Containers

This chapter covers a special type of Component called Container. A Container is a subclass of Component that can contain other components, including other containers. Container allows you to create groupings of objects on the screen. This chapter covers the methods in the Container class and its subclasses: Panel, Window, Frame, Dialog, and FileDialog. It also covers the Insets class, which provides an internal border area for the Container classes.

Every container has a layout associated with it that controls how the container organizes the components in it. The layouts are described in Chapter 7, Layouts.

Java 1.1 introduces a special Container called ScrollPane. Because of the similarities between scrolling and ScrollPane, the new ScrollPane container is covered with the Scrollbar class in Chapter 11, Scrolling.

6.1 Container

Container is an abstract class that serves as a general purpose holder of other Component objects. The Container class holds the methods for grouping the components together, laying out the components inside it, and dealing with events occurring within it. Because Container is an abstract class, you never see a pure Container object; you only see subclasses that add specific behaviors to a generic container.
6.1.1 Container Methods

Constructors

The abstract Container class contains a single constructor to be called by its children. Prior to Java 1.1, the constructor was package private.

protected Container()

The constructor for Container creates a new component without a native peer. Since you no longer have a native peer, you must rely on your container to provide a display area. This allows you to create containers that require fewer system resources. For example, if you are creating panels purely for layout management, you might consider creating a LightweightPanel class to let you assign a layout manager to a component group. Using LightweightPanel will speed things up since events do not have to propagate through the panel and you do not have to get a peer from the native environment. The following code creates the LightweightPanel class:

```java
import java.awt.*;
public class LightweightPanel extends Container {
   LightweightPanel () {} 
   LightweightPanel (LayoutManager lm) {
      setLayout(lm); 
   }
}
```

Grouping

A Container holds a set of objects within itself. This set of methods describes how to examine and add components to the set.

public int getComponentCount ()

public int countComponents ()

The getComponentCount() method returns the number of components within the container at this level. getComponentCount() does not count components in any child Container (i.e., containers within the current container).

countComponents() is the Java 1.0 name for this method.

public Component getComponent (int position)

The getComponent() method returns the component at the specific position within it. If position is invalid, this method throws the run-time exception ArrayIndexOutOfBoundsException.
public Component[] getComponents ()
    getComponents() returns an array of all the components held within the container. Since these are references to the actual objects on the screen, any changes made to the components returned will be reflected on the display.

public Component add (Component component, int position)
The add() method adds component to the container at position. If position is -1, add() inserts component as the last object within the container. What the container does with position depends upon the LayoutManager of the container. If position is invalid, the add() method throws the run-time exception IllegalArgumentException. If you try to add component’s container to itself (anywhere in the containment tree), this method throws an IllegalArgumentException. In Java 1.1, if you try to add a Window to a container, add() throws the run-time exception IllegalArgumentException. If you try to add component to a container that already contains it, the container is removed and re-added, probably at a different position.

Assuming that nothing goes wrong, the parent of component is set to the container, and the container is invalidated. add() returns the component just added.

Calling this method generates a ContainerEvent with the id COMPONENT_ADDED.

public Component add (Component component)
The add() method adds component to the container as the last object within the container. This is done by calling the earlier version of add() with a position of -1. If you try to add component’s container to itself (anywhere in the containment tree), this method throws the run-time exception IllegalArgumentException. In Java 1.1, if you try to add a Window to a container, add() throws the run-time exception IllegalArgumentException.

Calling this method generates a ContainerEvent with the id COMPONENT_ADDED.

public void add (Component component, Object constraints)
public void add (String name, Component component)
This next version of add() is necessary for layouts that require additional information in order to place components. The additional information is provided by the constraints parameter. This version of the add() method calls the addLayoutComponent() method of the LayoutManager. What the container does with constraints depends upon the actual LayoutManager. It can be used for naming containers within a CardLayout, specifying a screen area for BorderLayout, or providing a set of GridBagConstraints for a GridBagLayout. In the event that this add() is called and the current LayoutManager does not take advantage of constraints, component is added at the end with a position
of -1. If you try to add component’s container to itself (anywhere in the containment tree), this method throws the run-time exception `IllegalArgumentException`. In Java 1.1, if you try to add a `Window` to a container, `add()` throws the run-time exception `IllegalArgumentException`.

The `add(String, Component)` method was changed to `add(component, object)` in Java 1.1 to accommodate the `LayoutManager2` interface (discussed in Chapter 7) and to provide greater flexibility. In all cases, you can just flip the parameters to bring the code up to 1.1 specs. The string used as an identifier in Java 1.0 is just treated as a particular kind of constraint.

Calling this method generates a `ContainerEvent` with the id `COMPONENT_ADDED`.

```java
public void add (Component component, Object constraints, int index)
```

This final version of `add()` is necessary for layouts that require an index and need additional information to place components. The additional information is provided by the `constraints` parameter. This version of `add()` also calls the `addLayoutComponent()` method of the `LayoutManager`. `component` is added with a position of `index`. If you try to add `component`’s container to itself (anywhere in the containment tree), this method throws the run-time exception `IllegalArgumentException`. In Java 1.1, if you try to add a `Window` to a `Container`, `add()` throws the run-time exception `IllegalArgumentException`.

Some layout managers ignore any index. For example, if you call `add(aButton, BorderLayout.NORTH, 3)` to add a `Button` to a `BorderLayout` panel, the `Button` appears in the north region of the layout, no matter what the index.

Calling this method generates a `ContainerEvent` with the id `COMPONENT_ADDED`.

```java
protected void addImpl(Component comp, Object constraints, int index)
```

The protected `addImpl()` method is the helper method that all the others call. It deals with synchronization and enforces all the restrictions on adding components to containers.

The `addImpl()` method tracks the container’s components in an internal list. The index with which each component is added determines its position in the list. The lower the component’s index, the higher it appears in the stacking order. In turn, the stacking order determines how components are displayed when sufficient space isn’t available to display all of them. Components that are added without indices are placed at the end of the list (i.e., at the end of the stacking order) and therefore displayed behind other components. If all components are added without indices, the first component added to the container is first in the stacking order and therefore displayed in front.
You could override `addImpl()` to track when components are added to a container. However, the proper way to find out when components are added is to register a `ContainerListener` and watch for the `COMPONENT_ADDED` and the `COMPONENT_REMOVED` events.

```java
public void remove(int index)
```

The `remove()` method deletes the component at position `index` from the container. If `index` is invalid, the `remove()` method throws the run-time exception `IllegalArgumentException`. This method calls the `removeLayoutComponent()` method of the container’s `LayoutManager`.

`removeAll()` generates a `ContainerEvent` with the id `COMPONENT_REMOVED`.

```java
public void remove(Component component)
```

The `remove()` method deletes component from the container, if the container directly contains component. `remove()` does not look through nested containers trying to find component. This method calls the `removeLayoutComponent()` method of the container’s `LayoutManager`.

When you call this method, it generates a `ContainerEvent` with the id `COMPONENT_REMOVED`.

```java
public void removeAll()
```

The `removeAll()` method removes all components from the container. This is done by looping through all the components, setting each component’s parent to `null`, setting the container’s reference to the component to `null`, and invalidating the container.

When you call this method, it generates a `ContainerEvent` with the id `COMPONENT_REMOVED` for each component removed.

```java
public boolean isAncestorOf(Component component)
```

The `isAncestorOf()` method checks to see if component is a parent (or grandparent or great grandparent) of this container. It could be used as a helper method for `addImpl()` but is not. If `component` is an ancestor of the container, `isAncestorOf()` returns `true`; otherwise, it returns `false`.

### Layout and sizing

Every container has a `LayoutManager`. The `LayoutManager` is responsible for positioning the components inside the container. The `Container` methods listed here are used in sizing the objects within the container and specifying a layout.

```java
public LayoutManager getLayout()
```

The `getLayout()` method returns the container’s current `LayoutManager`. 


public void setLayout (LayoutManager manager)

The setLayout() method changes the container’s LayoutManager to manager and invalidates the container. This causes the components contained inside to be repositioned based upon manager’s rules. If manager is null, there is no layout manager, and you are responsible for controlling the size and position of all the components within the container yourself.

public Dimension getPreferredSize () ★

public Dimension preferredSize () ☆

The getPreferredSize() method returns the Dimension (width and height) for the preferred size of the components within the container. The container determines its preferred size by calling the preferredLayoutSize() method of the current LayoutManager, which says how much space the layout manager needs to arrange the components. If you override this method, you are overriding the default preferred size.

preferredSize() is the Java 1.0 name for this method.

public Dimension getMinimumSize () ★

public Dimension minimumSize () ☆

The getMinimumSize() method returns the minimum Dimension (width and height) for the size of the components within the container. This container determines its minimum size by calling the minimumLayoutSize() method of the current LayoutManager, which computes the minimum amount of space the layout manager needs to arrange the components. It is possible for getMinimumSize() and getPreferredSize() to return the same dimensions. There is no guarantee that you will get this amount of space for the layout.

minimumSize() is the Java 1.0 name for this method.

public Dimension getMaximumSize () ★

The getMaximumSize() method returns the maximum Dimension (width and height) for the size of the components within the container. This container determines its maximum size by calling the maximumLayoutSize() method of the current LayoutManager2, which computes the maximum amount of space the layout manager needs to arrange the components. If the layout manager is not an instance of LayoutManager2, this method calls the getMaximumSize() method of the Component, which returns Integer.MAX_VALUE for both dimensions. None of the java.awt layout managers use the concept of maximum size yet.
public float getAlignmentX ()

The getAlignmentX() method returns the alignment of the components within the container along the x axis. This container determines its alignment by calling the current LayoutManager2’s getLayoutAlignmentX() method, which computes it based upon its children. The return value is between 0.0 and 1.0. Values nearer 0 indicate that the component should be placed closer to the left edge of the area available. Values nearer 1 indicate that the component should be placed closer to the right. The value 0.5 means the component should be centered. If the layout manager is not an instance of LayoutManager2, this method calls Component’s getAlignmentX() method, which returns the constant Component.CENTER_ALIGNMENT. None of the java.awt layout managers use the concept of alignment yet.

public float getAlignmentY ()

The getAlignmentY() method returns the alignment of the components within the container along the y axis. This container determines its alignment by calling the current LayoutManager2’s getLayoutAlignmentY() method, which computes it based upon its children. The return value is between 0.0 and 1.0. Values nearer 0 indicate that the component should be placed closer to the top of the area available. Values nearer 1 indicate that the component should be placed closer to the bottom. The value 0.5 means the component should be centered. If the layout manager is not an instance of LayoutManager2, this method calls Component’s getAlignmentY() method, which returns the constant Component.CENTER_ALIGNMENT. None of the java.awt layout managers use the concept of alignment yet.

public void doLayout ()

public void layout ()

The doLayout() method of Container instructs the LayoutManager to lay out the container. This is done by calling the layoutContainer() method of the current LayoutManager.

layout() is the Java 1.0 name for this method.

public void validate ()

The validate() method sets the container’s valid state to true and recursively validates all of its children. If a child is a Container, its children are in turn validated. Some components are not completely initialized until they are validated. For example, you cannot ask a Button for its display dimensions or position until it is validated.
**protected void validateTree () ★**
The `validateTree()` method is a helper for `validate()` that does all the work.

**public void invalidate () ★**
The `invalidate()` method invalidates the container and recursively invalidates the children. If the layout manager is an instance of `LayoutManager2`, its `invalidateLayout()` method is called to invalidate any cached values.

**Event delivery**
The event model for Java is described in Chapter 4, *Events*. These methods help in the handling of the various system events at the container level.

**public void deliverEvent (Event e) ☆**
The `deliverEvent()` method is called by the system when the Java 1.0 `Event e` happens. `deliverEvent()` tries to locate a component contained in the container that should receive it. If one is found, the `x` and `y` coordinates of `e` are translated for the new target, and `Event e` is delivered to this by calling its `deliverEvent()`. If `getComponentAt()` fails to find an appropriate target, the event is just posted to the container with `postEvent()`.

**public Component getComponentAt (int x, int y) ★**
**public Component locate (int x, int y) ☆**
The container’s `getComponentAt()` method calls each component’s `contains()` method to see if the `x` and `y` coordinates are within it. If they are, that component is returned. If the coordinates are not in any child component of this container, the container is returned. It is possible for `getComponentAt()` to return `null` if the `x` and `y` coordinates are not within the container. The method `getComponentAt()` can return another `Container` or a lightweight component.
`locate()` is the Java 1.0 name for this method.

**public Component getComponentAt (Point p) ★**
This `getComponentAt()` method is identical to the previous method, with the exception that the location is passed as a single point, rather than as separate `x` and `y` coordinates.

**Listeners and 1.1 event handling**
With the 1.1 event model, you register listeners, which are told when events occur. Container events occur when a component is added or removed.

**public synchronized void addContainerListener(ContainerListener listener) ★**
The `addContainerListener()` method registers `listener` as an object
interested in receiving notifications when an ContainerEvent passes through the EventQueue with this Container as its target. The listener.componentAdded() or listener.componentRemoved() method is called when these events occur. Multiple listeners can be registered. The following code demonstrates how to use a ContainerListener to register action listeners for all buttons added to an applet. It is similar to the ButtonTest11 example in Section 5.3.2. The trick that makes this code work is the call to enableEvents() in init(). This method makes sure that container events are delivered in the absence of listeners. In this applet, we know there won't be any container listeners, so we must enable container events explicitly before adding any components.

// Java 1.1 only
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class NewButtonTest11 extends Applet implements ActionListener {
    Button b;
    public void init () {
        enableEvents (AWTEvent.CONTAINER_EVENT_MASK);
        add (b = new Button ("One"));
        add (b = new Button ("Two"));
        add (b = new Button ("Three"));
        add (b = new Button ("Four"));
    }

    protected void processContainerEvent (ContainerEvent e) {
        if (e.getID() == ContainerEvent.COMPONENT_ADDED) {
            if (e.getChild() instanceof Button) {
                Button b = (Button)e.getChild();
                b.addActionListener (this);
            }
        }
    }

    public void actionPerformed (ActionEvent e) {
        System.out.println ("Selected: " + e.getActionCommand());
    }
}

public void removeContainerListener(ContainerListener listener) ★

The removeContainerListener() method removes listener as an interested listener. If listener is not registered, nothing happens.

protected void processEvent(AWTEvent e) ★

The processEvent() method receives all AWTEvents with this Container as its target. processEvent() then passes them along to any listeners for processing. When you subclass Container, overriding processEvent() allows you to process all events yourself, before sending them to any listeners. There is no equivalent under the 1.0 event model.
If you override `processEvent()`, remember to call `super.processEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it’s a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

```java
protected void processContainerEvent(ContainerEvent e)
```

The `processContainerEvent()` method receives all `ContainerEvent`s with this `Container` as its target. `processContainerEvent()` then passes them along to any listeners for processing. When you subclass `Container`, overriding the `processContainerEvent()` method allows you to process all container events yourself, before sending them to any listeners. There is no equivalent under the 1.0 event model.

If you override the `processContainerEvent()` method, remember to call `super.processContainerEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it’s a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

### Painting

The following methods are early vestiges of an approach to painting and printing. They are not responsible for anything that couldn’t be done with a call to `paintAll()` or `printAll()`. However, they are available if you wish to call them.

```java
public void paintComponents(Graphics g)
```

The `paintComponents()` method of `Container` paints the different components it contains. It calls each component’s `paintAll()` method with a clipped graphics context `g`, which is eventually passed to `paint()`.

```java
public void printComponents(Graphics g)
```

The `printComponents()` method of `Container` prints the different components it contains. It calls each component’s `printAll()` method with a clipped graphics context `g`, which is passed to `print()`, and eventually works its way to `paint()`.

Since it is the container’s responsibility to deal with painting lightweight peers, the `paint()` and `print()` methods are overridden in Java 1.1.

```java
public void paint(Graphics g)
```

The `paint()` method of `Container` paints the different lightweight components it contains.
public void print(Graphics g) ★

The print() method of Container prints the different lightweight components it contains.

NOTE

If you override paint() or print() in your containers (especially applets), call super.paint(g) or super.print(g), respectively, to make sure that lightweight components are rendered. This is a good practice even if you don’t currently use any lightweight components; you don’t want your code to break mysteriously if you add a lightweight component later.

Peers

The container is responsible for creating and destroying all the peers of the components within it.

public void addNotify () ★

The addNotify() method of Container creates the peer of all the components within it. After addNotify() is called, the Container is invalid. It is useful for top-level containers to call this method explicitly before calling the method setVisible(true) to guarantee that the container is laid out before it is displayed.

public void removeNotify () ★

The removeNotify() method destroys the peer of all the top-level objects contained within it. This in effect destroys the peers of all the components within the container.

Miscellaneous methods

protected String paramString () ★

When you call the toString() method of a container, the default toString() method of Component is called. This in turn calls paramString() which builds up the string to display. At the Container level, paramString() appends the layout manager name, like layout=java.awt.BorderLayout, to the output.

public Insets getInsets () ★

public Insets insets () ★

The getInsets() method gets the container’s current insets. An inset is the amount of space reserved for the container to use between its edge and the area actually available to hold components. For example, in a Frame, the inset for the top would be the space required for the title bar and menu bar. Insets exist for top, bottom, right, and left. When you override this method, you are providing an area within the container that is reserved for free space. If the container has insets, they would be the default. If not, the default values are
all zeroes.

The following code shows how to override `insets()` to provide values other than the default. The top and bottom have 20 pixels of inset. The left and right have 50. Section 6.3 describes the `Insets` class in more detail.

```java
public Insets insets () { // getInsets() for Java 1.1
   return new Insets (20, 50, 20, 50);
}
```

To find out the current value, just call the method and look at the results. For instance, for a `Frame` the results could be the following in the format used by `toString()`:

```
java.awt.Insets[top=42,left=4,right=4,bottom=4]
```

The 42 is the space required for the title and menu bar, while the 4 around the edges are for the window decorations. These results are platform specific and allow you to position items based upon the user’s run-time environment.

When drawing directly onto the graphics context of a container with a large inset such as `Frame`, remember to work around the insets. If you do something like `g.drawString("Hello World", 5, 5)` onto a `Frame`, the user won’t see the text. It will be under the title bar and menu bar.

`insets()` is the Java 1.0 name for this method.

```java
public void list (PrintWriter output, int indentation)
public void list (PrintStream output, int indentation)
```

The `list()` method is very helpful if you need to find out what is inside a container. It recursively calls itself for each container level of objects inside it, increasing the indentation at each level. The results are written to the `PrintStream` or `PrintWriter` output.

### 6.2 Panel

The `Panel` class provides a generic container within an existing display area. It is the simplest of all the containers. When you load an applet into Netscape Navigator or an `appletviewer`, you have a `Panel` to work with at the highest level.

A `Panel` has no physical appearance. It is just a rectangular display area. The default `LayoutManager` of `Panel` is `FlowLayout`; `FlowLayout` is described in Section 7.2.
6.2.1 Panel Methods

Constructors

*public Panel ()*

The first constructor creates a Panel with a LayoutManager of FlowLayout.

*public Panel (LayoutManager layout)★*

This constructor allows you to set the initial LayoutManager of the new Panel to layout. If layout is null, there is no LayoutManager, and you must shape and position the components within the Panel yourself.

Miscellaneous methods

*public void addNotify ()*

The addNotify() method creates the Panel peer. If you override this method, first call super.addNotify(), then add your customizations for the new class. Then you can do everything you need with the information about the newly created peer.

6.2.2 Panel Events

In Java 1.0, a Panel peer generates all the events that are generated by the Component class; it does not generate events that are specific to a particular type of component. That is, it generates key events, mouse events, and focus events; it doesn’t generate action events or list events. If an event happens within a child component of a Panel, the target of the event is the child component, not the Panel. There’s one exception to this rule: if a component uses the LightweightPeer (new to Java 1.1), it cannot be the target of an event.

With Java 1.1, events are delivered to whatever listener is associated with a contained component. The fact that the component is within a Panel has no relevance.

6.3 Insets

The Insets class provides a way to encapsulate the layout margins of the four different sides of a container. The class helps in laying out containers. The Container can retrieve their values through the getInsets() method, then analyze the settings to position components. The different inset values are measured in pixels. The space reserved by insets can still be used for drawing directly within paint(). Also, if the LayoutManager associated with the container does not look at the insets, the request will be completely ignored.
6.3.1 Insets Methods

Variables
There are four variables for insets, one for each border.

\texttt{public int top}
This variable contains the border width in pixels for the top of a container.

\texttt{public int bottom}
This variable contains the border width in pixels for the bottom of a container.

\texttt{public int left}
This variable contains the border width in pixels for the left edge of a container.

\texttt{public int right}
This variable contains the border width in pixels for the right edge of a container.

Constructors

\texttt{public Insets (int top, int left, int bottom, int right)}
The constructor creates an Insets object with \texttt{top}, \texttt{left}, \texttt{bottom}, and \texttt{right} being the size of the insets in pixels. If this object was the return object from the \texttt{getInsets()} method of a container, these values represent the size of a border inside that container.

Miscellaneous methods

\texttt{public Object clone ()}
The \texttt{clone()} method creates a clone of the Insets so the same Insets object can be associated with multiple containers.

\texttt{public boolean equals(Object object) ★}
The \texttt{equals()} method defines equality for insets. Two Insets objects are equal if the four settings for the different values are equal.

\texttt{public String toString ()}
The \texttt{toString()} method of Insets returns the current settings. Using the new Insets \texttt{(10, 20, 30, 40)} constructor, the results would be:

\texttt{java.awt.Insets[top=10,left=20,bottom=30,right=40]}
6.3.2 Insets Example

The following source code demonstrates the use of insets within an applet’s Panel. The applet displays a button that takes up the entire area of the Panel, less the insets, then draws a rectangle around that area. This is shown visually in Figure 6-1. The example demonstrates that if you add components to a container, the Layout-Manager deals with the insets for you in positioning them. But if you are drawing directly to the Panel, you must look at the insets if you want to avoid the requested area within the container.

```java
import java.awt.*;
import java.applet.*;
public class myInsets extends Applet {
    public Insets insets () {
        return new Insets (50, 50, 50, 50);
    }
    public void init () {
        setLayout (new BorderLayout ());
        add ("Center", new Button ("Insets"));
    }
    public void paint (Graphics g) {
        Insets i = insets();
        int width = size().width - i.left - i.right;
        int height = size().height - i.top - i.bottom;
        g.drawRect (i.left-2, i.top-2, width+4, height+4);
        g.drawString ("Insets Example", 25, size().height - 25);
    }
}
```

To change the applet’s insets from the default, we override the `insets()` method to return a new `Insets` object, with the new values.

6.4 Window

A Window is a top-level display area that exists outside the browser or applet area you are working in. It has no adornments, such as the borders, window title, or menu bar that a typical window manager might provide. A Frame is a subclass of Window that adds these parts (borders, window title). Normally you will work with the children of Window and not Window directly. However, you might use a Window to create your own pop-up menu or some other GUI component that requires its own window and isn’t provided by AWT. This technique isn’t as necessary in Java 1.1, which has a PopupMenu component.

The default LayoutManager for Window is BorderLayout, which is described in Section 7.3.
6.4.1 Window Methods

Constructors

`public Window (Frame parent)`

There is one public constructor for `Window`. It has one parameter, which specifies the parent of the `Window`. When the parent is minimized, so is the `Window`. In an application, you must therefore create a `Frame` before you can create a `Window`; this isn't much of an inconvenience since you usually need a `Frame` in which to build your user interface. In an applet, you often do not have access to a `Frame` to use as the parent, so you can pass `null` as the argument.

Figure 6-2 shows a simple `Window` on the left. Notice that there are no borders or window management adornments present. The `Window` on the right was created by an applet loaded over the network. Notice the warning message you get in the status bar at the bottom of the screen. This is to warn users that the `Window` was created by an applet that comes from an untrusted source, and you can't necessarily trust it to do what it says. The warning is particularly appropriate for windows, since a user can't necessarily tell whether a window was created by an applet or any other application. It is therefore possible to write applets that mimic windows from well-known applications, to trick the user into giving away passwords, credit card numbers, or other sensitive information.

In some environments, you can get the browser's `Frame` to use with the `Window`'s constructor. This is one way to create a `Dialog`, as we shall see. By
repeatedly calling `getParent()` until there are no more parents, you can discover an applet’s top-level parent, which should be the browser’s `Frame`. Example 6-1 contains the code you would write to do this. You should then check the return value to see if you got a `Frame` or `null`. This code is completely nonportable, but you may happen to be in an environment where it works.

**Example 6–1: Finding a Parent Frame**

```java
import java.awt.*;
public class ComponentUtilities {
    public static Frame getTopLevelParent (Component component) {
        Component c = component;
        while (c.getParent() != null)
            c = c.getParent();
        if (c instanceof Frame)
            return (Frame)c;
        else
            return null;
    }
}
```

### Appearance methods

A handful of methods assist with the appearance of the `Window`.

**public void pack ()**

The `pack()` method resizes the `Window` to the preferred size of the components it contains and validates the `Window`. 
public void show ()

The show() method displays the Window. When a Window is initially created it is hidden. If the window is already showing when this method is called, it calls toFront() to bring the window to the foreground. To hide the window, just call the hide() method of Component. After you show() a window, it is validated for you.

The first call to show() for any Window generates a WindowEvent with the ID WINDOW_OPENED.

public void dispose ()

The dispose() method releases the resources of the Window by hiding it and removing its peer. Calling this method generates a WindowEvent with the ID WINDOW_CLOSED.

public void toFront ()

The toFront() method brings the Window to the foreground of the display. This is automatically called if you call show() and the Window is already shown.

public void toBack ()

The toBack() method puts the Window in the background of the display.

public boolean isShowing() ★

The isShowing() method returns true if the Window is visible on the screen.

Miscellaneous methods

public Toolkit getToolkit ()

The getToolkit() method returns the current Toolkit of the window. The Toolkit provides you with information about the native platform. This will allow you to size the Window based upon the current screen resolution and get images for an application. See Section 6.5.5 for a usage example.

public Locale getLocale () ★

The getLocale() method retrieves the current Locale of the window, if it has one. Using a Locale allows you to write programs that can adapt themselves to different languages and different regional variants. If no Locale has been set, getLocale() returns the default Locale. The default Locale has a user language of English and no region. To change the default Locale, set the system properties user.language and user.region or call Locale.setDefault() (setDefault() verifies access rights with the security manager).*

* For more on the Locale class, see the Java Fundamental Classes Reference from O'Reilly & Associates.
public final String getWarningString ()

The getWarningString() method returns null or a string that is displayed on the bottom of insecure Window instances. If the SecurityManager says that top-level windows do not get a warning message, this method returns null. If a message is required, the default text is “Warning: Applet Window”. However, Java allows the user to change the warning by setting the system property awt.appletWarning. (Netscape Navigator and Internet Explorer do not allow the warning message to be changed. Netscape Navigator’s current (V3.0) warning string is “Unsigned Java Applet Window.”) The purpose of this string is to warn users that the Window was created by an untrusted source, as opposed to a standard application, and should be used with caution.

public Component getFocusOwner () ★

The getFocusOwner() method allows you to ask the Window which of its components currently has the input focus. This is useful if you are cutting and pasting from the system clipboard; asking who has the input focus tells you where to put the data you get from the clipboard. The system clipboard is covered in Chapter 16, Data Transfer. If no component in the Window has the focus, getFocusOwner() returns null.

public synchronized void addNotify ()

The addNotify() method creates the Window peer. This is automatically done when you call the show() method of the Window. If you override this method, first call super.addNotify(), then add your customizations for the new class. Then you can do everything you need to with the information about the newly created peer.

6.4.2 Window Events

In Java 1.0, a Window peer generates all the events that are generated by the Component class; it does not generate events that are specific to a particular type of component. That is, it generates key events, mouse events, and focus events; it doesn’t generate action events or list events. If an event occurs within a child component of a Window, the target of the event is the child component, not the Window.

In addition to the Component events, five events are specific to windows, none of which are passed on by the window’s peer. These events happen at the Frame and Dialog level. The events are WINDOW_DESTROY, WINDOW_EXPOSE, WINDOW_ICONIFY, WINDOW_DEICONIFY, and WINDOW_MOVED. The default event handler, handleEvent(), doesn’t call a convenience method to handle any of these events. If you want to work with them, you must override handleEvent(). See Section 6.5.4 for an example that catches the WINDOW_DESTROY event.
public boolean postEvent (Event e) ★
The postEvent() method tells the Window to deal with Event e. It calls the
handleEvent() method, which returns true if somebody handled e and false
if no one handles it. This method, which overrides Component.postEvent(), is
necessary because a Window is, by definition, an outermost container, and
therefore does not need to post the event to its parent.

Listeners and 1.1 event handling
With the 1.1 event model, you register listeners for different event types; the listen-
ers are told when the event happens. These methods register listeners and let the
Window component inspect its own events.

public void addWindowListener(WindowListener listener) ★
The addWindowListener() method registers listener as an object interested
in being notified when an WindowEvent passes through the EventQueue with
this Window as its target. When such an event occurs, one of the methods in
the WindowListener interface is called. Multiple listeners can be registered.

public void removeWindowListener(WindowListener listener) ★
The removeWindowListener() method removes listener as an interested lis-
tener. If listener is not registered, nothing happens.

protected void processEvent(AWTEvent e) ★
The processEvent() method receives every AWTEvent with this Window as its
target. processEvent() then passes them along to any listeners for processing.
When you subclass Window, overriding processEvent() allows you to process
all events yourself, before sending them to any listeners. In a way, overriding
processEvent() is like overriding handleEvent() using the 1.0 event model.

If you override processEvent(), remember to call super.processEvent(e) last
to ensure that regular event processing can occur. If you want to process your
own events, it’s a good idea to call enableEvents() (inherited from Component) to
ensure that events are delivered even in the absence of registered listeners.

protected void processWindowEvent(WindowEvent e) ★
The processWindowEvent() method receives every WindowEvent with this Win-
dow as its target. processWindowEvent() then passes them along to any listeners
for processing. When you subclass Window, overriding processWindow-
Event() allows you to process all events yourself, before sending them to any
listeners. In a way, overriding processWindowEvent() is like overriding han-
dleEvent() using the 1.0 event model.
If you override `processWindowEvent()`, you must remember to call `super.processWindowEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it’s a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

### 6.5 Frames

The `Frame` is a special type of `Window` that looks like other high level programs in your windowing environment. It adds a `MenuBar`, window title, and window gadgets (like resize, maximize, minimize, window menu) to the basic `Window` object. All the menu-related pieces are discussed in Chapter 10, *Would You Like to Choose from the Menu?*

The default layout manager for a `Frame` is `BorderLayout`.

#### 6.5.1 Frame Constants

The `Frame` class includes a number of constants used to specify cursors. These constants are left over from Java 1.0 and maintained for compatibility. In Java 1.1, you should use the new `Cursor` class, introduced in the previous chapter, and the `Component.setCursor()` method to change the cursor over a frame. Avoid using the `Frame` constants for new code. To see these cursors, refer to Figure 5-6.

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

**NOTE**  
`HAND_CURSOR` and `MOVE_CURSOR` are not available on Windows platforms with Java 1.0. If you ask to use these and they are not available, you get `DEFAULT_CURSOR`. 
6.5.2 Frame Constructors

public Frame ()

The constructor for Frame creates a hidden window with a window title of “Untitled” (Java1.0) or an empty string (Java1.1). Like Window, the default LayoutManager of a Frame is BorderLayout. DEFAULT_CURSOR is the initial cursor. To position the Frame on the screen, call Component.move(). Since the Frame is initially hidden, you need to call the show() method before the user sees the Frame.

public Frame (String title)

This version of Frame’s constructor is identical to the first but sets the window title to title. Figure 6-3 shows the results of a call to new Frame("My Frame") followed by resize() and show().

Figure 6–3: A typical Frame

6.5.3 Frame Methods

public String getTitle ()

The getTitle() method returns the current title for the Frame. If there is no title, this method returns null.

public void setTitle (String title)

The setTitle() method changes the Frame’s title to title.

public Image getIconImage ()

The getIconImage() method returns the image used as the icon. Initially, this returns null. For some platforms, the method should not be used because the platform does not support the concept.
public void setIconImage (Image image)
The setIconImage() method changes the image to display when the Frame is
iconified to image. Not all platforms utilize this resource.

public MenuBar getMenuBar ()
The getMenuBar() method retrieves the Frame’s current menu bar.

public synchronized void setMenuBar (MenuBar bar)
The setMenuBar() method changes the menu bar of the Frame to bar. If bar is
null, it removes the menu bar so that none is available. It is possible to have
multiple menu bars based upon the context of the application. However, the
same menu bar cannot appear on multiple frames and only one can appear at
a time. The MenuBar class, and everything to do with menus, is covered in
Chapter 10.

public synchronized void remove (MenuComponent component)
The remove() method removes component from Frame if component is the
frame’s menu bar. This is equivalent to calling setMenuBar() with a parameter
of null and in actuality is what remove() calls.

public synchronized void dispose ()
The dispose() method frees up the system resources used by the Frame. If
any Dialogs or Windows are associated with this Frame, their resources are
freed, too. Some people like to call Component.hide() before calling the dis-
pose() method so users do not see the frame decomposing.

public boolean isResizable ()
The isResizable() method will tell you if the current Frame is resizable.

public void setResizable (boolean resizable)
The setResizable() method changes the resize state of the Frame. A resiz-
able value of true means the user can resize the Frame, false means the user
cannot. This must be set before the Frame is shown or the peer created.

public void setCursor (int cursorType)
The setCursor() method changes the cursor of the Frame to cursorType.
cursorType must be one of the cursor constants provided with the Frame class.
You cannot create your own cursor image yet. When changing from the
DEFAULT_CURSOR to another cursor, the mouse must be moved for the cursor
icon to change to the new cursor. If cursorType is not one of the predefined
cursor types, setCursor() throws the IllegalArgumentException run-time
exception.

This method has been replaced by the Component.setCursor() method. Both
function equivalently, but this method is being phased out.
public int getCursorType ()

The getCursorType() method retrieves the current cursor.

This method has been replaced by the Component.getCursor() method. Both function equivalently, but this method is being phased out.

Miscellaneous methods

public synchronized void addNotify ()

The addNotify() method creates the Frame peer. This is automatically done when you call the show() method of the Frame. If you override this method, first call super.addNotify(), then add your customizations for the new class. Then you can do everything you need to do with the information about the newly created peer.

protected String paramString ()

When you call the toString() method of Frame, the default toString() method of Component is called. This in turn calls paramString(), which builds up the string to display. At the Frame level, paramString() appends resizable (if true) and the title (if present). Using the default Frame constructor, the results would be:

java.awt.Frame[0,0,0x0,invalid,hidden,layout=java.awt.BorderLayout, resizable,title=]

Until the Frame is shown, via show(), the position and size are not known and therefore appear as zeros. After showing the Frame, you might see:

java.awt.Frame[44,44,300x300,layout=java.awt.BorderLayout, resizable,title=]

6.5.4 Frame Events

In Java 1.0, a Frame peer generates all the events that are generated by the Component class; it does not generate events that are specific to a particular type of component. That is, it generates key events, mouse events, and focus events; it doesn’t generate action events or list events. If an event happens within a child component of a Frame, the target of the event is the child component, not the Frame.

Window

In addition to the Component events, Frame generates the WINDOW events. These events are WINDOW_DESTROY, WINDOW_EXPOSE, WINDOW_ICONIFY, WINDOW_DEICONIFY, and WINDOW_MOVED.
One common event, `WINDOW_DESTROY`, is generated when the user tries to close the Frame by selecting Quit, Close, or Exit (depending on your windowing environment) from the window manager’s menu. By default, this event does nothing. You must provide an event handler that explicitly closes the Frame. If you do not, your Frame will close only when the Java Virtual Machine exits—for example, when you quit Netscape Navigator. The `handleEvent()` method in the following example, or one like it, should therefore be included in all classes that extend Frame. If a `WINDOW_DESTROY` event occurs, it gets rid of the Frame and exits the program. Make sure your method calls `super.handleEvent()` to process the other events.

```java
public boolean handleEvent (Event e) {
    if (e.id == Event.WINDOW_DESTROY) {
        hide();
        dispose();
        System.exit(0); // boolean method, must return something
        return true;
    } else {
        // handle other events we find interesting
    }
    // make sure normal event processing happens
    return super.handleEvent (e);
}
```

### Listeners and 1.1 event handling

With the 1.1 event model, you register listeners for different event types; the listeners are told when the event happens. The Frame class inherits all its listener handling from Window.

Here’s the Java 1.1 code necessary to handle `WINDOW_CLOSING` events; it is equivalent to the `handleEvent()` method in the previous example. First, you must add the following line to the Frame’s constructor:

```java
enableEvents (AWTEvent.WINDOW_EVENT_MASK);
```

This line guarantees that we will receive window events, even if there is no listener. The `processWindowEvent()` method in the following code does the actual work of closing things down:

```java
// Java 1.1 only
protected void processWindowEvent (WindowEvent e) {
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
        // Notify others we are closing
        if (windowListener != null)
            windowListener.windowClosing(e);
        System.exit(0);
    } else {
        super.processEvent(e);
    }
}
```
If you forget to enable events, `processWindowEvent()` may never be called, and your windows will not shut down until the Java Virtual Machine exits. All subclasses of `Frame` should include code like this to make sure they terminate gracefully.

### 6.5.5 Building a New Component from a Window

Now that we have discussed the `Frame` and `Window` objects, we can briefly investigate some ways to use them together. Previously I said that you can use a `Window` to build your own pop-up menu. That’s no longer necessary in Java 1.1, but the same techniques apply to plenty of other objects. In the following example, we build a set of pop-up buttons; it also uses the `Toolkit` of a `Frame` to load images within an application. The pop-up button set appears when the user presses the right mouse button over the image. It is positioned at the coordinates of the `mouseDown()` event; to do so, we add the current `location()` of the `Frame` to the mouse’s `x` and `y` coordinates. Figure 6-4 shows what this application looks like when the pop-up button set is on the screen.

```java
import java.awt.*;
public class PopupButtonFrame extends Frame {
    Image im;
    Window w = new PopupWindow (this);
    PopupButtonFrame () {
        super ("PopupButton Example");
        resize (250, 100);
        show();
        im = getToolkit().getImage ("rosey.jpg");
        MediaTracker mt = new MediaTracker (this);
        mt.addImage (im, 0);
        try {
            mt.waitForAll();
            w.move (location().x+x, location().y+y);
            w.show();
            return true;
        } catch (Exception e) {e.printStackTrace(); }
    }
    public static void main (String args[]) {
        Frame f = new PopupMenuFrame ();
    }
    public void paint (Graphics g) {
        if (im != null)
            g.drawImage (im, 20, 20, this);
    }
    public boolean mouseDown (Event e, int x, int y) {
        if (e.modifiers == Event.META_MASK) {
            w.move (location().x|x, location().y+y);
            w.show();
            return true;
        }
        return false;
    }
}

class PopupWindow extends Window {
    PopupWindow (Frame f) {
```
super (f);
Panel p = new Panel ()
    .add (new Button ("About"));
    .add (new Button ("Save"));
    .add (new Button ("Quit"));
    .add ("North", p);
    .setBackground(Color.gray);
    .pack();}

public boolean action (Event e, Object o) {
    if ("About".equals (o))
        System.out.println ("About");
    else if ("Save".equals (o))
        System.out.println ("Save Me");
    else if ("Quit".equals (o))
        System.exit (0);
    hide();
    return true;
}
playing with the window's insets if you want, but that usually isn't necessary. The class PopupWindow has an action() method that is called when the user clicks one of the buttons. When the user clicks on a button, action() prints a message and hides the window.

6.6 Dialogs

The Dialog class provides a special type of display window that is normally used for pop-up messages or input from the user. It should be associated with a Frame (a required parameter for the constructor), and whenever anything happens to this Frame, the same thing will happen to the Dialog. For instance, if the parent Frame is iconified, the Dialog disappears until the Frame is de-iconified. If the Frame is destroyed, so are all the associated dialogs. Figure 6-5 and Figure 6-6 show typical dialog boxes.

In addition to being associated with a Frame, Dialog is either modeless or modal. A modeless Dialog means a user can interact with both the Frame and the Dialog at the same time. A modal Dialog is one that blocks input to the remainder of the application, including the Frame, until the Dialog box is acted upon. Note that the parent Frame is still executing; unlike some windowing systems, Java does not suspend the entire application for a modal Dialog. Normally, blocking access would be done to get input from the user or to show a warning message. Example 6-2 shows how to create and use a modal Dialog box, as we will see later in the chapter.

Since Dialog subclasses Window, its default LayoutManager is BorderLayout.

In applets, when you create a Dialog, you need to provide a reference to the browser's Frame, not the applet. In order to get this, you can try to go up the container hierarchy of the Applet with getParent() until it returns null. (You cannot specify a null parent as you can with a Window.) See Example 6-1 for a utility method to do this. Simple include a line like the following in your applet:

```
Frame top = ComponentUtilities.getTopLevelParent (this);
```

Then pass top to the Dialog constructor. Another alternative is to create a new Frame to associate with your dialog.
6.6.1 Dialog Constructors and Methods

Constructors

If any constructor is passed a null parent, the constructor throws the run-time exception IllegalArgumentException.

`public Dialog (Frame parent)`★

This constructor creates an instance of Dialog with no title and with parent as the Frame owning it. It is not modal and is initially resizable.

`public Dialog (Frame parent, boolean modal)`✩

This constructor creates an instance of Dialog with no title and with parent as the Frame owning it. If modal is true, the Dialog grabs all the user input of the program until it is closed. If modal is false, there is no special behavior associated with the Dialog. Initially, the Dialog will be resizable. This constructor is comment-flagged as deprecated.

`public Dialog (Frame parent, String title)`★

This version of the constructor creates an instance of Dialog with parent as the Frame owning it and a window title of title. It is not modal and is initially resizable.

`public Dialog (Frame parent, String title, boolean modal)`

This version of the constructor creates an instance of Dialog with parent as the Frame owning it and a window title of title. If modal is true, the Dialog grabs all the user input of the program until it is closed. If modal is false, there is no special behavior associated with the Dialog. Initially, the Dialog will be resizable.

**NOTE**

In some 1.0 versions of Java, modal dialogs were not supported properly. You needed to create some multithreaded contraption that simulated modality. Modal dialogs work properly in 1.1.

Appearance methods

`public String getTitle ()`

The getTitle() method returns the current title for the Dialog. If there is no title for the Dialog, getTitle() returns null.

`public void setTitle (String title)`

The setTitle() method changes the current title of the Dialog to title. To turn off any title for the Dialog, use null for title.
public boolean isResizable()

The `isResizable()` method tells you if the current Dialog is resizable.

public void setResizable(boolean resizable)

The `setResizable()` method changes the resize state of the Dialog. A resizable value of `true` means the user can resize the Dialog, while `false` means the user cannot. This must be set before the Dialog is shown or the peer created.

**Modal methods**

public boolean isModal()

The `isModal()` method returns the current mode of the Dialog. `true` indicates the dialog traps all user input.
public void setModal (boolean mode) ★
The `setModal()` method changes the current mode of the `Dialog` to `mode`. The next time the dialog is displayed via `show()`, it will be modal. If the dialog is currently displayed, `setModal()` has no immediate effect. The change will take place the next time `show()` is called.

public void show () ★
The `show()` method brings the `Dialog` to the front and displays it. If the dialog is modal, `show()` takes care of blocking events so that they don’t reach the parent `Frame`.

Miscellaneous methods
public synchronized void addNotify ()
The `addNotify()` method creates the `Dialog` peer. The peer is created automatically when you call the dialog’s `show()` method. If you override the method `addNotify()`, first call `super.addNotify()`, then add your customizations for the new class. You will then be able to do everything you need with the information about the newly created peer.

protected String paramString ()
When you call the `toString()` method of `Dialog`, the default `toString()` method of `Component` is called. This in turn calls `paramString()` which builds up the string to display. At the `Dialog` level, `paramString()` appends the current mode (modal/modeless) and title (if present). Using the constructor `Dialog(top, "Help", true)`, the results would be as follows:

```
java.awt Dialog[0,0,0x0,invalid,hidden,layout=java.awt.BorderLayout,
modal,title=Help]
```

6.6.2 Dialog Events
In Java 1.0, a `Dialog` peer generates all the events that are generated by the `Component` class; it does not generate events that are specific to a particular type of component. That is, it generates key events, mouse events, and focus events; it doesn’t generate action events or list events. If an event happens within a child component of a `Dialog`, the target of the event is the child component, not the `Dialog`.

Window
In addition to the `Component` events, `Dialog` generates the `WINDOW` events. These events are `WINDOW_DESTROY`, `WINDOW_EXPOSE`, `WINDOW_ICONIFY`, `WINDOW_DEICONIFY`, and `WINDOW_MOVED`. 
Listeners and 1.1 event handling

With the 1.1 event model, you register listeners for different event types; the listeners are told when the event happens. The Dialog class inherits all its listener handling from Window.

6.6.3 Dialog Example

Example 6-2 demonstrates how a modal Dialog tries to work in Java 1.0. In some windowing systems, “modal” means that the calling application, and sometimes the entire system stops, and input to anything other than the Dialog is blocked. With Java 1.0, a modal Dialog acts only on the parent frame and simply prevents it from getting screen-oriented input by disabling all components within the frame. The Java program as a whole continues to execute.

Example 6-2 displays a Dialog window with username and password fields, and an Okay button. When the user selects the Okay button, a realistic application would validate the username and password; in this case, they are just displayed on a Frame. Since the Frame must wait for the Dialog to finish before looking at the values of the two fields, the Dialog must tell the Frame when it can look. This is done through a custom interface implemented by the parent Frame and invoked by the Dialog in its action method.

Figure 6-7 is the initial Dialog; Figure 6-8 shows the result after you click Okay. Example 6-2 contains the source code.

Notice the use of the newly created DialogHandler interface when the user selects the Okay button. Also, see how the pre- and post-event-handling methods are separated. All the pre-event processing takes place before the Dialog is shown. The post-event processing is called by the Dialog through the new DialogHandler interface method, dialogDoer(). The interface provides a common method name for all your Dialog boxes to call.
import java.awt.*;
interface DialogHandler {
    void dialogDoer (Object o);
}
class modeTest extends Dialog {
    TextField user;
    TextField pass;
    modeTest (DialogHandler parent) {
        super ((Frame)parent, "Mode Test", true);
        add ("North", new Label ("Please enter username/password"));
        Panel left = new Panel ();
        left.setLayout (new BorderLayout ());
        left.add ("North", new Label ("Username"));
        left.add ("South", new Label ("Password"));
        add ("West", left);
        Panel right = new Panel ();
        right.setLayout (new BorderLayout ());
        user = new TextField (15);
        pass = new TextField (15);
        pass.setEchoCharacter ('*');
        right.add ("North", user);
        right.add ("South", pass);
        add ("East", right);
        add ("South", new Button ("Okay"));
        resize (250, 125);
    }
    public boolean handleEvent (Event e) {
        if (e.id == Event.WINDOW_DESTROY) {
            dispose();
            return true;
        } else if ((e.target instanceof Button) &&
            (e.id == Event.ACTION_EVENT)) {
            ((DialogHandler)getParent ()).dialogDoer(e.arg);
        }
        return super.handleEvent (e);
    }
}
public class modeFrame extends Frame implements DialogHandler {
    modeTest d;
    modeFrame (String s) {
        super (s);
    }
}
Example 6–2: Modal Dialog Usage (continued)

    resize (100, 100);
    d = new modeTest (this);
    d.show ();
}

public static void main (String []args) {
    Frame f = new modeFrame ("Frame");
}

public boolean handleEvent (Event e) {
    if (e.id == Event.WINDOW_DESTROY) {
        hide();
        dispose();
        System.exit (0);
    }
    return super.handleEvent (e);
}

public void dialogDoer(Object o) {
    d.dispose();
    add ("North", new Label (d.user.getText()));
    add ("South", new Label (d.pass.getText()));
    show ();
}

Since the Java 1.1 modal Dialog blocks the calling Frame appropriately, the overhead of the DialogHandler interface is not necessary and all the work can be combined into the main() method, as shown in the following:

    // only reliable in Java 1.1
    import java.awt.*;
    class modeTest11 extends Dialog {
        TextField user;
        TextField pass;
        modeTest11 (Frame parent) {
            super (parent, "Mode Test", true);
            add ("North", new Label ("Please enter username/password"));
            Panel left = new Panel ();
            left.setLayout (new BorderLayout ());
            left.add ("North", new Label ("Username"));
            left.add ("South", new Label ("Password"));
            add ("West", left);
            Panel right = new Panel ();
            right.setLayout (new BorderLayout ());
            user = new TextField (15);
            pass = new TextField (15);
            pass.setEchoCharacter ("*");
            right.add ("North", user);
            right.add ("South", pass);
            add ("East", right);
            add ("South", new Button ("Okay"));
            resize (250, 125);
        }

        public boolean handleEvent (Event e) {
            //
if (e.id == Event.WINDOW_DESTROY) {
    dispose();
    return true;
} else if ((e.target instanceof Button) &&
    (e.id == Event.ACTION_EVENT)) {
    hide();
}
return super.handleEvent (e);
}

public class modeFrame11 extends Frame {
    modeFrame11 (String s) {
        super (s);
        resize (100, 100);
    }
    public static void main (String [] args) {
        Frame f = new modeFrame11 ("Frame");
        modeTest11 d;
        d = new modeTest11 (f);
        d.show ();
        d.dispose();
        f.add ("North", new Label (d.user.getText()));
        f.add ("South", new Label (d.pass.getText()));
        f.show ();
    }
    public boolean handleEvent (Event e) {
        if (e.id == Event.WINDOW_DESTROY) {
            hide();
            dispose();
            System.exit (0);
        }
        return super.handleEvent (e);
    }
}

The remainder of the code is virtually identical. The most significant difference is that the dialog's handleEvent() method just hides the dialog, rather than calling DialogHandler.dialogDoer().

### 6.7 FileDialog

FileDialog is a subclass of Dialog that lets the user select files for opening or saving. You must load or save any files yourself. If used in an application or appletviewer, the FileDialog always looks like the local system's file dialog. The FileDialog is always a modal Dialog, meaning that the calling program is blocked from continuing (and cannot accept input) until the user responds to the FileDialog. Figure 6-9 shows the FileDialog component in Motif, Windows NT/95, and the Macintosh.
Unlike the other Window subclasses, there is no LayoutManager for FileDialog, since you are creating the environment's actual file dialog. This means you cannot subclass FileDialog to alter its behavior or appearance. However, the class is not "final."

NOTE Netscape Navigator throws an AWTError when you try to create a FileDialog because Navigator does not permit local file system access.

### 6.7.1 FileDialog Methods

#### Constants

A FileDialog has two modes: one for loading a file (input) and one for saving (output). The following variables provide the mode to the constructor. The FileDialog functions the same way in both modes. The only visible difference is whether a button on the screen is labeled Load or Save. You must load or save the requested file yourself. On certain platforms there may be functional differences: in SAVE mode, the FileDialog may ask if you want to replace a file if it already exists; in LOAD mode, the FileDialog may not accept a filename that does not exist.

- `public final static int LOAD
  LOAD is the constant for load mode. It is the default mode.
- `public final static int SAVE
  SAVE is the constant for save mode.

#### Constructors

- `public FileDialog (Frame parent) ★
  The first constructor creates a FileDialog for loading with a parent Frame of parent. The window title is initially empty.
- `public FileDialog (Frame parent, String title)
  This constructor creates a FileDialog for loading with a parent Frame of parent. The window title is title.
- `public FileDialog (Frame parent, String title, int mode)
  The final constructor creates a FileDialog with an initial mode of mode. If mode is neither LOAD nor SAVE, the FileDialog is in SAVE mode.
Appearance methods

`public String getDirectory()`

getDirectory() returns the current directory for the FileDialog. Normally, you check this when FileDialog returns after a `show()` and a call to `getFileDialog()` returns something other than null.

`public void setDirectory(String directory)`

The `setDirectory()` method changes the initial directory displayed in the FileDialog to directory. You must call `setDirectory()` prior to displaying the FileDialog.

`public String getFile()`

The `getFile()` method returns the current file selection from the FileDialog. If the user pressed the Cancel button on the FileDialog, `getFile()` returns null. This is the only way to determine if the user pressed Cancel.

NOTE

On some platforms in Java 1.0 `getFile()` returns a string that ends in `.*.*` (two periods and two asterisks) if the file does not exist. You need to remove the extra characters before you can create the file.
public void setFile (String file)
The `setFile()` method changes the default file for the `FileDialog` to file. Because the `FileDialog` is modal, this must be done before you call `show()`. The string may contain a filename filter like `*.java` to show a preliminary list of files to select. This has nothing to do with the use of the `FilenameFilter` class.

public FilenameFilter getFilenameFilter ()
The `getFilenameFilter()` method returns the current `FilenameFilter`. The `FilenameFilter` class is part of the `java.io` package. `FilenameFilter` is an interface that allows you to restrict choices to certain directory and filename combinations. For example, it can be used to limit the user to selecting `.jpg`, `.gif`, and `.xbm` files. The class implementing `FilenameFilter` would not return other possibilities as choices.

public void setFilenameFilter (FilenameFilter filter)
The `setFilenameFilter()` method changes the current filename filter to `filter`. This needs to be done before you `show()` the `FileDialog`.

**NOTE**
The JDK does not support the `FilenameFilter` with `FileDialog` boxes. `FilenameFilter` works but can't be used with `FileDialog`.

---

**Miscellaneous methods**

public int getMode ()
The `getMode()` method returns the current mode of the `FileDialog`. If an invalid mode was used in the constructor, this method returns an invalid mode here. No error checking is performed.

public void setMode (int mode) ★
The `setMode()` method changes the current mode of the `FileDialog` to `mode`. If `mode` is not one of the class constants `LOAD` or `SAVE`, `setMode()` throws the run-time exception `IllegalArgumentException`.

public synchronized void addNotify ()
The `addNotify()` method creates the `FileDialog` peer. This is automatically done when you call the `show()` method of the `FileDialog`. If you override this method, first call `super.addNotify()`, then add your customizations for the new class. Then you can do everything you need with the information about the newly created peer.
protected String paramString ()

When you call the toString() method of FileDialog, the default toString() method of Component is called. This in turn calls paramString(), which builds up the string to display. At the FileDialog level, paramString() appends the directory (if not null) and current mode to the return value. Using the constructor FileDialog(top, "Load Me"), the results would be as follows:

```
java.awt.FileDialog[0,0,0x0,invalid,hidden,modal,title=Load Me,load]
```

### 6.7.2 A FileDialog Example

To get a better grasp of how the FileDialog works, the following application uses a FileDialog to select a file for display in a TextArea. You can also use FileDialog to save the file back to disk. Figure 6-10 shows the application, with a file displayed in the text area; the FileDialog itself looks like any other file dialog on the runtime system. Example 6-3 shows the code.

**CAUTION** This example can overwrite an existing file.
Example 6-3: Complete FileDialog (continued)

```java
Button loadButton;
Button saveButton;
FDTest () {
    super ("File Dialog Tester");
    Panel p = new Panel ();
    p.add (loadButton = new Button ("Load"));
    p.add (saveButton = new Button ("Save"));
    add ("North", myLabel = new Label ());
    add ("South", p);
    add ("Center", myTextArea = new TextArea (10, 40));
    Menu m = new Menu ("File");
    m.add (new MenuItem ("Quit"));
    MenuBar mb = new MenuBar();
    mb.add (m);
    setMenuBar (mb);
    pack();
}
public static void main (String args[]) {
    FDTest f = new FDTest();
    f.show();
}
public boolean handleEvent (Event e) {
    if (e.id == Event.WINDOW_DESTROY) {
        hide();
        dispose ();
        System.exit(0);
        return true; // never gets here
    }
    return super.handleEvent (e);
}
public boolean action (Event e, Object o) {
    if (e.target instanceof MenuItem) {
        hide();
        dispose ();
        System.exit(0);
        return true; // never gets here
    } else if (e.target instanceof Button) {
        int state;
        String msg;
        if (e.target == loadButton) {
            state = FileDialog.LOAD;
            msg = "Load File";
        } else if (e.target == saveButton) {
            state = FileDialog.SAVE;
            msg = "Save File";
        }
        FileDialog file = new FileDialog (this, msg, state);
        file.setFile ("*.java"); // set initial filename filter
        file.show(); // Blocks
        String curFile;
        if ((curFile = file.getFile()) != null) {
            String filename = file.getDirectory() + curFile;
```
Example 6–3: Complete FileDialog (continued)

// curFile ends in .*.* if file does not exist
byte[] data;
setCursor (Frame.WAIT_CURSOR);
if (state == FileDialog.LOAD) {
    File f = new File (filename);
    try {
        FileInputStream fin = new FileInputStream (f);
        int filesize = (int)f.length();
        data = new byte[filesize];
        fin.read (data, 0, filesize);
    } catch (FileNotFoundException exc) {
        String errorString = "File Not Found: " + filename;
        data = new byte[errorString.length()];
        errorString.getBytes (0, errorString.length(), data, 0);
    } catch (IOException exc) {
        String errorString = "IOException: " + filename;
        data = new byte[errorString.length()];
        errorString.getBytes (0, errorString.length(), data, 0);
    }
    myLabel.setText ("Load: " + filename);
} else {
    // Remove trailing ".*.*" if present - signifies file does not exist
    if (filename.indexOf (".*.*") != -1) {
        filename = filename.substring (0, filename.length()-4);
    }
    File f = new File (filename);
    try {
        FileOutputStream fout = new FileOutputStream (f);
        String text = myTextArea.getText();
        int textsize = text.length();
        data = new byte[textsize];
        text.getBytes (0, textsize, data, 0);
        fout.write (data);
        fout.close ();
    } catch (IOException exc) {
        String errorString = "IOException: " + filename;
        data = new byte[errorString.length()];
        errorString.getBytes (0, errorString.length(), data, 0);
    }
    myLabel.setText ("Save: " + filename);
}
// Note - on successful save, text is redisplayed
myTextArea.setText (new String (data, 0));
setCursor (Frame.DEFAULT_CURSOR);
return true;
} else {
    return false;
}
Most of this application is one long `action()` method that handles all the action events that take place within the `Frame`. The constructor doesn’t do much besides arrange the display; it includes code to create a File menu with one item, Quit. This menu is visible in the upper left corner of the `Frame`; we’ll see more about working with menus in Chapter 10. We provide a `main()` method to display the `Frame` and a `handleEvent()` method to shut the application down if the event `WINDOW_DESTROY` occurs.

But the heart of this program is clearly its `action()` method. `action()` starts by checking whether the user selected a menu item; if so, it shuts down the application because the only item on our menu is Quit. It then checks whether the user clicked on one of the buttons and sets the `FileDialog` mode to `LOAD` or `SAVE` accordingly. It then sets a default filename, `*.java`, which limits the display to filenames ending in `.java`. Next, `action()` shows the dialog. Because file dialogs are modal, `show()` blocks until the user selects a file or clicks Cancel.

The next line detects whether or not `getFile()` returns `null`. A `null` return indicates that the user selected Cancel; in this case, the dialog disappears, but nothing else happens. We then build a complete filename from the directory name and the name the user selected. If the dialog’s state is `LOAD`, we read the file and display it in the text area. Otherwise, the dialog’s state must be `SAVE`, so we save the contents of the text area under the given filename. Note that we first check for the string `*.*` and remove it if it is present. In Java 1.1, these two lines are unnecessary, but they don’t hurt, either.