how we can use several threads
running simultaneous parallel processes. Now that we understand how
threads work we can talk about how to read in and display images.