
In this chapter:

- *Component*
- *Labels*
- *Buttons*
- *A Simple Calculator*
- *Canvas*
- *Creating Your Own Component*
- *Cursor*

Components

This chapter introduces the generic graphical widget used within the AWT package, `Component`, along with a trio of specific components: `Label`, `Button`, and `Canvas`. It also covers the `Cursor` class, new to Java 1.1. (`Cursor` support was previously part of the `Frame` class.) Although many objects within AWT don't subclass `Component`, and though you will never create an instance of `Component`, anything that provides screen-based user interaction and relies on the system for its layout will be a child of `Component`. As a subclass of `Component`, each child inherits a common set of methods and an API for dealing with the different events (i.e., mouse click, keyboard input) that occur within your Java programs.

After discussing the methods in `Component` classes, this chapter goes into detail about two specific components, `Label` and `Button`. A `Label` is a widget that contains descriptive text, usually next to an input field. A `Button` is a basic mechanism that lets the user signal the desire to perform an action. You will learn about the `Canvas` object and how to use a `Canvas` to create your own component. Finally, we cover the `Cursor` class, which lets you change the cursor over a `Component`.

Before going into the mechanics of the `Component` class, it's necessary to say a little about the relationship between components and containers. A `Container` is also a component with the ability to contain other components. There are several different kinds of containers; they are discussed in Chapter 6, *Containers*. To display a component, you have to put it in a container by calling the container's `add()` method. We often call the container that holds a component the component's *parent*; likewise, we call the components a container holds its *children*. Certain operations are legal only if a component has a parent—that is, the component is in a container. Of course, since containers are components, containers can contain other containers, *ad infinitum*.

NOTE If you think some component is missing a method that should obviously be there, check the methods it inherits. For example, the `Label` class appears to lack a `setFont()` method. Obviously, labels ought to be able to change their fonts. The `setFont()` method really is there; it is inherited from the `Component` class, and therefore, not documented as part of the `Label` class. Even if you're familiar with object-oriented techniques, the need to work up a class hierarchy to find all of the class's methods can lead to confusion and frustration. While all Java objects inherit methods from other classes, the potential for confusion is worst with components, which inherit over a hundred methods from `Component` and may only have a few methods of their own.

5.1 Component

Every GUI-based program consists of a screen with a set of objects. With Java, these objects are called components. Some of the more frequently used components are buttons, text fields, and containers.

A container is a special component that allows you to group different components together within it. You will learn more about containers in the next chapter, but they are in fact just another kind of component. Also, some of the parameters and return types for the methods of `Component` have not been explained yet and have their own sections in future chapters.

5.1.1 Component Methods

Constants

Prior to Java 1.1, you could not subclass `Component` or `Container`. With the introduction of the `LightweightPeer`, you can now subclass either `Component` or `Container`. However, since you no longer have a native peer, you must rely on your container to provide a display area and other services that are normally provided by a full-fledged peer. Because you cannot rely on your peer to determine your alignment, the `Component` class now has five constants to indicate six possible alignment settings (one constant is used twice). The alignment constants designate where to position a lightweight component; their values range from 0.0 to 1.0. The lower the number, the closer the component will be placed to the origin (top left corner) of the space allotted to it.*

* As of Beta 3, these constants appear to be seldom used. The `getAlignmentX()` and `getAlignmentY()` methods return these values, but there are no `setAlignment` methods.

public static final float BOTTOM_ALIGNMENT ★

The `BOTTOM_ALIGNMENT` constant indicates that the component should align itself to the bottom of its available space. It is a return value from the method `getAlignmentY()`.

public static final float CENTER_ALIGNMENT ★

The `CENTER_ALIGNMENT` constant indicates that the component should align itself to the middle of its available space. It is a return value from either the `getAlignmentX()` or `getAlignmentY()` method. This constant represents both the horizontal and vertical center.

public static final float LEFT_ALIGNMENT ★

The `LEFT_ALIGNMENT` constant indicates that the component should align itself to the left side of its available space. It is a return value from `getAlignmentX()`.

public static final float RIGHT_ALIGNMENT ★

The `RIGHT_ALIGNMENT` constant indicates that the component should align itself to the right side of its available space. It is a return value from the method `getAlignmentX()`.

public static final float TOP_ALIGNMENT ★

The `TOP_ALIGNMENT` constant indicates that the component should align itself to the top of its available space. It is a return value from `getAlignmentY()`.

Variables

protected Locale locale ★

The protected `locale` variable can be accessed by calling the `getLocale()` method.

Constructor

Prior to Java 1.1, there was no public or protected constructor for `Component`. Only package members were able to subclass `Component` directly. With the introduction of lightweight peers, components can exist without a native peer, so the constructor was made protected, allowing you to create your own `Component` subclasses.

protected Component() ★

The constructor for `Component` 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 components that require fewer system resources than components that subclass `Canvas`. The example in the “Using an event multicaster” section of the previous chapter is of a lightweight component. Use the `SystemColor` class to help you colorize the new component appropriately or make it transparent.

Appearance

public Toolkit getToolkit ()

The `getToolkit()` method returns the current `Toolkit` of the `Component`. This returns the parent's `Toolkit` (from a `getParent()` call) when the `Component` has not been added to the screen yet or is lightweight. If there is no parent, `getToolkit()` returns the default `Toolkit`. Through the `Toolkit`, you have access to the details of the current platform (like screen resolution, screen size, and available fonts), which you can use to adjust screen real estate requirements or check on the availability of a font.

public Color getForeground ()

The `getForeground()` method returns the foreground color of the component. If no foreground color is set for the component, you get its parent's foreground color. If none of the component's parents have a foreground color set, `null` is returned.

public void setForeground (Color c)

The `setForeground()` method changes the current foreground color of the area of the screen occupied by the component to `c`. After changing the color, it is necessary for the screen to refresh before the change has any effect. To refresh the screen, call `repaint()`.

public Color getBackground ()

The `getBackground()` method returns the background color of the component. If no background color is set for the component, its parent's background color is retrieved. If none of the component's parents have a background color set, `null` is returned.

public void setBackground (Color c)

The `setBackground()` method changes the current background color of the area of the screen occupied by the component to `c`. After changing the color, it is necessary for the screen to refresh before the change has any affect. To refresh the screen, call `repaint()`.

public Font getFont ()

The `getFont()` method returns the font of the component. If no font is set for the component, its parent's font is retrieved. If none of the component's parents have a font set, `null` is returned.

public synchronized void setFont (Font f)

The `setFont()` method changes the component's font to `f`. If the font family (such as `TimesRoman`) provided within `f` is not available on the current platform, the system uses a default font family, along with the supplied size and style (plain, bold, italic). Depending upon the platform, it may be necessary to refresh the component/screen before seeing any changes.

Changing the font of a component could have an affect on the layout of the component.

public synchronized ColorModel getColorModel ()

The `getColorModel()` method returns the `ColorModel` used to display the current component. If the component is not displayed, the `ColorModel` from the component's `Toolkit` is used. The normal `ColorModel` for a Java program is 8 bits each for red, green, and blue.

public Graphics getGraphics ()

The `getGraphics()` method gets the component's graphics context. Most non-container components do not manage them correctly and therefore throw an `InternalError` exception when you call this method. The `Canvas` component is one that does since you can draw on that directly. If the component is not visible, `null` is returned.

public FontMetrics getFontMetrics (Font f)

The `getFontMetrics()` method retrieves the component's view of the `FontMetrics` for the requested font `f`. Through the `FontMetrics`, you have access to the platform-specific sizing for the appearance of a character or string.

public Locale getLocale () ★

The `getLocale()` method retrieves the current `Locale` of the component, 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 parent's `Locale`.* If the component has no locale of its own and no parent (i.e., it isn't in a container), `getLocale()` throws the run-time exception `IllegalComponentStateException`.

public void setLocale (Locale l) ★

The `setLocale()` method changes the current `Locale` of the component to `l`. In order for this change to have any effect, you must localize your components so that they have different labels or list values for different environments. Localization is part of the broad topic of internationalization and is beyond the scope of this book.

public Cursor getCursor () ★

The `getCursor()` method retrieves the component's current `Cursor`. If one hasn't been set, the default is `Cursor.DEFAULT_CURSOR`. The `Cursor` class is described fully in Section 5.7. Prior to Java 1.1, the ability to associate cursors with components was restricted to frames.

* For more on the `Locale` class, see the *Java Fundamental Classes Reference* from O'Reilly & Associates.

public synchronized void setCursor (Cursor c) ★

The `setCursor()` method changes the current `Cursor` of the component to `c`. The change takes effect as soon as the cursor is moved. Lightweight components cannot change their cursors.

Positioning/Sizing

`Component` provides a handful of methods for positioning and sizing objects. Most of these are used behind the scenes by the system. You will also need them if you create your own `LayoutManager` or need to move or size an object. All of these depend on support for the functionality from the true component's peer.

public Point getLocation () ★

public Point location () ☆

The `getLocation()` method returns the current position of the `Component` in its parent's coordinate space. The `Point` is the top left corner of the bounding box around the `Component`.

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

public Point getLocationOnScreen () ★

The `getLocationOnScreen()` method returns the current position of the `Component` in the screen's coordinate space. The `Point` is the top left corner of the bounding box around the `Component`. If the component is not showing, the `getLocationOnScreen()` method throws the `IllegalComponentStateException` run-time exception.

public void setLocation (int x, int y) ★

public void move (int x, int y) ☆

The `setLocation()` method moves the `Component` to the new position (`x`, `y`). The coordinates provided are in the parent container's coordinate space. This method calls `setBounds()` to move the component. The `LayoutManager` of the container may make it impossible to change a component's location.

Calling this method with a new position for the component generates a `ComponentEvent` with the ID `COMPONENT_MOVED`.

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

public void setLocation (Point p) ★

This `setLocation()` method moves the component to the position specified by the given `Point`. It is the same as calling `setLocation(p.x, p.y)`.

Calling this method with a new position for the component generates a `ComponentEvent` with the ID `COMPONENT_MOVED`.

public Dimension getSize () ★

public Dimension size () ☆

The `getSize()` method returns the width and height of the component as a `Dimension` object.

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

public void setSize (int width, int height) ★

public void resize (int width, int height) ☆

The `setSize()` method changes the component's width and height to the width and height provided. width and height are specified in pixels. The component is resized by a call to `setBounds()`. The `LayoutManager` of the `Container` that contains the component may make it impossible to change a component's size.

Calling this method with a new size for the component generates a `Component-Event` with the ID `COMPONENT_RESIZED`.

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

public void setSize (Dimension d) ★

public void resize (Dimension d) ☆

This `setSize()` method changes the component's width and height to the `Dimension d` provided. The `Dimension` object includes the width and height attributes in one object. The component is resized by a call to the `setBounds()` method. The `LayoutManager` of the `Container` that contains the component may make it impossible to change a component's size.

Calling this method with a new size for the component generates a `Component-Event` with the ID `COMPONENT_RESIZED`.

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

public Rectangle getBounds () ★

public Rectangle bounds () ☆

The `getBounds()` method returns the bounding rectangle of the object. The fields of the `Rectangle` that you get back contain the component's position and dimensions.

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

public void setBounds (int x, int y, int width, int height) ★

public void reshape (int x, int y, int width, int height) ☆

The `setBounds()` method moves and resizes the component to the bounding rectangle with coordinates of `(x, y)` (top left corner) and width height. If the size and shape have not changed, no reshaping is done. If the component is resized, it is invalidated, along with its parent container. The `LayoutManager` of

the `Container` that contains the component may make it impossible to change the component's size or position. Calling `setBounds()` invalidates the container, which results in a call to the `LayoutManager` to rearrange the container's contents. In turn, the `LayoutManager` calls `setBounds()` to give the component its new size and position, which will probably be the same size and position it had originally. In short, if a layout manager is in effect, it will probably undo your attempts to change the component's size and position.

Calling this method with a new size for the component generates a `ComponentEvent` with the ID `COMPONENT_RESIZED`. Calling this method with a new position generates a `ComponentEvent` with the ID `COMPONENT_MOVED`.

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

public void setBounds (Rectangle r) ★

This `setBounds()` method calls the previous method with parameters of `r.x`, `r.y`, `r.width`, and `r.height`.

Calling this method with a new size for the component generates a `ComponentEvent` with the ID `COMPONENT_RESIZED`. Calling this method with a new position generates a `ComponentEvent` with the ID `COMPONENT_MOVED`.

public Dimension getPreferredSize () ★

public Dimension preferredSize () ☆

The `getPreferredSize()` method returns the `Dimension` (width and height) for the preferred size of the component. Each component's peer knows its preferred size. Lightweight objects return `getSize()`.

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

public Dimension getMinimumSize () ★

public Dimension minimumSize () ☆

The `getMinimumSize()` method returns the `Dimension` (width and height) for the minimum size of the component. Each component's peer knows its minimum size. Lightweight objects return `getSize()`. It is possible that the methods `getMinimumSize()` and `getPreferredSize()` will return the same dimensions.

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

public Dimension getMaximumSize () ★

The `getMaximumSize()` method returns the `Dimension` (width and height) for the maximum size of the component. This may be used by a layout manager to prevent a component from growing beyond a predetermined size. None of the `java.awt` layout managers call this method. By default, the value returned is `Short.MAX_VALUE` for both dimensions.

public float getAlignmentX () ★

The `getAlignmentX()` method returns the alignment of the component along the x axis. The alignment could be used by a layout manager to position this component relative to others. 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. The default setting is `Component.CENTER_ALIGNMENT`.

public float getAlignmentY () ★

The `getAlignmentY()` method returns the alignment of the component along the y axis. The alignment could be used by a layout manager to position this component relative to others. 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. The default setting is `Component.CENTER_ALIGNMENT`.

public void doLayout () ★

public void layout () ☆

The `doLayout()` method of `Component` does absolutely nothing. It is called when the `Component` is validated (through the `validate()` method). The `Container` class overrides this method.

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

public boolean contains (int x, int y) ★

public boolean inside (int x, int y) ☆

The `contains()` method checks if the x and y coordinates are within the bounding box of the component. If the `Component` is not rectangular, the method acts as if there is a rectangle around the `Component`. `contains()` returns `true` if the x and y coordinates are within the component, `false` otherwise.

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

public boolean contains (Point p) ★

This `contains()` method calls the previous method with parameters of `p.x` and `p.y`.

public Component getComponentAt (int x, int y) ★

public Component locate (int x, int y) ☆

The `getComponentAt()` method uses `contains()` to see if the x and y coordinates are within the component. If they are, this method returns the `Component`. If they aren't, it returns `null`. `getComponentAt()` is overridden by `Container` to provide enhanced functionality.

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

public Component getComponentAt (Point p) ★

This `getComponentAt()` method calls the previous method with parameters of `p.x` and `p.y`.

Painting

The only methods in this section that you call directly are the versions of `repaint()`. The `paint()` and `update()` methods are called by the system when the display area requires refreshing, such as when a user resizes a window. When your program changes the display you should call `repaint()` to trigger a call to `update()` and `paint()`. Otherwise, the system is responsible for updating the display.

public void paint (Graphics g)

The `paint()` method is offered so the system can display whatever you want in a `Component`. In the base `Component` class, this method does absolutely nothing. Ordinarily, it would be overridden in an applet to do something other than the default, which is display a box in the current background color. `g` is the graphics context of the component being drawn on.

public void update (Graphics g)

The `update()` method is automatically called when you ask to repaint the `Component`. If the component is not lightweight, the default implementation of `update()` clears graphics context `g` by drawing a filled rectangle in the background color, resetting the color to the current foreground color, and calling `paint()`. If you do not override `update()` when you do animation, you will see some flickering because `Component` clears the screen. Animation is discussed in Chapter 2, *Simple Graphics*.

public void paintAll (Graphics g)

The `paintAll()` method validates the component and paints its peer if it is visible. `g` represents the graphics context of the component. This method is called when the `paintComponents()` method of `Container` is called.

public void repaint ()

The `repaint()` method requests the scheduler to redraw the component as soon as possible. This will result in `update()` getting called soon thereafter. There is not a one-to-one correlation between `repaint()` and `update()` calls. It is possible that multiple `repaint()` calls can result in a single `update()`.

public void repaint (long tm)

This version of `repaint()` allows for a delay of `tm` milliseconds. It says, please update this component within `tm` milliseconds, which may happen immediately.

public void repaint (int x, int y, int width, int height)

This version of `repaint()` allows you to select the region of the `Component` you desire to be updated. `(x, y)` are the coordinates of the upper left corner of the bounding box of the component with dimensions of `width` `height`. This is similar to creating a clipping area and results in a quicker repaint.

public void repaint (long tm, int x, int y, int width, int height)

This final version of `repaint()` is what the other three `repaint()` methods call. `tm` is the maximum delay in milliseconds before update should be called. `(x, y)` are the coordinates of the upper left corner of the clipping area of the component with dimensions of `width` `height`.

public void print (Graphics g)

The default implementation of the `print()` method calls `paint()`.

In Java 1.0, there was no way to print; in Java 1.1, if the `graphics` parameter implements `PrintGraphics`, anything drawn on `g` will be printed. Printing is covered in Chapter 17, *Printing*.

public void printAll (Graphics g)

The `printAll()` method validates the component and paints its peer if it is visible. `g` represents the graphics context of the component. This method is called when the `printComponents()` method of `Container` is called or when you call it with a `PrintGraphics` parameter.

The default implementation of `printAll()` is identical to `paintAll()`. As with `paintAll()`, `g` represents the graphics context of the component; if `g` implements `PrintGraphics`, it can be printed.

Imaging

Background information about using images is discussed in Chapter 2 and Chapter 12, *Image Processing*. The `imageUpdate()` method of `Component` is the sole method of the `ImageObserver` interface. Since images are loaded in a separate thread, this method is called whenever additional information about the image becomes available.

public boolean imageUpdate (Image image, int infoflags, int x, int y, int width, int height)

`imageUpdate()` is the `java.awt.image.ImageObserver` method implemented by `Component`. It is an asynchronous update interface for receiving

notifications about `Image` information as `image` is loaded and is automatically called when additional information becomes available. This method is necessary because image loading is done in a separate thread from the `getImage()` call. Ordinarily, `x` and `y` would be the coordinates of the upper left corner of the image loaded so far, usually (0, 0). However, the method `imageUpdate()` of the component ignores these parameters. `width` and `height` are the `image`'s dimensions, so far, in the loading process.

The `infoflags` parameter is a bit-mask of information available to you about `image`. Please see the text about `ImageObserver` in Chapter 12 for a complete description of the different flags that can be set. When overriding this method, you can wait for some condition to be true by checking a flag in your program and then taking the desired action. To check for a particular flag, perform an AND (&) of `infoflags` and the constant. For example, to check if the `FRAMEBITS` flag is set:

```
if ((infoflags & ImageObserver.FRAMEBITS) == ImageObserver.FRAMEBITS)
    System.out.println ("The Flag is set");
```

The return value from a call to `imageUpdate()` is `true` if `image` has changed and `false` otherwise.

Two system properties let the user control the behavior of updates:

- `awt.image.incrementaldraw` allows the user to control whether or not partial images are displayed. Initially, the value of `incrementaldraw` is unset and defaults to `true`, which means that partial images are drawn. If `incrementaldraw` is set to `false`, the image will be drawn only when it is complete or when the screen is resized or refreshed.
- `awt.image.redrawrate` allows the user to change the delay between successive repaints. If not set, the default redraw rate is 100 milliseconds.

public Image createImage (int width, int height)

The `createImage()` method creates an empty `Image` of size `width` `height`. The returned `Image` is an in-memory image that can be drawn on for double buffering to manipulate an image in the background. If an image of size `width` `height` cannot be created, the call returns `null`. In order for `createImage()` to succeed, the peer of the `Component` must exist; if the component is lightweight, the peer of the component's container must exist.

public Image createImage (ImageProducer producer)

This `createImage()` method allows you to take an existing image and modify it in some way to produce a new `Image`. This can be done through `ImageFilter` and `FilteredImageSource` or a `MemoryImageSource`, which accepts an array of pixel information. You can learn more about these classes and this method in Chapter 12.

public boolean prepareImage (Image image, ImageObserver observer)

The `prepareImage()` method forces `image` to start loading, asynchronously, in another thread. `observer` is the `Component` that `image` will be rendered on and is notified (via `imageUpdate()`) as `image` is being loaded. In the case of an `Applet`, this would be passed as the `ImageObserver`. If `image` has already been fully loaded, `prepareImage()` returns `true`. Otherwise, `false` is returned. Since `image` is loaded asynchronously, `prepareImage()` returns immediately. Ordinarily, `prepareImage()` would be called by the system when `image` is first needed to be displayed (in `drawImage()` within `paint()`). As more information about the image gets loaded, `imageUpdate()` is called periodically.

If you do not want to go through the trouble of creating a `MediaTracker` instance to start the loading of the image objects, you can call `prepareImage()` to trigger the start of image loading prior to a call to `drawImage()`.

If `image` has already started loading when this is called or if this is an in-memory image, there is no effect.

public boolean prepareImage (Image image, int width, int height, ImageObserver observer)

This version of `prepareImage()` is identical to the previous one, with the addition of a scaling factor of `widthheight`. As with other `width` and `height` parameters, the units for these parameters are pixels. Also, if `width` and `height` are `-1`, no scaling factor is assumed. This method is called by one of the internal `MediaTracker` methods.

public int checkImage (Image image, ImageObserver observer)

The `checkImage()` method returns the status of the construction of a screen representation of `image`, being watched by `observer`. If `image` has not started loading yet, this will not start it. The return value is the `ImageObserver` flags ORed together for the data that is now available. The available `ImageObserver` flags are: `WIDTH`, `HEIGHT`, `PROPERTIES`, `SOMEBITS`, `FRAMEBITS`, `ALLBITS`, `ERROR`, and `ABORT`. See Chapter 12 for a complete description of `ImageObserver`.

public int checkImage (Image image, int width, int height, ImageObserver observer)

This version of `checkImage()` is identical to the previous one, with the addition of a scaling factor of `widthheight`. If you are using the `drawImage()` version with `width` and `height` parameters, you should use this version of `checkImage()` with the same `width` and `height`.

Peers

public ComponentPeer getPeer () ☆

The `getPeer()` method returns a reference to the component's peer as a `ComponentPeer` object. For example, if you issue this method from a `Button` object, `getPeer()` returns an instance of the `ComponentPeer` subclass `ButtonPeer`.

This method is flagged as deprecated in comments but not with `@deprecated`. There is no replacement method for Java 1.1.

public void addNotify ()

The `addNotify()` method is overridden by each individual component type. When `addNotify()` is called, the peer of the component gets created, and the `Component` is invalidated. The `addNotify()` method is called by the system when it needs to create the peer. The peer needs to be created when a `Component` is first shown, or when a new `Component` is added to a `Container` and the `Container` is already being shown (in which case it already has a peer, but a new one must be created to take account of the new `Component`). If you override this method for a specific `Component`, call `super.addNotify()` first, then do what you need for the `Component`. You will then have information available about the newly created peer.

Certain tasks cannot succeed unless the peer has been created. An incomplete list includes finding the size of a component, laying out a container (because it needs the component's size), and creating an `Image` object. Peers are discussed in more depth in Chapter 15, *Toolkit and Peers*.

public synchronized void removeNotify ()

The `removeNotify()` method destroys the peer of the component and removes it from the screen. The state information about the `Component` is retained by the specific subtype. The `removeNotify()` method is called by the system when it determines the peer is no longer needed. Such times would be when the `Component` is removed from a `Container`, when its container changes, or when the `Component` is disposed. If you override this method for a specific `Component`, issue the particular commands for you need for this `Component`, then call `super.removeNotify()` last.

State Procedures

These methods determine whether the component is ready to be displayed and can be seen by the user. The first requirement is that it be *valid*—that is, whether the system knows its size, and (in the case of a container) whether the layout manager is aware of all its parts and has placed them as requested. A component becomes invalid if the size has changed since it was last displayed. If the component is a container, it becomes invalid when one of the components contained within it becomes invalid.

Next, the component must be *visible*—a possibly confusing term, because components can be considered “visible” without being seen by the user. Frames (because they have their own top-level windows) are not visible until you request that they be shown, but other components are visible as soon as you create them.

Finally, to be seen, a component must be *showing*. You show a component by adding it to its container. For something to be showing, it must be visible and be in a container that is visible and showing.

A subsidiary aspect of state is the *enabled* quality, which determines whether a component can accept input.

public boolean isValid ()

The `isValid()` method tells you whether or not the component needs to be laid out.

public void validate ()

The `validate()` method sets the component's valid state to `true`. Ordinarily, this is done for you when the `Component` is laid out by its `Container`. Since objects are invalid when they are first drawn on the screen, you should call `validate()` to tell the system you are finished adding objects so that it can validate the screen and components. One reason you can override `validate()` is to find out when the container that the component exists in has been resized. The only requirement when overriding is that the original `validate()` be called. With Java 1.1, instead of overriding, you can listen for resize events.

public void invalidate ()

The `invalidate()` method sets the component's valid state to `false` and propagates the invalidation to its parent. Ordinarily, this is done for you, or should be, whenever anything that affects the layout is changed.

public boolean isVisible ()

The `isVisible()` methods tells you if the component is currently visible. Most components are initially visible, except for top-level objects like frames. Any component that is visible will be shown on the screen when the screen is painted.

public boolean isShowing ()

The `isShowing()` method tells you if the component is currently shown on the screen. It is possible for `isVisible()` to return `true` and `isShowing()` to return `false` if the screen has not been painted yet.

Table 5-1 compares possible return values from `isVisible()` and `isShowing()`. The first two entries are for objects that have their own `Window`. These will always return the same values for `isVisible()` and `isShowing()`. The next three are for `Component` objects that exist within a `Window`, `Panel`, or `Applet`. The visible setting is always initially `true`. However, the showing setting is not `true` until the object is actually drawn. The last case shows another possibility. If the component exists within an invisible `Container`, the component will be visible but will not be shown.

Table 5-1: *isVisible* vs. *isShowing*

Happenings	isVisible	isShowing
Frame created Frame f = new Frame ()	false	false
Frame showing f.show ()	true	true
Component created Button b= new Button ("Help")	true	false
Button added to screen in init() add (b)	true	false
Container laid out with Button in it	true	true
Button within Panel that is not visible	true	false

public void show ()

The `show()` method displays a component by making it visible and showing its peer. The parent `Container` becomes invalid because the set of children to display has changed. You would call `show()` directly to display a `Frame` or `Dialog`.

In Java 1.1, you should use `setVisible()` instead.

public void hide ()

The `hide()` method hides a component by making it invisible and hiding its peer. The parent `Container` becomes invalid because the set of children to display has changed. If you call `hide()` for a `Component` that does not subclass `Window`, the component's `Container` reserves space for the hidden object.

In Java 1.1, you should use `setVisible()` instead.

public void setVisible(boolean condition) ★

public void show (boolean condition) ☆

The `setVisible()` method calls either `show()` or `hide()` based on the value of `condition`. If `condition` is `true`, `show()` is called. When `condition` is `false`, `hide()` is called.

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

public boolean isEnabled ()

The `isEnabled()` method checks to see if the component is currently enabled. An enabled `Component` can be selected and trigger events. A disabled `Component` usually has a slightly lighter font and doesn't permit the user to select or interact with it. Initially, every `Component` is enabled.

public synchronized void enable ()

The `enable()` method allows the user to interact with the component. Components are enabled by default but can be disabled by a call to `disabled()` or `setEnabled(false)`.

In Java 1.1, you should use `setEnabled()` instead.

public synchronized void disable ()

The `disable()` method disables the component so that it is unresponsive to user interactions.

In Java 1.1, you should use `setEnabled()` instead.

public void setEnabled (boolean condition) ★

public void enable (boolean condition) ☆

The `setEnabled()` method calls either `enable()` or `disable()` based on the value of `condition`. If `condition` is `true`, `enable()` is called. When `condition` is `false`, `disable()` is called. Enabling and disabling lets you create components that can be operated only under certain conditions—for example, a `Button` that can be pressed only after the user has typed into a `TextArea`.

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

Focus

Although there was some support for managing input focus in version 1.0, 1.1 improved on this greatly by including support for `Tab` and `Shift+Tab` to move input focus to the next or previous component, and by being more consistent across different platforms. This support is provided by the package-private class `FocusManager`.

public boolean isFocusTraversable() ★

The `isFocusTraversable()` method is the support method that tells you whether or not a component is capable of receiving the input focus. Every component asks its peer whether or not it is traversable. If there is no peer, this method returns `false`.

If you are creating a component by subclassing `Component` or `Canvas` and you want it to be traversable, you should override this method; a `Canvas` is not traversable by default.

public void requestFocus ()

The `requestFocus()` method allows you to request that a component get the input focus. If it can't (`isFocusTraversable()` returns `false`), it won't.

public void transferFocus () ★

public void nextFocus () ☆

The `transferFocus()` method moves the focus from the current component to the next one.

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

Miscellaneous methods

public final Object getTreeLock () ★

The `getTreeLock()` method retrieves the synchronization lock for all AWT components. Instead of using `synchronized` methods in Java 1.1, previously `synchronized` methods lock the tree within a `synchronized (component.getTreeLock()) {}` code block. This results in a more efficient locking mechanism to improve performance.

public String getName () ★

The `getName()` method retrieves the current name of the component. The component's name is useful for object serialization. Components are given a name by default; you can change the name by calling `setName()`.

public void setName (String name) ★

The `setName()` method changes the name of the component to `name`.

public Container getParent ()

The `getParent()` method returns the component's `Container`. The container for anything added to an applet is the applet itself, since it subclasses `Panel`. The container for the applet is the browser. In the case of Netscape Navigator versions 2.0 and 3.0, the return value would be a specific instance of the `netscape.applet.EmbeddedAppletFrame` class. If the applet is running within the *appletviewer*, the return value would be an instance of `sun.applet.AppletViewerPanel`.

public synchronized void add(PopupMenu popup) ★

The `add()` method introduced in Java 1.1 provides the ability to associate a `PopupMenu` with a `Component`. The pop-up menu can be used to provide context-sensitive menus for specific components. (On some platforms for some components, pop-up menus exist already and cannot be overridden.) Interaction with the menu is discussed in Chapter 10, *Would You Like to Choose from the Menu?*

Multiple pop-up menus can be associated with a component. To display the appropriate pop-up menu, call the pop-up menu's `show()` method.

public synchronized void remove(MenuComponent popup) ★

The `remove()` method is the `MenuContainer` interface method to disassociate the `popup` from the component. (`PopupMenu` is a subclass of `MenuComponent`.) If `popup` is not associated with the `Component`, nothing happens.

protected String paramString ()

The `paramString()` method is a protected method that helps build a `String` listing the different parameters of the `Component`. When the `toString()` method is called for a specific `Component`, `paramString()` is called for the lowest level and works its way up the inheritance hierarchy to build a complete parameter string to display. At the `Component` level, potentially seven (Java1.0) or eight (1.1) items are added. The first five items added are the component's name (if non-null and using Java 1.1), `x` and `y` coordinates (as returned by `getLocation()`), along with its width and height (as returned by `getSize()`). If the component is not valid, "invalid" is added next. If the component is not visible, "hidden" is added next. Finally, if the component is not enabled, "disabled" is added.

public String toString ()

The `toString()` method returns a `String` representation of the object's values. At the `Component` level, the class's name is placed before the results of `paramString()`. This method is called automatically by the system if you try to print an object using `System.out.println()`.

public void list ()

The `list()` method prints the contents of the `Component` (as returned by `toString()`) to `System.out`. If `c` is a type of `Component`, the two statements `System.out.println(c)` and `c.list()` are equivalent. This method is more useful at the `Container` level, because it prints all the components within the container.

public void list (PrintWriter out) ★

public void list (PrintStream out)

This version of `list()` prints the contents of the `Component` (as returned by `toString()`) to a different `PrintStream`, `out`.

public void list (PrintWriter out, int indentation) ★

public void list (PrintStream out, int indentation)

These versions of `list()` are called by the other two. They print the component's contents (as returned by `toString()`) with the given indentation. This allows you to prepare nicely formatted lists of a container's contents for debugging; you could use the indentation to reflect how deeply the component is nested within the container.

5.1.2 Component Events

Chapter 4, *Events* covers event handling in detail. This section summarizes what `Component` does for the different event-related methods.

With the Java 1.0 event model, many methods return `true` to indicate that the program has handled the event and `false` to indicate that the event was not handled (or only partially handled); when `false` is returned, the system passes the event up to the parent container. Thus, it is good form to return `true` only when you have fully handled the event, and no further processing is necessary.

With the Java 1.1 event model, you register a listener for a specific event type. When that type of event happens, the listener is notified. Unlike the 1.0 model, you do not need to override any methods of `Component` to handle the event.

Controllers

The Java 1.0 event model controllers are `deliverEvent()`, `postEvent()`, and `handleEvent()`. With 1.1, the controller is a method named `dispatchEvent()`.

public void deliverEvent (Event e) ☆

The `deliverEvent()` method delivers the 1.0 `Event e` to the `Component` in which an event occurred. Internally, this method calls `postEvent()`. The `deliverEvent()` method is an important enhancement to `postEvent()` for `Container` objects since they have to determine which component in the `Container` gets the event.

public boolean postEvent (Event e) ☆

The `postEvent()` method tells the `Component` to deal with 1.0 `Event e`. It calls `handleEvent()`, which returns `true` if some other object handled `e` and `false` if no one handles it. If `handleEvent()` returns `false`, `postEvent()` posts the `Event` to the component's parent. You can use `postEvent()` to hand any events you generate yourself to some other component for processing. (Creating your own events is a useful technique that few developers take advantage of.) You can also use `postEvent()` to reflect an event from one component into another.

public boolean handleEvent (Event e) ☆

The `handleEvent()` method determines the type of event `e` and passes it along to an appropriate method to deal with it. For example, when a mouse motion event is delivered to `postEvent()`, it is passed off to `handleEvent()`, which calls `mouseMove()`. As shown in the following listing, `handleEvent()` can be implemented as one big `switch` statement. Since not all event types have default event handlers, you may need to override this method. If you do, remember to

call the overridden method to ensure that the default behavior still takes place. To do so, call `super.handleEvent(event)` for any event your method does not deal with.

```
public boolean handleEvent(Event event) {
    switch (event.id) {
        case Event.MOUSE_ENTER:
            return mouseEnter (event, event.x, event.y);
        case Event.MOUSE_EXIT:
            return mouseExit (event, event.x, event.y);
        case Event.MOUSE_MOVE:
            return mouseMove (event, event.x, event.y);
        case Event.MOUSE_DOWN:
            return mouseDown (event, event.x, event.y);
        case Event.MOUSE_DRAG:
            return mouseDrag (event, event.x, event.y);
        case Event.MOUSE_UP:
            return mouseUp (event, event.x, event.y);
        case Event.KEY_PRESS:
        case Event.KEY_ACTION:
            return keyDown (event, event.key);
        case Event.KEY_RELEASE:
        case Event.KEY_ACTION_RELEASE:
            return keyUp (event, event.key);
        case Event.ACTION_EVENT:
            return action (event, event.arg);
        case Event.GOT_FOCUS:
            return gotFocus (event, event.arg);
        case Event.LOST_FOCUS:
            return lostFocus (event, event.arg);
    }
    return false;
}
```

public final void dispatchEvent(AWTEvent e) ★

The `dispatchEvent()` method allows you to post new AWT events to this component's listeners. `dispatchEvent()` tells the `Component` to deal with the `AWTEvent e` by calling its `processEvent()` method. This method is similar to Java 1.0's `postEvent()` method. Events delivered in this way bypass the system's event queue. It's not clear why you would want to bypass the event queue, except possibly to deliver some kind of high priority event.

Action

public boolean action (Event e, Object o) ☆

The `action()` method is called when the user performs some action in the `Component`. `e` is the 1.0 `Event` instance for the specific event, while the content of `o` varies depending upon the specific `Component`. The particular action that

triggers a call to `action()` depends on the `Component`. For example, with a `TextField`, `action()` is called when the user presses the carriage return. This method should not be called directly; to deliver any event you generate, call `postEvent()`, and let it decide how the event should propagate.

The default implementation of the `action()` method does nothing and returns `false`. When you override this method, return `true` only if you fully handle the event. Your method should always have a default case that returns `false` or calls `super.action(e, o)` to ensure that the event propagates to the component's container or component's superclass, respectively.

Keyboard

public boolean keyDown (Event e, int key) ★

The `keyDown()` method is called whenever the user presses a key. `e` is the `Event` instance for the specific event, while `key` is the integer representation of the character pressed. The identifier for the event (`e.id`) could be either `Event.KEY_PRESS` for a regular key or `Event.KEY_ACTION` for an action-oriented key (e.g., arrow or function key). The default `keyDown()` method does nothing and returns `false`. If you are doing input validation, return `true` if the character is invalid; this keeps the event from propagating to a higher component. If you wish to alter the input (i.e., convert to uppercase), return `false`, but change `e.key` to the new character.

public boolean keyUp (Event e, int key)

The `keyUp()` method is called whenever the user releases a key. `e` is the `Event` instance for the specific event, while `key` is the integer representation of the character pressed. The identifier for the event (`e.id`) could be either `Event.KEY_RELEASE` for a regular key or `Event.KEY_ACTION_RELEASE` for an action-oriented key (e.g., arrow or function key). `keyUp()` may be used to determine how long `key` has been pressed. The default `keyUp()` method does nothing and returns `false`.

Mouse

NOTE Early releases of Java (1.0.2 and earlier) propagated only mouse events from `Canvas` and `Container` objects. However, Netscape Navigator seems to have jumped the gun and corrected the situation with their 3.0 release, which is based on Java release 1.0.2.1. Until other Java releases catch up, use these events with care. For more information on platform dependencies, see Appendix C, *Platform-Specific Event Handling*.

public boolean mouseDown (Event e, int x, int y) ☆

The `mouseDown()` method is called when the user presses a mouse button over the `Component`. `e` is the `Event` instance for the specific event, while `x` and `y` are the coordinates where the cursor was located when the event was initiated. It is necessary to examine the `modifiers` field of `e` to determine which mouse button the user pressed. The default `mouseDown()` method does nothing and returns `false`. When you override this method, return `true` only if you fully handle the event. Your method should always have a default case that returns `false` or calls `super.mouseDown(e, x, y)` to ensure that the event propagates to the component's container or component's superclass, respectively.

public boolean mouseDrag (Event e, int x, int y) ☆

The `mouseDrag()` method is called when the user is pressing a mouse button and moves the mouse. `e` is the `Event` instance for the specific event, while `x` and `y` are the coordinates where the cursor was located when the event was initiated. `mouseDrag()` could be called multiple times as the mouse is moved. The default `mouseDrag()` method does nothing and returns `false`. When you override this method, return `true` only if you fully handle the event. Your method should always have a default case that returns `false` or calls `super.mouseDrag(e, x, y)` to ensure that the event propagates to the component's container or component's superclass, respectively.

public boolean mouseEnter (Event e, int x, int y) ☆

The `mouseEnter()` method is called when the mouse enters the `Component`. `e` is the `Event` instance for the specific event, while `x` and `y` are the coordinates where the cursor was located when the event was initiated. The default `mouseEnter()` method does nothing and returns `false`. `mouseEnter()` can be used for implementing balloon help. When you override this method, return `true` only if you fully handle the event. Your method should always have a default case that returns `false` or calls `super.mouseEnter(e, x, y)` to ensure that the event propagates to the component's container or component's superclass, respectively.

public boolean mouseExit (Event e, int x, int y) ☆

The `mouseExit()` method is called when the mouse exits the `Component`. `e` is the `Event` instance for the specific event, while `x` and `y` are the coordinates where the cursor was located when the event was initiated. The default method `mouseExit()` does nothing and returns `false`. When you override this method, return `true` only if you fully handle the event. Your method should always have a default case that returns `false` or calls `super.mouseExit(e, x, y)` to ensure that the event propagates to the component's container or component's superclass, respectively.

public boolean mouseMove (Event e, int x, int y) ☆

The `mouseMove()` method is called when the user moves the mouse without pressing a mouse button. `e` is the `Event` instance for the specific event, while `x` and `y` are the coordinates where the cursor was located when the event was initiated. `mouseMove()` will be called numerous times as the mouse is moved. The default `mouseMove()` method does nothing and returns `false`. When you override this method, return `true` only if you fully handle the event. Your method should always have a default case that returns `false` or calls `super.mouseMove(e, x, y)` to ensure that the event propagates to the component's container or component's superclass, respectively.

public boolean mouseUp (Event e, int x, int y) ☆

The `mouseUp()` method is called when the user releases a mouse button over the `Component`. `e` is the `Event` instance for the specific event, while `x` and `y` are the coordinates where the cursor was located when the event was initiated. The default `mouseUp()` method does nothing and returns `false`. When you override this method, return `true` only if you fully handle the event. Your method should always have a default case that returns `false` or calls `super.mouseUp(e, x, y)` to ensure that the event propagates to the component's container or component's superclass, respectively.

Focus

Focus events indicate whether a component can get keyboard input. Not all components can get focus (e.g., `Label` cannot). Precisely which components can get the focus is platform specific.

Ordinarily, the item with the focus has a light gray rectangle around it, though the actual display depends on the platform and the component. Figure 5-1 displays the effect of focus for buttons in Windows 95.

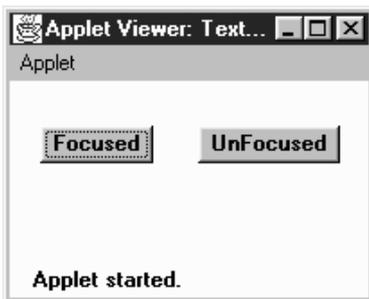


Figure 5-1: Focused and UnFocused buttons

NOTE Early releases of Java (1.0.2 and earlier) do not propagate all focus events on all platforms. Java 1.1 seems to propagate them properly. For more information on platform dependencies, see Appendix C.

public boolean gotFocus (Event e, Object o) ☆

The `gotFocus()` method is triggered when the `Component` gets the input focus. `e` is the `Event` instance for the specific event, while the content of `o` varies depending upon the specific `Component`. The default `gotFocus()` method does nothing and returns `false`. For a `TextField`, when the cursor becomes active, it has the focus. When you override this method, return `true` to indicate that you have handled the event completely or `false` if you want the event to propagate to the component's container.

public boolean lostFocus (Event e, Object o) ☆

The `lostFocus()` method is triggered when the input focus leaves the `Component`. `e` is the `Event` instance for the specific event, while the content of `o` varies depending upon the specific `Component`. The default `lostFocus()` method does nothing and returns `false`. When you override this method, return `true` to indicate that you have handled the event completely or `false` if you want the event to propagate to the component's container.

Listeners and 1.1 Event Handling

With the 1.1 event model, you receive events by registering event listeners, which are told when the event happens. Components don't have to receive and handle their own events; you can cleanly separate the event-handling code from the user interface itself. This section covers the methods used to add and remove event listeners, which are part of the `Component` class. There is a pair of methods to add and remove listeners for each event type that is appropriate for a `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, and `MouseEvent`. Subclasses of `Component` may have additional event types and therefore will have additional methods for adding and removing listeners. For example, `Button`, `List`, `MenuItem`, and `TextField` each generate action events and therefore have methods to add and remove action listeners. These additional listeners are covered with their respective components.

public void addComponentListener(ComponentListener listener) ★

The `addComponentListener()` method registers `listener` as an object interested in being notified when a `ComponentEvent` passes through the `EventQueue` with this `Component` as its target. When such an event occurs, a method in the `ComponentListener` interface is called. Multiple listeners can be registered.

public void removeComponentListener(ComponentListener listener) ★

The `removeComponentListener()` method removes `listener` as a interested listener. If `listener` is not registered, nothing happens.

public void addFocusListener(FocusListener listener) ★

The `addFocusListener()` method registers `listener` as an object interested in being notified when a `FocusEvent` passes through the `EventQueue` with this `Component` as its target. When such an event occurs, a method in the `FocusListener` interface is called. Multiple listeners can be registered.

public void removeFocusListener(FocusListener listener) ★

The `removeFocusListener()` method removes `listener` as a interested listener. If `listener` is not registered, nothing happens.

public void addKeyListener(KeyListener listener) ★

The `addKeyListener()` method registers `listener` as an object interested in being notified when a `KeyEvent` passes through the `EventQueue` with this `Component` as its target. When such an event occurs, a method in the `KeyListener` interface is called. Multiple listeners can be registered.

public void removeKeyListener(KeyListener listener) ★

The `removeKeyListener()` method removes `listener` as a interested listener. If `listener` is not registered, nothing happens.

public void addMouseListener(MouseListener listener) ★

The `addMouseListener()` method registers `listener` as an object interested in being notified when a nonmotion-oriented `MouseEvent` passes through the `EventQueue` with this `Component` as its target. When such an event occurs, a method in the `MouseListener` interface is called. Multiple listeners can be registered.

public void removeMouseListener(MouseListener listener) ★

The `removeMouseListener()` method removes `listener` as a interested listener. If `listener` is not registered, nothing happens.

public void addMouseMotionListener(MouseMotionListener listener) ★

The `addMouseMotionListener()` method registers `listener` as an object interested in being notified when a motion-oriented `MouseEvent` passes through the `EventQueue` with this `Component` as its target. When such an event occurs, a method in the `MouseMotionListener` interface is called. Multiple listeners can be registered.

The mouse motion-oriented events are separate from the other mouse events because of their frequency of generation. If they do not have to propagate around, resources can be saved.

public void removeMouseListener(MouseMotionListener listener) ★

The `removeMouseListener()` method removes `listener` as a interested listener. If `listener` is not registered, nothing happens.

Handling your own events

Under the 1.1 event model, it is still possible for components to receive their own events, simulating the old event mechanism. If you want to write components that process their own events but are also compatible with the new model, you can override `processEvent()` or one of its related methods. `processEvent()` is logically similar to `handleEvent()` in the old model; it receives all the component's events and sees that they are forwarded to the appropriate listeners. Therefore, by overriding `processEvent()`, you get access to every event the component generates. If you want only a specific type of event, you can override `processComponentEvent()`, `processKeyEvent()`, or one of the other event-specific methods.

However, there is one problem. In Java 1.1, events aren't normally generated if there are no listeners. Therefore, if you want to receive your own events without registering a listener, you should first enable event processing (by a call to `enableEvent()`) to make sure that the events you are interested in are generated.

protected final void enableEvents(long eventsToEnable) ★

The `enableEvents()` method allows you to configure a component to listen for events without having any active listeners. Under normal circumstances (i.e., if you are not subclassing a component), it is not necessary to call this method.

The `eventsToEnable` parameter contains a mask specifying which event types you want to enable. The `AWTEvent` class (covered in Chapter 4) contains constants for the following types of events:

COMPONENT_EVENT_MASK
CONTAINER_EVENT_MASK
FOCUS_EVENT_MASK
KEY_EVENT_MASK
MOUSE_EVENT_MASK
MOUSE_MOTION_EVENT_MASK
WINDOW_EVENT_MASK
ACTION_EVENT_MASK
ADJUSTMENT_EVENT_MASK
ITEM_EVENT_MASK
TEXT_EVENT_MASK

OR the masks for the events you want; for example, call `enableEvents(MOUSE_EVENT_MASK | MOUSE_MOTION_EVENT_MASK)` to enable all mouse events. Any previous event mask settings are retained.

protected final void disableEvents(long eventsToDisable) ★

The `disableEvents()` method allows you to stop the delivery of events when they are no longer needed. `eventsToDisable` is similar to the `eventsToEnable` parameter but instead contains a mask specifying which event types to stop. A disabled event would still be delivered if someone were listening.

protected void processEvent(AWTEvent e) ★

The `processEvent()` method receives all `AWTEvent` with this `Component` as its target. `processEvent()` then passes them along to one of the event-specific processing methods (e.g., `processKeyEvent()`). When you subclass `Component`, overriding `processEvent()` allows you to process all events without providing listeners. Remember to call `super.processEvent(e)` last to ensure that normal event processing still occurs; if you don't, events won't get distributed to any registered listeners. Overriding `processEvent()` is like overriding the `handleEvent()` method using the 1.0 event model.

protected void processComponentEvent(ComponentEvent e) ★

The `processComponentEvent()` method receives `ComponentEvent` with this `Component` as its target. If any listeners are registered, they are then notified. When you subclass `Component`, overriding `processComponentEvent()` allows you to process component events without providing listeners. Remember to call `super.processComponentEvent(e)` last to ensure that normal event processing still occurs; if you don't, events won't get distributed to any registered listeners. Overriding `processComponentEvent()` is roughly similar to overriding `resize()`, `move()`, `show()`, and `hide()` to add additional functionality when those methods are called.

protected void processFocusEvent(FocusEvent e) ★

The `processFocusEvent()` method receives `FocusEvent` with this `Component` as its target. If any listeners are registered, they are then notified. When you subclass `Component`, overriding `processFocusEvent()` allows you to process the focus event without providing listeners. Remember to call `super.processFocusEvent(e)` last to ensure that normal event processing still occurs; if you don't, events won't get distributed to any registered listeners. Overriding `processFocusEvent()` is like overriding the methods `gotFocus()` and `lostFocus()` using the 1.0 event model.

protected void processKeyEvent(KeyEvent e) ★

The `processKeyEvent()` method receives `KeyEvent` with this `Component` as its target. If any listeners are registered, they are then notified. When you subclass `Component`, overriding `processKeyEvent()` allows you to process key events without providing listeners. Be sure to remember to call `super.processKeyEvent(e)` last to ensure that normal event processing still occurs; if you don't, events won't get distributed to any registered listeners. Overriding `processKeyEvent()` is roughly similar to overriding `keyDown()` and `keyUp()` with one method using the 1.0 event model.

protected void processMouseEvent(MouseEvent e) ★

This `processMouseEvent()` method receives all nonmotion-oriented `MouseEvents` with this `Component` as its target. If any listeners are registered, they are then notified. When you subclass `Component`, overriding the method `processMouseEvent()` allows you to process mouse events without providing listeners. Remember to call `super.processMouseEvent(e)` last to ensure that normal event processing still occurs; if you don't, events won't get distributed to any registered listeners. Overriding the method `processMouseEvent()` is roughly similar to overriding `mouseDown()`, `mouseUp()`, `mouseEnter()`, and `mouseExit()` with one method using the 1.0 event model.

protected void processMouseMotionEvent(MouseEvent e) ★

The `processMouseMotionEvent()` method receives all motion-oriented `MouseEvents` with this `Component` as its target. If there are any listeners registered, they are then notified. When you subclass `Component`, overriding `processMouseMotionEvent()` allows you to process mouse motion events without providing listeners. Remember to call `super.processMouseMotionEvent(e)` last to ensure that normal event processing still occurs; if you don't, events won't get distributed to any registered listeners. Overriding the method `processMouseMotionEvent()` is roughly similar to overriding `mouseMove()` and `mouseDrag()` with one method using the 1.0 event model.

5.2 Labels

Having covered the features of the `Component` class, we can now look at some of the simplest components. The first component introduced here is a `Label`. A label is a `Component` that displays a single line of static text.* It is useful for putting a title or message next to another component. The text can be centered or justified to the left or right. Labels react to all events they receive. However, they do not get any events from their peers.

* *Java in A Nutshell* (from O'Reilly & Associates) includes a multiline `Label` component.

5.2.1 Label Methods

Constants

There are three alignment specifiers for labels. The alignment tells the `Label` where to position its text within the space allotted. Setting an alignment for a `Label` might not do anything noticeable if the `LayoutManager` being used does not resize the `Label` to give it more space. With `FlowLayout`, the alignment is barely noticeable. See Chapter 7, *Layouts*, for more information.

public final static int LEFT

`LEFT` is the constant for left alignment. If no alignment is specified in the constructor, left alignment is the default.

public final static int CENTER

`CENTER` is the constant for center alignment.

public final static int RIGHT

`RIGHT` is the constant for right alignment.

Constructors

public Label ()

This constructor creates an empty `Label`. By default, the label's text is left justified.

public Label (String label)

This constructor creates a `Label` whose initial text is `label`. By default, the label's text is left justified.

public Label (String label, int alignment)

This constructor creates a `Label` whose initial text is `label`. The alignment of the label is `alignment`. If `alignment` is invalid (not `LEFT`, `RIGHT`, or `CENTER`), the constructor throws the run-time exception `IllegalArgumentException`.

Text

public String getText ()

The `getText ()` method returns the current value of `Label`.

public void setText (String label)

The `setText ()` method changes the text of the `Label` to `label`. If the new label is a different size from the old one, you should revalidate the display to ensure the label's entire contents will be seen.

Alignment

public int getAlignment ()

The `getAlignment ()` method returns the current alignment of the `Label`.

public void setAlignment (int alignment)

The `setAlignment ()` method changes the alignment of the `Label` to `alignment`. If `alignment` is invalid (not `LEFT`, `RIGHT`, or `CENTER`), `setAlignment ()` throws the run-time exception `IllegalArgumentException`. Figure 5-2 shows all three alignments.

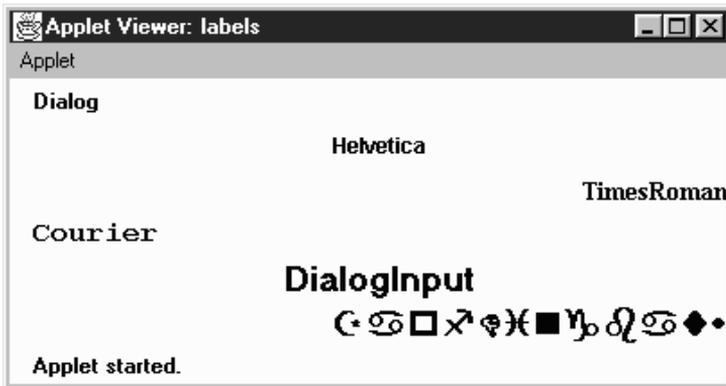


Figure 5-2: Labels with different alignments

Miscellaneous methods

public synchronized void addNotify ()

The `addNotify ()` method creates the `Label` peer. If you override this method, first call `super.addNotify ()`, then put in your customizations. Then you will be able to do everything you need with the information about the newly created peer.

protected String paramString ()

The `paramString ()` method overrides `Component`'s `paramString ()` method. It is a protected method that calls the overridden `paramString ()` to build a `String` from the different parameters of the `Component`. When the method `paramString ()` is called for a `Label`, the alignment and label's text are added. Thus, for the `Label` created by the constructor `new Label ("ZapfDingbats", Label.RIGHT)`, the results displayed from a call to `toString ()` would be:

```
java.awt.Label[0,0,0x0,invalid,align=right,label=ZapfDingbats]
```

5.2.2 Label Events

The `Label` component can react to any event it receives, though the `Label` peer normally does not send any. However, there is nothing to stop you from posting an event yourself.

5.3 Buttons

The `Button` component provides one of the most frequently used objects in graphical applications. When the user selects a button, it signals the program that something needs to be done by sending an action event. The program responds in its `handleEvent()` method (for Java 1.0) or its `actionPerformed()` method (defined by Java 1.1's `ActionListener` interface). Next to `Label`, which does nothing, `Button` is the simplest component to understand. Because it is so simple, we will use a lot of buttons in our examples for the next few chapters.

5.3.1 Button Methods

Constructors

public Button ()

This constructor creates an empty `Button`. You can set the label later with `setLabel()`.

public Button (String label)

This constructor creates a `Button` whose initial text is `label`.

Button Labels

public String getLabel ()

The `getLabel()` method retrieves the current text of the label on the `Button` and returns it as a `String`.

public synchronized void setLabel (String label)

The `setLabel()` method changes the text of the label on the `Button` to `label`. If the new text is a different size from the old, it is necessary to revalidate the screen to ensure that the button size is correct.

Action Commands

With Java 1.1, every button can have two names. One is what the user sees (the button's label); the other is what the programmer sees and is called the button's *action command*. Distinguishing between the label and the action command is a major help to internationalization. The label can be localized for the user's environment.

However, this means that labels can vary at run-time and are therefore useless for comparisons within the program. For example, you can't test whether the user pushed the Yes button if that button might read Oui or Ja, depending on some run-time environment setting. To give the programmer something reliable for comparisons, Java 1.1 introduces the action command. The action command for our button might be Yes, regardless of the button's actual label.

By default, the action command is equivalent to the button's label. Java 1.0 code, which only relies on the label, will continue to work. Furthermore, you can continue to write in the Java 1.0 style as long as you're sure that your program will never have to account for other languages. These days, that's a bad bet. Even if you aren't implementing multiple locales now, get in the habit of testing a button's action command rather than its label; you will have less work to do when internationalization does become an issue.

public String getActionCommand () ★

The `getActionCommand()` method returns the button's current action command. If no action command was explicitly set, this method returns the label.

public void setActionCommand (String command) ★

The `setActionCommand()` method changes the button's action command to `command`.

Miscellaneous methods

public synchronized void addNotify ()

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

protected String paramString ()

The `paramString()` method overrides the component's `paramString()` method. It is a protected method that calls the overridden `paramString()` to build a `String` from the different parameters of the `Component`. When the method `paramString()` is called for a `Button`, the button's label is added. Thus, for the `Button` created by the constructor `new Button ("ZapfDingbats")`, the results displayed from a call to `toString()` could be:

```
java.awt.Button[77,5,91x21,label=ZapfDingbats]
```

5.3.2 Button Events

With the 1.0 event model, Button components generate an ACTION_EVENT when the user selects the button.

With the version 1.1 event model, you register an ActionListener with the method addActionListener(). When the user selects the Button, the method ActionListener.actionPerformed() is called through the protected Button.processActionEvent() method. Key, mouse, and focus listeners are registered through the Component methods of addKeyListener(), addMouseListener(), or addMouseMotionListener(), and addFocusListener(), respectively.

Action

public boolean action (Event e, Object o)

The action() method for a Button is called when the user presses and releases the button. e is the Event instance for the specific event, while o is the button's label. The default implementation of action() does nothing and returns false, passing the event to the button's container for processing. For a button to do something useful, you should override either this method or the container's action() method. Example 5-1 is a simple applet called ButtonTest that demonstrates the first approach; it creates a Button subclass called TheButton, which overrides action(). This simple subclass doesn't do much; it just labels the button and prints a message when the button is pressed. Figure 5-3 shows what ButtonTest looks like.

Example 5-1: Button Event Handling

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

class TheButton extends Button {
    TheButton (String s) {
        super (s);
    }
    public boolean action (Event e, Object o) {
        if ("One".equals(o)) {
            System.out.println ("Do something for One");
        } else if ("Two".equals(o)) {
            System.out.println ("Ignore Two");
        } else if ("Three".equals(o)) {
            System.out.println ("Reverse Three");
        } else if ("Four".equals(o)) {
            System.out.println ("Four is the one");
        } else {
            return false;
        }
        return true;
    }
}
```

Example 5-1: Button Event Handling (continued)

```

}
public class ButtonTest extends Applet {
    public void init () {
        add (new TheButton ("One"));
        add (new TheButton ("Two"));
        add (new TheButton ("Three"));
        add (new TheButton ("Four"));
    }
}

```

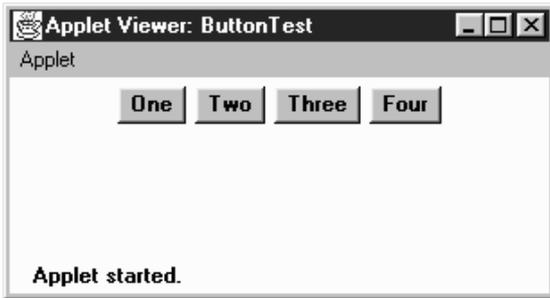


Figure 5-3: The ButtonTest applet

Keyboard

Buttons are able to capture keyboard-related events once the button has the input focus. In order to give a `Button` the input focus without triggering the action event, call `requestFocus()`. The button also gets the focus if the user selects it and drags the mouse off of it without releasing the mouse.

public boolean keyDown (Event e, int key) ☆

The `keyDown()` method is called whenever the user presses a key while the `Button` has the input focus. `e` is the `Event` instance for the specific event, while `key` is the integer representation of the character pressed. The identifier for the event (`e.id`) could be either `Event.KEY_PRESS` for a regular key or `Event.KEY_ACTION` for an action-oriented key (i.e., an arrow or a function key). There is no visible indication that the user has pressed a key over the button.

public boolean keyUp (Event e, int key) ☆

The `keyUp()` method is called whenever the user releases a key while the `Button` has the input focus. `e` is the `Event` instance for the specific event, while `key` is the integer representation of the character pressed. The identifier for the event (`e.id`) could be either `Event.KEY_RELEASE` for a regular key or `Event.KEY_ACTION_RELEASE` for an action-oriented key (i.e., an arrow or a function key). `keyUp()` may be used to determine how long key has been pressed.

Listeners and 1.1 event handling

With the 1.1 event model, you register listeners, which are told when the event happens.

public void addActionListener(ActionListener listener) ★

The `addActionListener()` method registers `listener` as an object interested in receiving notifications when an `ActionEvent` passes through the `EventQueue` with this `Button` as its target. The `listener.actionPerformed()` method is called when these events occur. Multiple listeners can be registered. The following code demonstrates how to use an `ActionListener` to handle the events that occur when the user selects a button. This applet has the same display as the previous one, shown in Figure 5-3.

```
// Java 1.1 only
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class ButtonTest11 extends Applet implements ActionListener {
    Button b;
    public void init () {
        add (b = new Button ("One"));
        b.addActionListener (this);
        add (b = new Button ("Two"));
        b.addActionListener (this);
        add (b = new Button ("Three"));
        b.addActionListener (this);
        add (b = new Button ("Four"));
        b.addActionListener (this);
    }
    public void actionPerformed (ActionEvent e) {
        String s = e.getActionCommand();
        if ("One".equals(s)) {
            System.out.println ("Do something for One");
        } else if ("Two".equals(s)) {
            System.out.println ("Ignore Two");
        } else if ("Three".equals(s)) {
            System.out.println ("Reverse Three");
        } else if ("Four".equals(s)) {
            System.out.println ("Four is the one");
        }
    }
}
```

public void removeActionListener(ActionListener listener) ★

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

protected void processEvent(AWTEvent e) ★

The `processEvent()` method receives `AWTEvent` with this `Button` as its target. `processEvent()` then passes them along to any listeners for processing. When you subclass `Button`, 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 processActionEvent(ActionEvent e) ★

The `processActionEvent()` method receives `ActionEvent` with this `Button` as its target. `processActionEvent()` then passes them along to any listeners for processing. When you subclass `Button`, overriding `processActionEvent()` allows you to process all action events yourself, before sending them to any listeners. In a way, overriding `processActionEvent()` is like overriding `action()` using the 1.0 event model.

If you override the `processActionEvent()` method, you must remember to call `super.processActionEvent(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.

5.4 A Simple Calculator

It is always helpful to see complete and somewhat useful examples after learning something new. Example 5-2 shows a working calculator that performs floating point addition, subtraction, multiplication, and division. Figure 5-4 shows the calculator in operation. The button in the lower left corner is a decimal point. This applet uses a number of classes that will be discussed later in the book (most notably, some layout managers and a `Panel`); try to ignore them for now. Focus on the `action()` and `compute()` methods; `action()` figures out which button was pressed, converting it to a digit (0–9 plus the decimal point) or an operator (=, +, −, *, /). As you build a number, it is displayed in the label `lab`, which conveniently serves to store the number in string form. The `compute()` method reads the label's text, converts it to a floating point number, does the computation, and displays the result in the label. The `addButtons()` method is a helper method to create a group of `Button` objects at one time.

Example 5-2: Calculator Source Code

```

import java.awt.*;
import java.applet.*;

public class JavaCalc extends Applet {
    Label lab;
    boolean firstDigit = true;
    float savedValue = 0.0f;    // Initial value
    String operator = "=";    // Initial operator
    public void addButtons (Panel p, String labels) {
        int count = labels.length();
        for (int i=0;i<count;i++)
            p.add (new Button (labels.substring(i,i+1)));
    }
    public void init () {
        setLayout (new BorderLayout());
        add ("North", lab = new Label ("0", Label.RIGHT));
        Panel p = new Panel();
        p.setLayout (new GridLayout (4, 4));
        addButtons (p, "789/");
        addButtons (p, "456*");
        addButtons (p, "123-");
        addButtons (p, ".0+=");
        add ("Center", p);
    }
    public boolean action (Event e, Object o) {
        if (e.target instanceof Button) {
            String s = (String)o;
            if ("0123456789.".indexOf (s) != -1) { // isDigit
                if (firstDigit) {
                    firstDigit = false;
                    lab.setText (s);
                } else {
                    lab.setText (lab.getText() + s);
                }
            } else { // isOperator
                if (!firstDigit) {
                    compute (lab.getText());
                    firstDigit = true;
                }
                operator = s;
            }
            return true;
        }
        return false;
    }
    public void compute (String s) {
        float sValue = new Float (s).floatValue();
        char c = operator.charAt (0);
        switch (c) {
            case '=':    savedValue = sValue;
                        break;
            case '+':    savedValue += sValue;
        }
    }
}

```

Example 5–2: Calculator Source Code (continued)

```

        break;
    case '-':  savedValue -= sValue;
              break;
    case '*':  savedValue *= sValue;
              break;
    case '/':  savedValue /= sValue;
              break;
    }
    lab.setText (String.valueOf(savedValue));
}
}

```

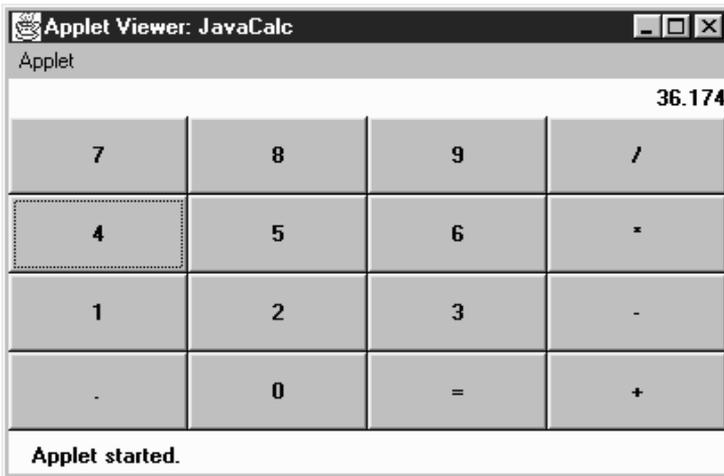


Figure 5–4: Calculator applet

5.5 Canvas

A Canvas is a class just waiting to be subclassed. Through Canvas, you can create additional AWT objects that are not provided by the base classes. Canvas is also useful as a drawing area, particularly when additional components are on the screen. It is tempting to draw directly onto a Container, but this often isn't a good idea. Anything you draw might disappear underneath the components you add to the container. When you are drawing on a container, you are essentially drawing on the background. The container's layout manager doesn't know anything about what you have drawn and won't arrange components with your artwork in mind. To be safe, do your drawing onto a Canvas and place that Canvas in a Container.

5.5.1 Canvas Methods

Constructors

public Canvas () ★

The constructor creates a new `Canvas` with no default size. If you place the canvas in a container, the container's layout manager sizes the canvas for you. If you aren't placing the canvas in a container, call `setBounds()` to specify the canvas's size.

Java 1.0 used the default constructor for `Canvas`; there was no explicit constructor.

Miscellaneous methods

public void paint (Graphics g) ★

The default implementation of the `paint()` method colors the entire `Canvas` with the current background color. When you subclass this method, your `paint()` method needs to draw whatever should be shown on the canvas.

public synchronized void addNotify ()

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

5.5.2 Canvas Events

The `Canvas` peer passes all events to you, which is why it's well suited to creating your own components.

5.6 Creating Your Own Component

If you find that no AWT component satisfies your needs, you can create your own. This is usually done either by extending an existing component or by starting from scratch. When extending an existing component, you start with the base functionality of an existing object and add to it. The users will not see anything new or different about the object until they start to interact with it, since it is not a new component. For example, a `TextField` could be subclassed to convert all letters input to uppercase. On the other hand, if you create a new component from scratch, it will appear the same on all platforms (regardless of what the platform's native components look like), and you have to make sure the user can fairly easily figure out how to work with it. Example 5-3 shows how to create your own `Component` by creating a `Label` that displays vertically, as opposed to the standard `Label` `Component` that displays horizontally. The whole process is fairly easy.

The third possibility for creating your own components involves adding functionality to containers. This is fairly easy to do and can be useful if you are constantly grouping components together. For example, if you are always adding a `TextField` or `Label` to go with a `Scrollbar` to display the value, do it once, and call it something meaningful like `LabeledScrollbarPanel`. Then whenever you need it again, reuse your `LabeledScrollbarPanel`. Think about reusability whenever you can.

With Java 1.1, the colors for these new components should be set to color values consistent to the user's platform. This is done through color constants provided in the `SystemColor` class introduced in Chapter 2.

5.6.1 *VerticalLabel*

When you create new components, they must meet three requirements:

- In Java 1.0, you must extend a subclass of `Component`, usually `Canvas`. In Java 1.1, you can extend `Component` itself, creating a lightweight component. In many cases, this alternative is more efficient.
- You must provide a constructor for the new component so that you can create new instances of it; if you really don't need a constructor, you can use the default constructor that you inherit from `Canvas` or `Component`.
- You must provide a way to draw the object on the screen by overriding the `paint()` method.

If initializing the component requires information about display characteristics (for example, you need to know the default `Font`), you must wait until the object is displayed on the screen before you initialize it. This is done by overriding the `addNotify()` method. First, call `super.addNotify()` to create the peer; you can now ask for platform-dependent information and initialize your component accordingly. Remember to override `getPreferredSize()` and `getMinimumSize()` (the Java 1.0 names are `preferredSize()` and `minimumSize()`) to return the proper dimensions for the new component, so that layout management works properly. There can be other support methods, depending upon the requirements of the object. For example, it is helpful, but not required, to provide a `toString()` or `paramString()` method.

Creating a new component sounds a lot harder than it is. Example 5-3 contains the source for a new component called `VerticalLabel`. It displays a label that reads from top to bottom, instead of from left to right, and can be configured to display its text right or left justified or centered. Figure 5-5 displays the new component `VerticalLabel` in action.

Example 5-3: Source for VerticalLabel Component

```

import java.awt.*;

public class VerticalLabel extends Canvas {
    public static final int LEFT = 0;
    public static final int CENTER = 1;
    public static final int RIGHT = 2;
    private String text;
    private int    vgap;
    private int    alignment;
    Dimension     mySize;
    int           textLength;
    char          chars[];
    // constructors
    public VerticalLabel () {
        this (null, 0, CENTER);
    }
    public VerticalLabel (String text) {
        this (text, 0, CENTER);
    }
    public VerticalLabel (String text, int vgap, int alignment) {
        this.text = text;
        this.vgap = vgap;
        this.alignment = alignment;
    }
    void init () {
        textLength = text.length();
        chars = new char[textLength];
        text.getChars (0, textLength, chars, 0);
        Font f = getFont();
        FontMetrics fm = getFontMetrics (f);
        mySize = new Dimension(0,0);
        mySize.height = (fm.getHeight() * textLength) + (vgap * 2);
        for (int i=0; i < textLength; i++) {
            mySize.width = Math.max (mySize.width, fm.charsWidth(chars, i, 1));
        }
    }
    public int getAlignment () {
        return alignment;
    }
    public void addNotify () {
        super.addNotify();
        init(); // Component must be visible for init to work
    }
    public void setText (String text)    {this.text = text; init();}
    public String getText ()             {return text; }
    public void setVgap (int vgap)       {this.vgap = vgap; init();}
    public int getVgap ()                 {return vgap; }
    public Dimension preferredSize ()    {return mySize; }
    public Dimension minimumSize ()      {return mySize; }
    public void paint (Graphics g) {
        int x,y;
        int xPositions[];

```

Example 5-3: Source for VerticalLabel Component (continued)

```

        int yPositions[];
// Must redo this each time since font/screen area might change
// Use actual width for alignment
        Font f = getFont();
        FontMetrics fm = getFontMetrics (f);
        xPositions = new int[textLength];
        for (int i=0; i < textLength; i++) {
            if (alignment == RIGHT) {
                xPositions[i] = size().width - fm.charWidth (chars[i]);
            } else if (alignment == LEFT) {
                xPositions[i] = 0;
            } else { // CENTER
                xPositions[i] = (size().width - fm.charWidth (chars[i])) / 2;
            }
        }
        yPositions = new int[textLength];
        for (int i=0; i < textLength; i++) {
            yPositions[i] = (fm.getHeight() * (i+1)) + vgap;
        }
        for (int i = 0; i < textLength; i++) {
            x = xPositions[i];
            y = yPositions[i];
            g.drawChars (chars, i, 1, x, y);
        }
    }
    protected String paramString () {
        String str=",align=";
        switch (alignment) {
            case LEFT:    str += "left"; break;
            case CENTER:  str += "center"; break;
            case RIGHT:   str += "right"; break;
        }
        if (vgap!=0) str+= ",vgap=" + vgap;
        return super.paramString() + str + ",label=" + text;
    }
}

```

The following code is a simple applet using the `VerticalLabel`. It creates five instances of `VerticalLabel` within a `BorderLayout` panel, with gaps (see Chapter 7 for more on `BorderLayout`). The top and bottom labels are justified to the left and right, respectively, to demonstrate justification.

```

import java.awt.*;
import java.applet.*;
public class vlabels extends Applet {
    public void init () {
        setLayout (new BorderLayout (10, 10));
        setFont (new Font ("TimesRoman", Font.BOLD, 12));
        add ("North", new VerticalLabel ("One", 10, VerticalLabel.LEFT));
        add ("South", new VerticalLabel ("Two", 10, VerticalLabel.RIGHT));
        add ("West", new VerticalLabel ("Three"));
        add ("East", new VerticalLabel ("Four"));
    }
}

```



Figure 5-5: Using `VerticalLabel`

```

        add ("Center", new VerticalLabel ("Five"));
        resize (preferredSize());
    }
}

```

5.6.2 *Lightweight VerticalLabel*

The `VerticalLabel` in Example 5-3 works in both Java 1.0 and 1.1 but is relatively inefficient. When you create one, the system must create a `Canvas` and the peer of the `Canvas`. This work doesn't gain you anything; since this is a new component, it doesn't have to match the native appearance of any other component.

In Java 1.1, there's a way to avoid the overhead if you are creating a component that doesn't have to match a native object. This is called a *lightweight component*. To create one, you just subclass `Component` itself. To make a lightweight version of our `VerticalLabel`, we have to change only one line of code.

```

// Java 1.1 only
public class VerticalLabel extends Component

```

Everything else remains unchanged.

5.7 Cursor

Introduced in Java 1.1, the `Cursor` class provides the different cursors that can be associated with a `Component`. Previously, cursors could only be associated with a whole `Frame`. Now any component can use fancy cursors when the user is interacting with the system.

To change the cursor, a component calls its `setCursor()` method; its argument is a `Cursor` object, which is defined by this class.

NOTE There is still no way to assign a user-defined cursor to a `Component`. You are restricted to the 14 predefined cursors.

5.7.1 Cursor Constants

The following is a list of `Cursor` constants. The cursors corresponding to the constants are shown in Figure 5-6.

```
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 HAND_CURSOR  
public final static int MOVE_CURSOR  
public final static int N_RESIZE_CURSOR  
public final static int S_RESIZE_CURSOR  
public final static int E_RESIZE_CURSOR  
public final static int W_RESIZE_CURSOR  
public final static int NE_RESIZE_CURSOR  
public final static int NW_RESIZE_CURSOR  
public final static int SE_RESIZE_CURSOR  
public final static int SW_RESIZE_CURSOR
```

5.7.2 Cursor Methods

```
public Cursor (int type) ★
```

The sole constructor creates a `Cursor` of the specified `type`. `type` must be one of the `Cursor` class constants. If `type` is not one of the class constants, the constructor throws the run-time exception `IllegalArgumentException`.

This constructor exists primarily to support object serialization; you don't need to call it in your code. It is more efficient to call `getPredefinedCursor()`, discussed later in this section.

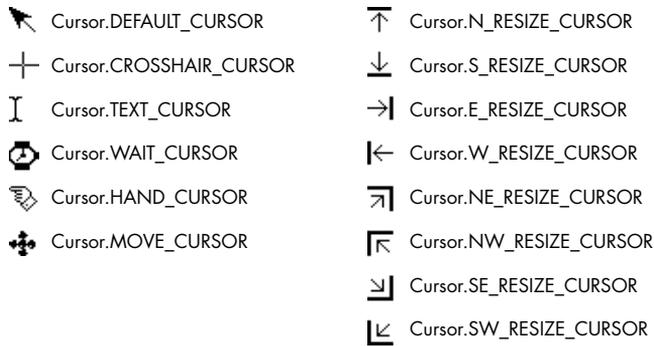


Figure 5–6: Standard Java cursors

Miscellaneous methods

public int getType() ★

The `getType()` method returns the cursor type. The value returned is one of the class constants.

static public Cursor getPredefinedCursor(int type) ★

The `getPredefinedCursor()` method returns the predefined `Cursor` of the given type. If `type` is not one of the class constants, this method throws the run-time exception `IllegalArgumentException`. This method checks what `Cursor` objects already exist and gives you a reference to a preexisting `Cursor` if it can find one with the appropriate type. Otherwise, it creates a new `Cursor` for you. This is more efficient than calling the `Cursor` constructor whenever you need one.

static public Cursor getDefaultCursor() ★

The `getDefaultCursor()` method returns the predefined `Cursor` for the `DEFAULT_CURSOR` type.