The Complete Reference

Part III

Software Development Using Java

This page intentionally left blank.

The Complete Reference

Chapter 25

Java Beans

This chapter provides an overview of an exciting technology that is at the forefront of Java programming: Java Beans. Beans are important, because they allow you to build complex systems from software components. These components may be provided by you or supplied by one or more different vendors. Java Beans defines an architecture that specifies how these building blocks can operate together.

To better understand the value of Beans, consider the following. Hardware designers have a wide variety of components that can be integrated together to construct a system. Resistors, capacitors, and inductors are examples of simple building blocks. Integrated circuits provide more advanced functionality. All of these different parts can be reused. It is not necessary or possible to rebuild these capabilities each time a new system is needed. Also, the same pieces can be used in different types of circuits. This is possible because the behavior of these components is understood and documented.

Unfortunately, the software industry has not been as successful in achieving the benefits of reusability and interoperability. Large applications grow in complexity and become very difficult to maintain and enhance. Part of the problem is that, until recently, there has not been a standard, portable way to write a software component. To achieve the benefits of component software, a component architecture is needed that allows programs to be assembled from software building blocks, perhaps provided by different vendors. It must also be possible for a designer to select a component, understand its capabilities, and incorporate it into an application. When a new version of a component becomes available, it should be easy to incorporate this functionality into existing code. Fortunately, Java Beans provides just such an architecture.

What Is a Java Bean?

A *Java Bean* is a software component that has been designed to be reusable in a variety of different environments. There is no restriction on the capability of a Bean. It may perform a simple function, such as checking the spelling of a document, or a complex function, such as forecasting the performance of a stock portfolio. A Bean may be visible to an end user. One example of this is a button on a graphical user interface. A Bean may also be invisible to a user. Software to decode a stream of multimedia information in real time is an example of this type of building block. Finally, a Bean may be designed to work autonomously on a user's workstation or to work in cooperation with a set of other distributed components. Software to generate a pie chart from a set of data points is an example of a Bean that can execute locally. However, a Bean that provides real-time price information from a stock or commodities exchange would need to work in cooperation with other distributed software to obtain its data.

You will see shortly what specific changes a software developer must make to a class so that it is usable as a Java Bean. However, one of the goals of the Java designers was to make it easy to use this technology. Therefore, the code changes are minimal.

Advantages of Java Beans

A software component architecture provides standard mechanisms to deal with software building blocks. The following list enumerates some of the specific benefits that Java technology provides for a component developer:

- A Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.
- The properties, events, and methods of a Bean that are exposed to an application builder tool can be controlled.
- A Bean may be designed to operate correctly in different locales, which makes it useful in global markets.
- Auxiliary software can be provided to help a person configure a Bean. This software is only needed when the design-time parameters for that component are being set. It does not need to be included in the run-time environment.
- The configuration settings of a Bean can be saved in persistent storage and restored at a later time.
- A Bean may register to receive events from other objects and can generate events that are sent to other objects.

Application Builder Tools

When working with Java Beans, most developers use an *application builder tool,* a utility that enables you to configure a set of Beans, connect them together, and produce a working application. In general, Bean builder tools have the following capabilities.

- A palette is provided that lists all of the available Beans. As additional Beans are developed or purchased, they can be added to the palette.
- A worksheet is displayed that allows the designer to lay out Beans in a graphical user interface. A designer may drag and drop a Bean from the palette to this worksheet.
- Special editors and customizers allow a Bean to be configured. This is the mechanism by which the behavior of a Bean may be adapted for a particular environment.
- Commands allow a designer to inquire about the state and behavior of a Bean. This information automatically becomes available when a Bean is added to the palette.
- Capabilities exist to interconnect Beans. This means that events generated by one component are mapped to method invocations on other components.

- When a collection of Beans has been configured and connected, it is possible to save all of this information in a persistent storage area. At a later time, this information can then be used to restore the state of the application.

Sun provides two Bean application builder tools. The first is the BeanBox, which is part of the Bean Developers Kit (BDK). The BDK is the original builder tool provided by Sun. The second is the new Bean Builder. Because Bean Builder is designed to supplant the BeanBox, Sun has stopped development of the BDK and all new Bean applications will be created using Bean Builder.

Although Bean Builder is the future of Bean development, it is not the sole focus of this chapter. Instead, both BeanBox and Bean Builder are discussed. The reason for this is that Bean Builder requires Java 2, version 1.4. It is incompatible with earlier versions of Java 2. This means that readers of this book using Java 2, version 1.2 or version 1.3 will not be able to use Bean Builder. Instead, they must continue to use the BDK. Further, readers using version 1.4 *cannot* use the BDK because it is not compatible with Java 2, version 1.4. So, if you are using version 1.4, then you *must use* Bean Builder. If you are using a version of Java prior to 1.4, you *must use* the BDK. Thus, both approaches are described here, beginning with the BDK. Keep in mind that the information about Beans, Bean architecture, JAR files, and so on, apply to either Bean development tool.

One other point: At the time of this writing, Java 2, version 1.4 is a released product, but Bean Builder is currently in beta testing. This means that the only way for a 1.4 user to create a Bean application is to do so using latest Bean Builder beta. For this reason, we will not examine its features in depth at this time. However, at the end of this chapter, a general overview is presented and a sample application is created.

Using the Bean Developer Kit (BDK)

The Bean Developer Kit (BDK), available from the JavaSoft site, is a simple example of a tool that enables you to create, configure, and connect a set of Beans. There is also a set of sample Beans with their source code. This section provides step-by-step instructions for installing and using this tool. Remember, the BDK is for use with versions of Java 2 prior to 1.4. For Java 2, v1.4 you must use the Bean Builder Tool described at the end of this chapter.

Note *In this chapter, instructions are provided for a Windows environment. The procedures for a UNIX platform are similar, but some of the commands are different.*

Installing the BDK

The Java 2 SDK must be installed on your machine for the BDK to work. Confirm that the SDK tools are accessible from your environment.

The BDK can then be downloaded from the JavaSoft site (**http://java.sun.com**). It is packaged as one file that is a self-extracting archive. Follow the instructions to install it on your machine. The discussion that follows assumes that the BDK is installed in

a directory called **bdk**. If this is not the case with your system, substitute the proper directory.

Starting the BDK

To start the BDK, follow these steps:

1. Change to the directory **c:\bdk\beanbox**.
2. Execute the batch file called **run.bat**. This causes the BDK to display the three windows shown in Figure 25-1. ToolBox lists all of the different Beans that have been included with the BDK. BeanBox provides an area to lay out and connect the Beans selected from the ToolBox. Properties provides the ability to configure a selected Bean. You may also see a window called Method Tracer, but we won't be using it.

Using the BDK

This section describes how to create an application by using some of the Beans provided with the BDK. First, the **Molecule** Bean displays a three-dimensional view of a molecule. It may be configured to present one of the following molecules: hyaluronic acid, benzene, buckminsterfullerine, cyclohexane, ethane, or water. This component also has methods that allow the molecule to be rotated in space along its X or Y axis.

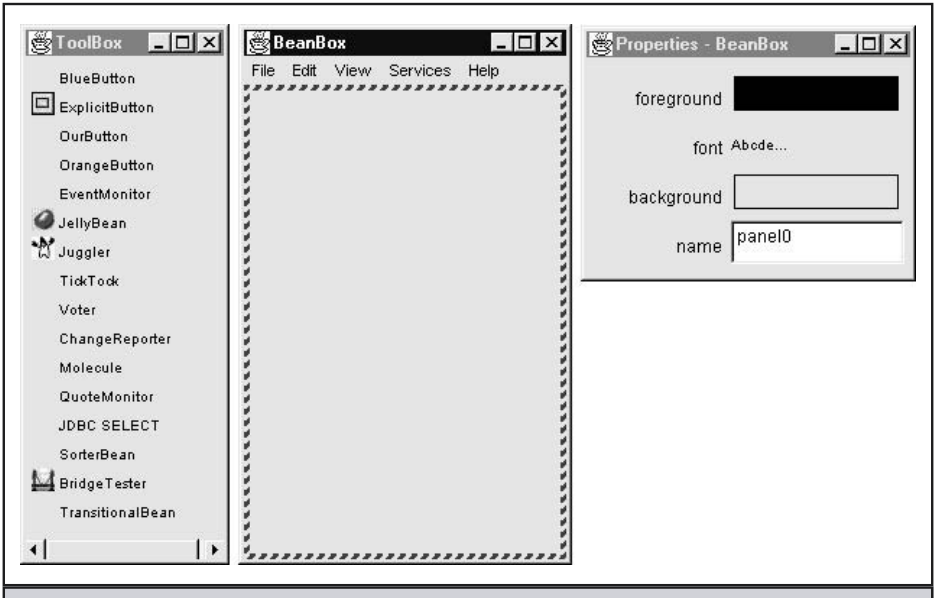

Figure 25-1. *The Bean Developer Kit (BDK)*

Second, the **OurButton** Bean provides a push-button functionality. We will have one button labeled "Rotate X" to rotate the molecule along its X axis and another button labeled "Rotate Y" to rotate the molecule along its Y axis.

Figure 25-2 shows how this application appears.

Create and Configure an Instance of the Molecule Bean

Follow these steps to create and configure an instance of the **Molecule** Bean:

1. Position the cursor on the ToolBox entry labeled **Molecule** and click the left mouse button. You should see the cursor change to a cross.
2. Move the cursor to the BeanBox display area and click the left mouse button in approximately the area where you wish the Bean to be displayed. You should see a rectangular region appear that contains a 3-D display of a molecule. This area is surrounded by a hatched border, indicating that it is currently selected.
3. You can reposition the **Molecule** Bean by positioning the cursor over one of the hatched borders and dragging the Bean.
4. You can change the molecule that is displayed by changing the selection in the Properties window. Notice that the Bean display changes immediately when you change the selected molecule.

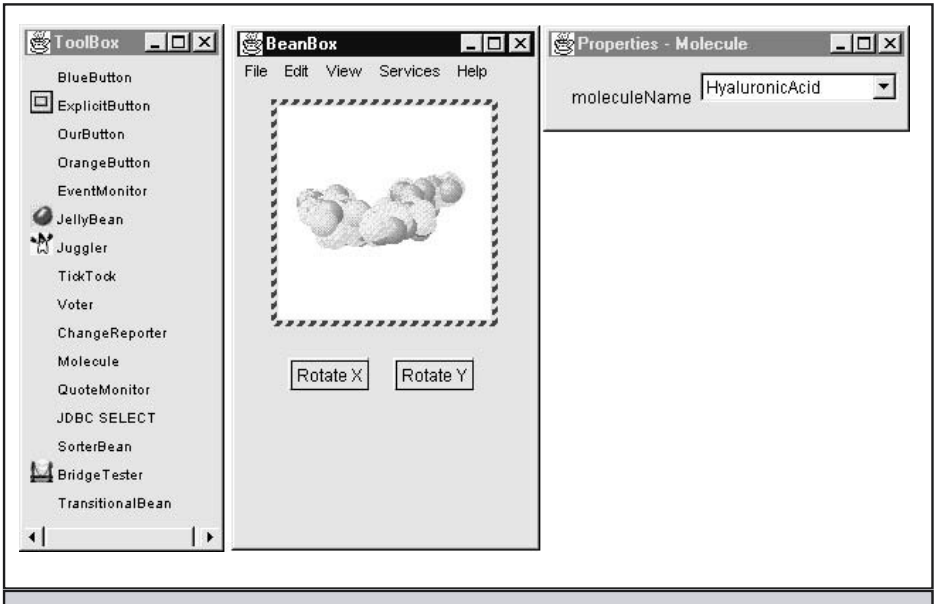

Figure 25-2. *The* Molecule *and* OurButton *Beans*

Create and Configure an Instance of the OurButton Bean

Follow these steps to create and configure an instance of the **OurButton** Bean and connect it to the **Molecule** Bean:

1. Position the cursor on the ToolBox entry labeled **OurButton** and click the left mouse button. You should see the cursor change to a cross.
2. Move the cursor to the BeanBox display area and click the left mouse button in approximately the area where you wish the Bean to be displayed. You should see a rectangular region appear that contains a button. This area is surrounded by a hatched border indicating that it is currently selected.
3. You may reposition the **OurButton** Bean by positioning the cursor over one of the hatched borders and dragging the Bean.
4. Go to the Properties window and change the label of the Bean to "Rotate X". The button appearance changes immediately when this property is changed.
5. Go to the menu bar of the BeanBox and select Edit | Events | action | actionPerformed. You should now see a line extending from the button to the cursor. Notice that one end of the line moves as the cursor moves. However, the other end of the line remains fixed at the button.
6. Move the cursor so that it is inside the **Molecule** Bean display area, and click the left mouse button. You should see the Event Target Dialog dialog box.
7. The dialog box allows you to choose a method that should be invoked when this button is clicked. Select the entry labeled "rotateOnX" and click the OK button. You should see a message box appear very briefly, stating that the tool is "Generating and compiling adaptor class."

Test the application. Each time you press the button, the molecule should move a few degrees around one of its axes.

Now create another instance of the **OurButton** Bean. Label it "Rotate Y" and map its action event to the "rotateY" method of the **Molecule** Bean. The steps to do this are very similar to those just described for the button labeled "Rotate X".

Test the application by clicking these buttons and observing how the molecule moves.

JAR Files

Before developing your own Bean, it is necessary for you to understand JAR (Java Archive) files, because tools such as the BDK expect Beans to be packaged within JAR files. A JAR file allows you to efficiently deploy a set of classes and their associated resources. For example, a developer may build a multimedia application that uses various sound and image files. A set of Beans can control how and when this information is presented. All of these pieces can be placed into one JAR file.

JAR technology makes it much easier to deliver and install software. Also, the elements in a JAR file are compressed, which makes downloading a JAR file much faster than separately downloading several uncompressed files. Digital signatures may also be associated with the individual elements in a JAR file. This allows a consumer to be sure that these elements were produced by a specific organization or individual.

Note *The package **java.util.zip** contains classes that read and write JAR files.*

Manifest Files

A developer must provide a *manifest file* to indicate which of the components in a JAR file are Java Beans. An example of a manifest file is provided in the following listing. It defines a JAR file that contains four **.gif** files and one **.class** file. The last entry is a Bean.

```
Name: sunw/demo/slides/slide0.gif
Name: sunw/demo/slides/slide1.gif
Name: sunw/demo/slides/slide2.gif
Name: sunw/demo/slides/slide3.gif
Name: sunw/demo/slides/Slides.class
Java-Bean: True
```

A manifest file may reference several **.class** files. If a **.class** file is a Java Bean, its entry must be immediately followed by the line "Java-Bean: True".

The JAR Utility

A utility is used to generate a JAR file. Its syntax is shown here:

jar *options files*

Table 25-1 lists the possible options and their meanings. The following examples show how to use this utility.

Creating a JAR File

The following command creates a JAR file named **Xyz.jar** that contains all of the **.class** and **.gif** files in the current directory:

```
jar cf Xyz.jar *.class *.gif
```

If a manifest file such as **Yxz.mf** is available, it can be used with the following command:

```
jar cfm Xyz.jar Yxz.mf *.class *.gif
```

Option	Description
c	A new archive is to be created.
C	Change directories during command execution.
f	The first element in the file list is the name of the archive that is to be created or accessed.
i	Index information should be provided.
m	The second element in the file list is the name of the external manifest file.
M	Manifest file not created.
t	The archive contents should be tabulated.
u	Update existing JAR file.
v	Verbose output should be provided by the utility as it executes.
x	Files are to be extracted from the archive. (If there is only one file, that is the name of the archive, and all files in it are extracted. Otherwise, the first element in the file list is the name of the archive, and the remaining elements in the list are the files that should be extracted from the archive.)
0	Do not use compression.

Table 25-1. *JAR Command Options*

Tabulating the Contents of a JAR File

The following command lists the contents of **Xyz.jar**:

```
jar tf Xyz.jar
```

Extracting Files from a JAR File

The following command extracts the contents of **Xyz.jar** and places those files in the current directory:

```
jar xf Xyz.jar
```

Updating an Existing JAR File

The following command adds the file **file1.class** to **Xyz.jar**:

```
jar -uf Xyz.jar file1.class
```

Recursing Directories

The following command adds all files below **directoryX** to **Xyz.jar**:

```
jar -uf Xyz.jar -C directoryX *
```

Introspection

Introspection is the process of analyzing a Bean to determine its capabilities. This is an essential feature of the Java Beans API, because it allows an application builder tool to present information about a component to a software designer. Without introspection, the Java Beans technology could not operate.

There are two ways in which the developer of a Bean can indicate which of its properties, events, and methods should be exposed by an application builder tool. With the first method, simple naming conventions are used. These allow the introspection mechanisms to infer information about a Bean. In the second way, an additional class is provided that explicitly supplies this information. The first approach is examined here. The second method is described later.

The following sections indicate the design patterns for properties and events that enable the functionality of a Bean to be determined.

Design Patterns for Properties

A *property* is a subset of a Bean's state. The values assigned to the properties determine the behavior and appearance of that component. This section discusses three types of properties: simple, Boolean, and indexed.

Simple Properties

A simple property has a single value. It can be identified by the following design patterns, where N is the name of the property and T is its type.

```
public T getN( );
public void setN(T arg);
```

A read/write property has both of these methods to access its values. A read-only property has only a get method. A write-only property has only a set method.

The following listing shows a class that has three read/write simple properties:

```
public class Box {
  private double depth, height, width;
  public double getDepth( ) {
    return depth;
  }
  public void setDepth(double d) {
    depth = d;
  }
  public double getHeight( ) {
    return height;
  }
  public void setHeight(double h) {
    height = h;
  }
  public double getWidth( ) {
    return width;
  }
  public void setWidth(double w) {
    width = w;
  }
}
```


Boolean Properties

A Boolean property has a value of **true** or **false**. It can be identified by the following design patterns, where N is the name of the property:

public boolean isN( );
public boolean getN( );
public void setN(boolean *value*);

Either the first or second pattern can be used to retrieve the value of a Boolean property. However, if a class has both of these methods, the first pattern is used.

The following listing shows a class that has one Boolean property:

```
public class Line {
  private boolean dotted = false;
  public boolean isDotted( ) {
    return dotted;
  }
  public void setDotted(boolean dotted) {
```

```
    this.dotted = dotted;
  }
}
```

Indexed Properties

An indexed property consists of multiple values. It can be identified by the following design patterns, where N is the name of the property and T is its type:

public T getN(int *index*);
public void setN(int *index*, T *value*);
public T[ ] getN( );
public void setN(T *values*[ ]);

The following listing shows a class that has one read/write indexed property:

```
public class PieChart {
  private double data[ ];
  public double getData(int index) {
    return data[index];
  }
 public void setData(int index, double value) {
    data[index] = value;
  }
  public double[ ] getData( ) {
    return data;
  }
  public void setData(double[ ] values) {
    data = new double[values.length];
    System.arraycopy(values, 0, data, 0, values.length);
  }
}
```

Design Patterns for Events

Beans use the delegation event model that was discussed earlier in this book. Beans can generate events and send them to other objects. These can be identified by the following design patterns, where T is the type of the event:

public void addTListener(TListener *eventListener*);
public void addTListener(TListener *eventListener*) throws TooManyListeners;
public void removeTListener(TListener *eventListener*);

These methods are used by event listeners to register an interest in events of a specific type. The first pattern indicates that a Bean can multicast an event to multiple listeners. The second pattern indicates that a Bean can unicast an event to only one listener. The third pattern is used by a listener when it no longer wishes to receive a specific type of event notification from a Bean.

The following listing outlines a class that notifies other objects when a temperature value moves outside a specific range. The two methods indicated here allow other objects that implement the **TemperatureListener** interface to receive notifications when this occurs.

```
public class Thermometer {
  public void addTemperatureListener(TemperatureListener tl) {
    ...
  }
  public void removeTemperatureListener(TemperatureListener tl) {
    ...
  }
}
```

Methods

Design patterns are not used for naming nonproperty methods. The introspection mechanism finds all of the public methods of a Bean. Protected and private methods are not presented.

Developing a Simple Bean Using the BDK

This section presents an example that shows how to develop a simple Bean and connect it to other components via the BDK.

Our new component is called the **Colors** Bean. It appears as either a rectangle or ellipse that is filled with a color. A color is chosen at random when the Bean begins execution. A public method can be invoked to change it. Each time the mouse is clicked on the Bean, another random color is chosen. There is one **boolean** read/write property that determines the shape.

The BDK is used to lay out an application with one instance of the **Colors** Bean and one instance of the **OurButton** Bean. The button is labeled "Change." Each time it is pressed, the color changes.

Figure 25-3 shows how this application appears.

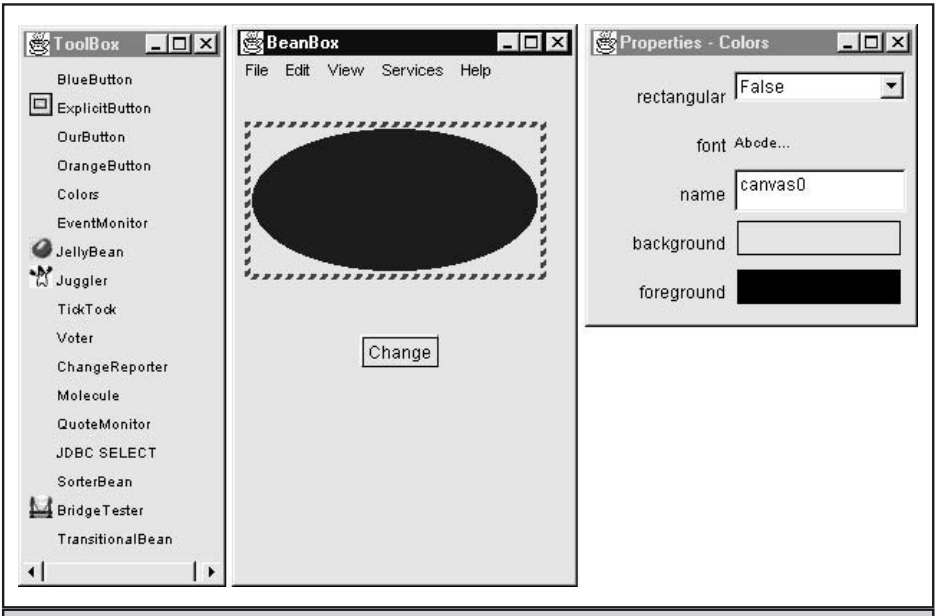

Figure 25-3. *The* Colors *and* OurButton *Beans*

Create a New Bean

Here are the steps that you must follow to create a new Bean:

1. Create a directory for the new Bean.
2. Create the Java source file(s).
3. Compile the source file(s).
4. Create a manifest file.
5. Generate a JAR file.
6. Start the BDK.
7. Test.

The following sections discuss each of these steps in detail.

Create a Directory for the New Bean

You need to make a directory for the Bean. To follow along with this example, create **c:\bdk\demo\sunw\demo\colors**. Then change to that directory.

Create the Source File for the New Bean

The source code for the **Colors** component is shown in the following listing. It is located in the file **Colors.java**.

The **import** statement at the beginning of the file places it in the package named **sunw.demo.colors**. Recall from Chapter 9 that the directory hierarchy corresponds to the package hierarchy. Therefore, this file must be located in a subdirectory named **sunw\demo\colors** relative to the **CLASSPATH** environment variable.

The color of the component is determined by the private **Color** variable **color**, and its shape is determined by the private **boolean** variable **rectangular**.

The constructor defines an anonymous inner class that extends **MouseAdapter** and overrides its **mousePressed( )** method. The **change( )** method is invoked in response to mouse presses. The component is initialized to a rectangular shape of 200 by 100 pixels. The **change( )** method is invoked to select a random color and repaint the component.

The **getRectangular( )** and **setRectangular( )** methods provide access to the one property of this Bean. The **change( )** method calls **randomColor( )** to choose a color and then calls **repaint( )** to make the change visible. Notice that the **paint( )** method uses the **rectangular** and **color** variables to determine how to present the Bean.


```
// A simple Bean.
package sunw.demo.colors;
import java.awt.*;
import java.awt.event.*;
public class Colors extends Canvas {
 transient private Color color;
  private boolean rectangular;
  public Colors() {
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        change();
      }
    });
    rectangular = false;
    setSize(200, 100);
    change();
  }
  public boolean getRectangular() {
    return rectangular;
  }
  public void setRectangular(boolean flag) {
    this.rectangular = flag;
    repaint();
  }
  public void change() {
```

```
    color = randomColor();
    repaint();
  }
  private Color randomColor() {
    int r = (int)(255*Math.random());
    int g = (int)(255*Math.random());
    int b = (int)(255*Math.random());
    return new Color(r, g, b);
  }
  public void paint(Graphics g) {
    Dimension d = getSize();
    int h = d.height;
    int w = d.width;
    g.setColor(color);
    if(rectangular) {
      g.fillRect(0, 0, w-1, h-1);
    }
    else {
      g.fillOval(0, 0, w-1, h-1);
    }
  }
}
```

Compile the Source Code for the New Bean

Compile the source code to create a class file. Type the following:

```
javac Colors.java.
```

Create a Manifest File

You must now create a manifest file. First, switch to the **c:\bdk\demo** directory. This is the directory in which the manifest files for the BDK demos are located. Put the source code for your manifest file in the file **colors.mft**. It is shown here:

```
Name: sunw/demo/colors/Colors.class
Java-Bean: True
```

This file indicates that there is one **.class** file in the JAR file and that it is a Java Bean. Notice that the **Colors.class** file is in the package **sunw.demo.colors** and in the subdirectory **sunw\demo\colors** relative to the current directory.

Generate a JAR File

Beans are included in the ToolBox window of the BDK only if they are in JAR files in the directory **c:\bdk\jars**. These files are generated with the jar utility. Enter the following:

```
jar cfm ..\jars\colors.jar colors.mft sunw\demo\colors\*.class
```

This command creates the file **colors.jar** and places it in the directory **c:\bdk\jars**. (You may wish to put this in a batch file for future use.)

Start the BDK

Change to the directory **c:\bdk\beanbox** and type **run**. This causes the BDK to start. You should see three windows, titled ToolBox, BeanBox, and Properties. The ToolBox window should include an entry labeled "Colors" for your new Bean.

Create an Instance of the Colors Bean

After you complete the preceding steps, create an instance of the **Colors** Bean in the BeanBox window. Test your new component by pressing the mouse anywhere within its borders. Its color immediately changes. Use the Properties window to change the **rectangular** property from **false** to **true**. Its shape immediately changes.

Create and Configure an Instance of the OurButton Bean

Create an instance of the **OurButton** Bean in the BeanBox window. Then follow these steps:

1. Go to the Properties window and change the label of the Bean to "Change". You should see that the button appearance changes immediately when this property is changed.
2. Go to the menu bar of the BeanBox and select Edit | Events | action | actionPerformed.
3. Move the cursor so that it is inside the **Colors** Bean display area, and click the left mouse button. You should see the Event Target Dialog dialog box.
4. The dialog box allows you to choose a method that should be invoked when this button is clicked. Select the entry labeled "change" and click the OK button. You should see a message box appear very briefly, stating that the tool is "Generating and compiling adaptor class."
5. Click on the button. You should see the color change.

You might want to experiment with the **Colors** Bean a bit before moving on.

Using Bound Properties

A Bean that has a bound property generates an event when the property is changed. The event is of type **PropertyChangeEvent** and is sent to objects that previously registered an interest in receiving such notifications.

The **TickTock** Bean is supplied with the BDK. It generates a property change event every N seconds. N is a property of the Bean that can be changed via the Properties window of the BDK. The next example builds an application that uses the **TickTock** Bean to automatically control the **Colors** Bean. Figure 25-4 shows how this application appears.

Steps

For this example, start the BDK and create an instance of the **Colors** Bean in the BeanBox window.

Create an instance of the **TickTock** Bean. The Properties window should show one property for this component. It is "Interval" and its initial value is 5. This represents the number of seconds that elapse between property change events generated by the **TickTock** Bean. Change the value to 1.

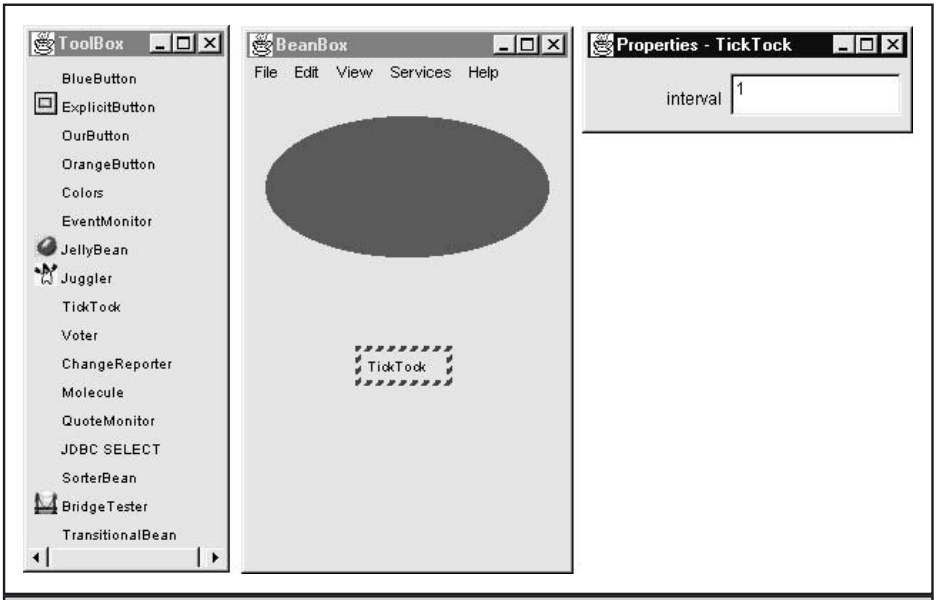

Figure 25-4. *The* Colors *and* TickTock *Beans*

Now you need to map events generated by the **TickTock** Bean into method calls on the **Colors** Bean. Follow these steps:

1. Go to the menu bar of the BeanBox and select Edit | Events | propertyChange | propertyChange. You should now see a line extending from the button to the cursor.
2. Move the cursor so that it is inside the **Colors** Bean display area, and click the left mouse button. You should see the Event Target Dialog dialog box.
3. The dialog box allows you to choose a method that should be invoked when this event occurs. Select the entry labeled "change" and click the OK button. You should see a message box appear very briefly, stating that the tool is "Generating and compiling adaptor class."

You should now see the color of your component change every second.

Using the BeanInfo Interface

In our previous examples, design patterns were used to determine the information that was provided to a Bean user. This section describes how a developer can use the **BeanInfo** interface to explicitly control this process.

This interface defines several methods, including these:

```
PropertyDescriptor[ ] getPropertyDescriptors( )
EventSetDescriptor[ ] getEventSetDescriptors( )
MethodDescriptor[ ] getMethodDescriptors( )
```

They return arrays of objects that provide information about the properties, events, and methods of a Bean. By implementing these methods, a developer can designate exactly what is presented to a user.

SimpleBeanInfo is a class that provides default implementations of the **BeanInfo** interface, including the three methods just shown. You may extend this class and override one or more of them. The following listing shows how this is done for the **Colors** Bean that was developed earlier. **ColorsBeanInfo** is a subclass of **SimpleBeanInfo**. It overrides **getPropertyDescriptors( )** in order to designate which properties are presented to a Bean user. This method creates a **PropertyDescriptor** object for the **rectangular** property. The **PropertyDescriptor** constructor that is used is shown here:

```
PropertyDescriptor(String property, Class beanCls)
    throws IntrospectionException
```

Here, the first argument is the name of the property, and the second argument is the class of the Bean.

```
// A Bean information class.
package sunw.demo.colors;
import java.beans.*;
public class ColorsBeanInfo extends SimpleBeanInfo {
  public PropertyDescriptor[] getPropertyDescriptors() {
    try {
      PropertyDescriptor rectangular = new
         PropertyDescriptor("rectangular", Colors.class);
      PropertyDescriptor pd[] = {rectangular};
      return pd;
    }
    catch(Exception e) {
    }
    return null;
  }
}
```

You must compile this file from the **BDK\demo** directory or set **CLASSPATH** so that it includes **c:\bdk\demo**. If you don't, the compiler won't find the **Colors.class** file properly. After this file is successfully compiled, the **colors.mft** file can be updated, as shown here:

```
Name: sunw/demo/colors/ColorsBeanInfo.class
Name: sunw/demo/colors/Colors.class
Java-Bean: True
```

Use the JAR tool to create a new **colors.jar** file. Restart the BDK and create an instance of the **Colors** Bean in the BeanBox.

The introspection facilities are designed to look for a **BeanInfo** class. If it exists, its behavior explicitly determines the information that is presented to a Bean user. Otherwise, design patterns are used to infer this information.

Figure 25-5 shows how the Properties window now appears. Compare it with Figure 24-3. You can see that the properties inherited from **Component** are no longer presented for the **Colors** Bean. Only the **rectangular** property appears.

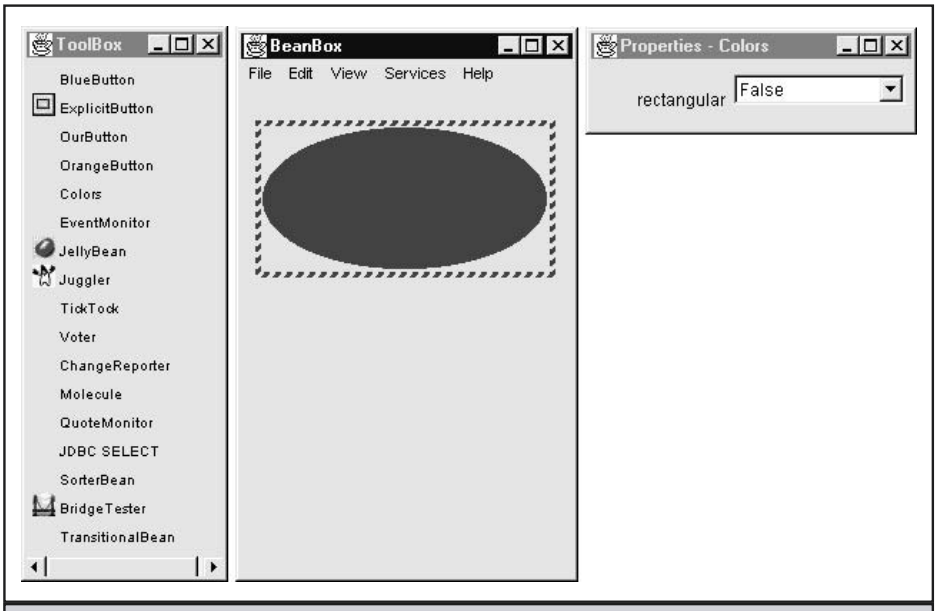

Figure 25-5. *Using the* ColorsBeanInfo *Class*

Constrained Properties

A Bean that has a *constrained* property generates an event when an attempt is made to change its value. The event is of type **PropertyChangeEvent**. It is sent to objects that previously registered an interest in receiving such notifications. Those other objects have the ability to veto the proposed change. This capability allows a Bean to operate differently according to its run-time environment. A full discussion of constrained properties is beyond the scope of this book.

Persistence

Persistence is the ability to save a Bean to nonvolatile storage and retrieve it at a later time. The information that is particularly important are the configuration settings.

Let us first see how the BDK allows you to save a set of Beans that have been configured and connected together to form an application. Recall our previous example involving both the **Colors** and **TickTock** Beans. The **rectangular** property of the **Colors** Bean was changed to **true**, and the **interval** property of the **TickTock** Bean was changed to one second. These changes can be saved.

To save the application, go to the menu bar of the BeanBox and select File | Save. A dialog box should appear, allowing you to specify the name of a file to which the Beans and their configuration parameters should be saved. Supply a filename and click the OK button on that dialog box. Exit from the BDK.

Start the BDK again. To restore the application, go to the menu bar of the BeanBox and select File | Load. A dialog box should appear, allowing you to specify the name of the file from which an application should be restored. Supply the name of the file in which the application was saved, and click the OK button. Your application should now be functioning. Confirm that the **rectangular** property of the **Colors** Bean is **true** and that the **interval** property for the **TickTock** Bean is equal to one second.

The object serialization capabilities provided by the Java class libraries are used to provide persistence for Beans. If a Bean inherits directly or indirectly from **java.awt.Component**, it is automatically serializable, because that class implements the **java.io.Serializable** interface. If a Bean does not inherit an implementation of the **Serializable** interface, you must provide this yourself. Otherwise, containers cannot save the configuration of your component.

The **transient** keyword can be used to designate data members of a Bean that should not be serialized. The **color** variable of the **Colors** class is an example of such an item.

Customizers

The Properties window of the BDK allows a developer to modify the properties of a Bean. However, this may not be the best user interface for a complex component with many interrelated properties. Therefore, a Bean developer can provide a *customizer* that helps another developer configure this software. A customizer can provide a step-by-step guide through the process that must be followed to use the component in a specific context. Online documentation can also be provided. A Bean developer has great flexibility to develop a customizer that can differentiate his or her product in the marketplace.

The Java Beans API

The Java Beans functionality is provided by a set of classes and interfaces in the **java.beans** package. This section provides a brief overview of its contents. Table 25-2 lists the interfaces in **java.beans** and provides a brief description of their functionality. Table 25-3 lists the classes in **java.beans**.

Interface	Description
AppletInitializer	Methods in this interface are used to initialize Beans that are also applets.
BeanInfo	This interface allows a designer to specify information about the properties, events, and methods of a Bean.
Customizer	This interface allows a designer to provide a graphical user interface through which a Bean may be configured.
DesignMode	Methods in this interface determine if a Bean is executing in design mode.
ExceptionListener	A method in this interface is invoked when an exception has occurred. (Added by Java 2, version 1.4.)
PropertyChangeListener	A method in this interface is invoked when a bound property is changed.
PropertyEditor	Objects that implement this interface allow designers to change and display property values.
VetoableChangeListener	A method in this interface is invoked when a constrained property is changed.
Visibility	Methods in this interface allow a Bean to execute in environments where a graphical user interface is not available.

Table 25-2. *The Interfaces Defined in* java.beans

Class	Description
BeanDescriptor	This class provides information about a Bean. It also allows you to associate a customizer with a Bean.
Beans	This class is used to obtain information about a Bean.

Table 25-3. *The Classes Defined in* java.beans

Class	Description
DefaultPersistenceDelegate	A concrete subclass of **PersistenceDelegate**. (Added by Java 2, version 1.4.)
Encoder	Encodes the state of a set of Beans. Can be used to write this information to a stream. (Added by Java 2, version 1.4.)
EventHandler	Supports dynamic event listener creation. (Added by Java 2, version 1.4.)
EventSetDescriptor	Instances of this class describe an event that can be generated by a Bean.
Expression	Encapsulates a call to a method that returns a result. (Added by Java 2, version 1.4.)
FeatureDescriptor	This is the superclass of the **PropertyDescriptor**, **EventSetDescriptor**, and **MethodDescriptor** classes.
IndexedPropertyDescriptor	Instances of this class describe an indexed property of a Bean.
IntrospectionException	An exception of this type is generated if a problem occurs when analyzing a Bean.
Introspector	This class analyzes a Bean and constructs a **BeanInfo** object that describes the component.
MethodDescriptor	Instances of this class describe a method of a Bean.
ParameterDescriptor	Instances of this class describe a method parameter.
PersistenceDelegate	Handles the state information of an object. (Added by Java 2, version 1.4.)
PropertyChangeEvent	This event is generated when bound or constrained properties are changed. It is sent to objects that registered an interest in these events and implement either the **PropertyChangeListener** or **VetoableChangeListener** interfaces.

Table 25-3. *The Classes Defined in* java.beans (continued)

Class	Description
PropertyChangeListenerProxy	Extends **EventListenerProxy** and implements **PropertyChangeListener**. (Added by Java 2, version 1.4.)
PropertyChangeSupport	Beans that support bound properties can use this class to notify **PropertyChangeListener** objects.
PropertyDescriptor	Instances of this class describe a property of a Bean.
PropertyEditorManager	This class locates a **PropertyEditor** object for a given type.
PropertyEditorSupport	This class provides functionality that can be used when writing property editors.
PropertyVetoException	An exception of this type is generated if a change to a constrained property is vetoed.
SimpleBeanInfo	This class provides functionality that can be used when writing **BeanInfo** classes.
Statement	Encapsulates a call to a method. (Added by Java 2, version 1.4.)
VetoableChangeListenerProxy	Extends **EventListenerProxy** and implements **VetoableChangeListener**. (Added by Java 2, version 1.4.)
VetoableChangeSupport	Beans that support constrained properties can use this class to notify **VetoableChangeListener** objects.
XMLDecoder	Used to read a Bean from an XML document. (Added by Java 2, version 1.4.)
XMLEncoder	Used to write a Bean to an XML document. (Added by Java 2, version 1.4.)

Table 25-3. *The Classes Defined in* java.beans (continued)

A complete discussion of these classes and interfaces is beyond the scope of this book. However, the following program illustrates the **Introspector**, **BeanDescriptor**,

PropertyDescriptor, and **EventSetDescriptor** classes and the **BeanInfo** interface. It lists the properties and events of the **Colors** Bean that was developed earlier in this chapter.

```
// Show properties and events.
package sunw.demo.colors;
import java.awt.*;
import java.beans.*;
public class IntrospectorDemo {
  public static void main(String args[]) {
    try {
      Class c = Class.forName("sunw.demo.colors.Colors");
      BeanInfo beanInfo = Introspector.getBeanInfo(c);
      BeanDescriptor beanDescriptor = beanInfo.getBeanDescriptor();

      System.out.println("Bean name = " +
                         beanDescriptor.getName());

      System.out.println("Properties:");
      PropertyDescriptor propertyDescriptor[] =
        beanInfo.getPropertyDescriptors();
      for(int i = 0; i < propertyDescriptor.length; i++) {
        System.out.println("\t" + propertyDescriptor[i].getName());
      }

      System.out.println("Events:");
      EventSetDescriptor eventSetDescriptor[] =
        beanInfo.getEventSetDescriptors();
      for(int i = 0; i < eventSetDescriptor.length; i++) {
        System.out.println("\t" + eventSetDescriptor[i].getName());
      }
    }
    catch(Exception e) {
      System.out.println("Exception caught. " + e);
    }
  }
}
```

The output from this program is the following:

```
Bean name = Colors
Properties:
```

```
            rectangular
Events:
            propertyChange
            component
            mouseMotion
            mouse
            hierarchy
            key
            focus
            hierarchyBounds
            inputMethod
```

Using Bean Builder

As explained at the start of the chapter, the BDK is not compatible with Java 2, version 1.4. Instead, 1.4 users will need to use the new Bean Builder tool for Bean development. At the time of this writing, Bean Builder is available only as a beta release, and its final form and feature set are subject to change. However, because it is the tool that Java 2, version 1.4 users must use to develop Beans, an overview of Bean Builder is presented here. (Subsequent editions of this book will cover Bean Builder in detail after it is a released product.) Keep in mind that the basic Bean information, such as introspection, described earlier, also applies to Beans used by Bean Builder. Bean Builder is available from **http://java.sun.com**.

Bean Builder is similar to the BeanBox offered by the BDK, except that it is more powerful and sophisticated. Its operation is also similar to the BeanBox except that it is easier to use. Perhaps the most striking feature of Bean Builder is that it supports two separate modes of operation: design and test. In *design mode*, you construct a Bean-based application, adding the various components, and wiring them together. In *test mode*, also called *run-time mode*, the application is executed and all of the components are live. Thus, it is extremely easy to construct and then test your application. Futhermore, you switch between these two modes by checking or clearing a single check box.

Bean Builder provides the three windows shown in Figure 25-6. The top (main) window holds the current palette set. This includes a default palette from which you can choose various user-interface objects, such as buttons, scroll bars, lists, and menus. These are Swing rather than AWT objects. (You will find an overview of Swing in Chapter 26, but no knowledge of Swing is required to follow along with the example developed later in this section.) You can also load other palettes and JAR files. Each component has associated with it a set of properties. You can examine and set these using the Property Inspector window provided by Bean Builder. The third window,

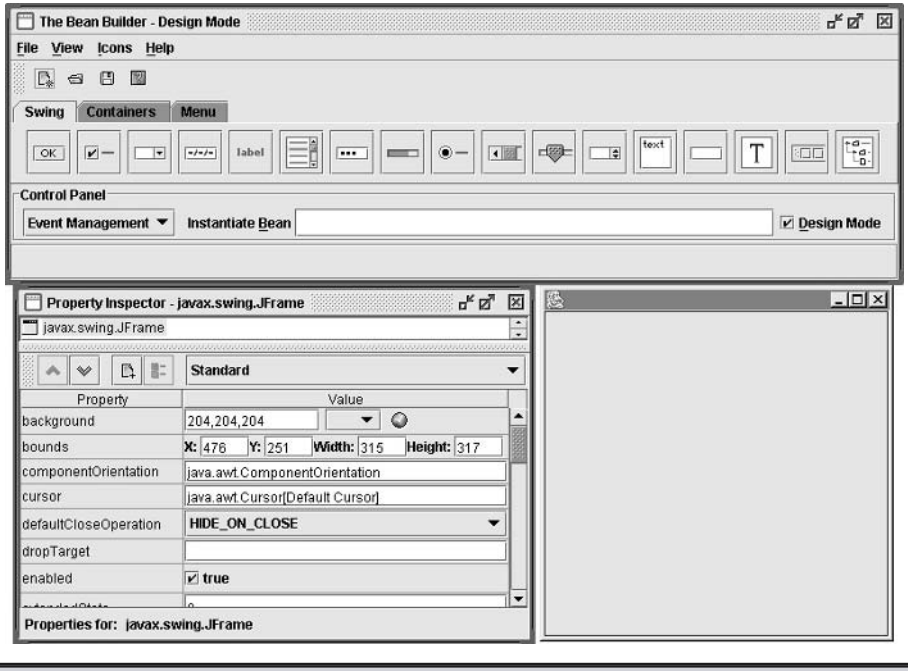

Figure 25-6. *The Bean Builder Windows*

called the design window (or, *designer* for short), is the window in which you will assemble various components into an application.

In general, to build an application, you will select items from a palette and instantiate them on the designer, setting their properties as necessary by using the Property Inspector window. Once you have assembled the components, you will wire them together by dragging a line from one to another. In the process, you will define the input and output methods that will be called, and what action causes them to be called. For example, you might wire a push button to a text field, specifying that when the push button is pressed, the text field will be cleared.

Building a Simple Bean Builder Application

It is really quite easy to build an application using Bean Builder. In this section, we will walk through the construction of a very simple one that contains a label, a slider control, and a scroll bar. When the slider control is moved, the scroll bar is also moved by the same amount, and vice versa. Thus, moving one causes the other to move, too. Once you have completed this walk through, you will be able to easily build other applications on your own.

First, create a new project by selecting New from the File menu. Next, select **javax.swing.JFrame** in the list at the top of the Property Inspector window. **JFrame** is the top-level Swing class for the design window. Next, scroll down in the Property Inspector window until you find **title**. Change the title to "A Bean Builder App". Your screen should look like the one shown in Figure 25-7.

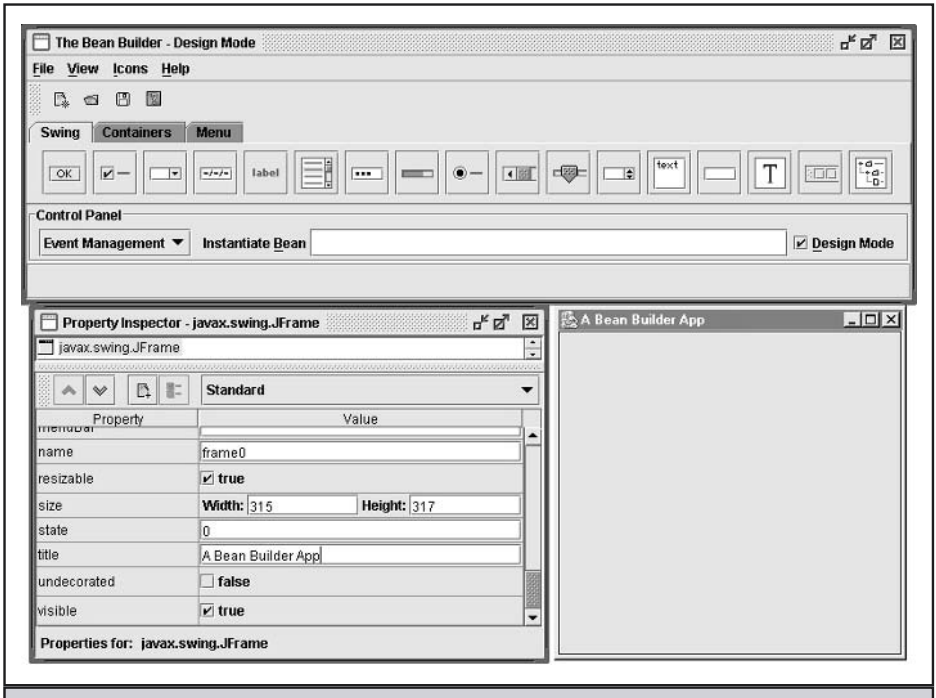

Figure 25-7. *A new Bean Builder Application*

Next, we will add a label to the design. Click on the label button in the Swing palette. This instantiates a **JLabel** object, which is the Swing class for a label. Then, move the mouse to the designer and outline a rectangle near the top of the window. This defines were the text will go. Then, using the Property Inspector window, find the **text** entry. Change it to "Move slider or scroll bar." After you do this, your screen will look like Figure 25-8. Now, find the **horizontalAlignment** field in the Property Inspector and change its value to CENTER. This will center the text within the label.

Next, select a slider from the palette and add it to the designer. Then, add a scroll bar. The slider is an instance of the Swing class **JSlider** and the scroll bar is an instance

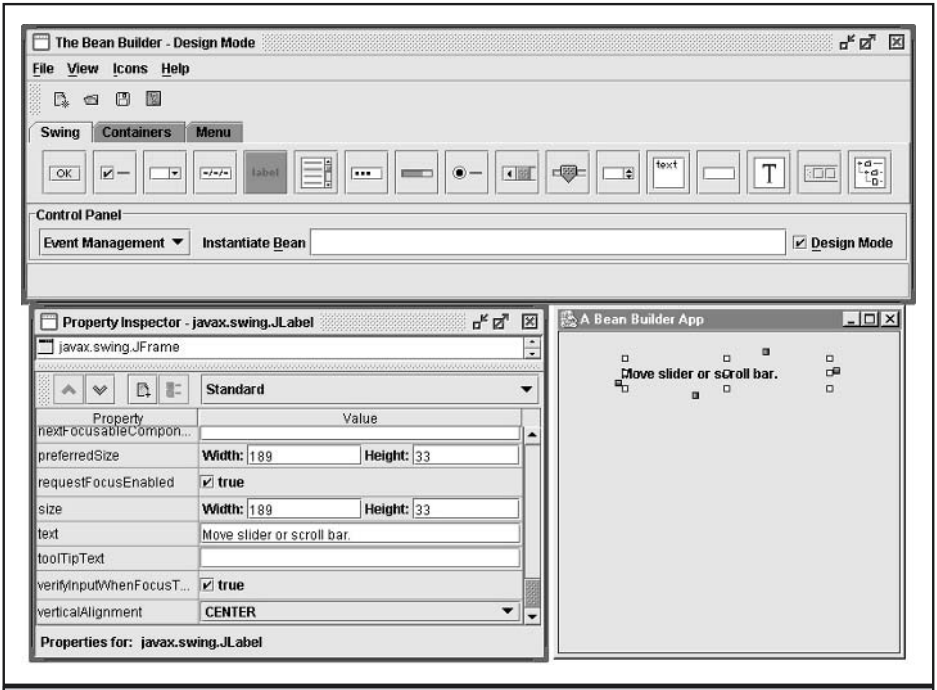

Figure 25-8. *After adding a label*

of the Swing class **JScrollbar**. By default, both the slider and the scroll bar have the same range (0 to 100), so the value of one will translate directly to the value of the other. To make your application look like the one in this book, position them as shown in Figure 25-9.

Now it is time to wire the components together. To do this, you will position the mouse pointer over one of the *connection handles*, then drag a "wire" from the connection handle on one component to a connection handle on another component. The component at which you start is the source of some event and the component at which you end is the recipient of the event. Each component has four connection

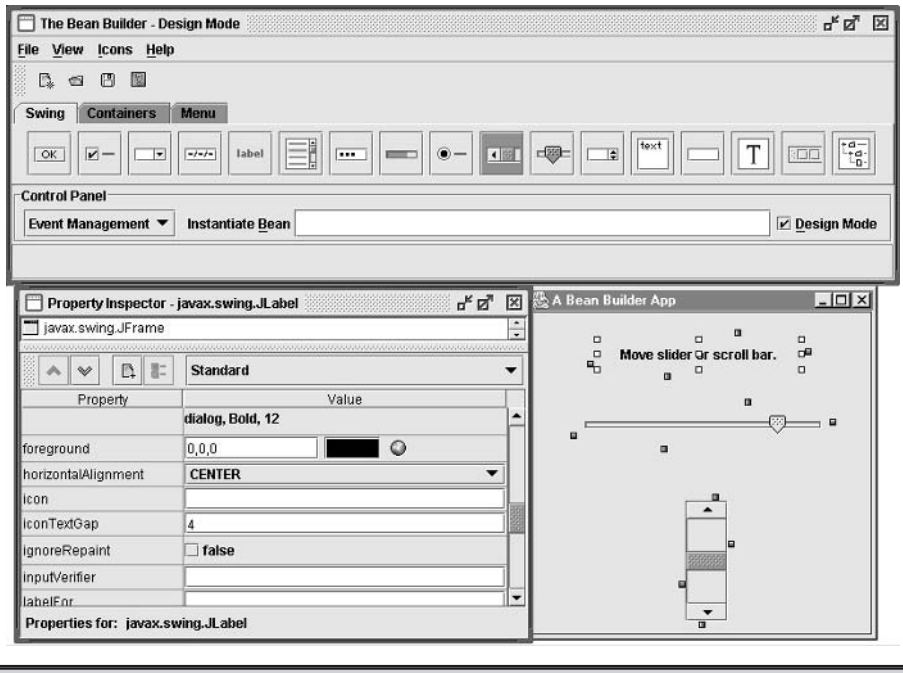

Figure 25-9. *The design window after added the slider and scroll bar*

handles, and it doesn't matter which one you choose. Begin by wiring a connection from the slider to the scroll bar, as shown in Figure 25-10.

After you have completed the connection, the Interaction Wizard will appear. It lets you specify how the two components communicate. In this case, you will define what takes place when the slider is moved. On the first page you will select the event method that will be called when the source object (in this case, the slider) changes position. First, select the Event Adapter radio button (if it is not already selected).

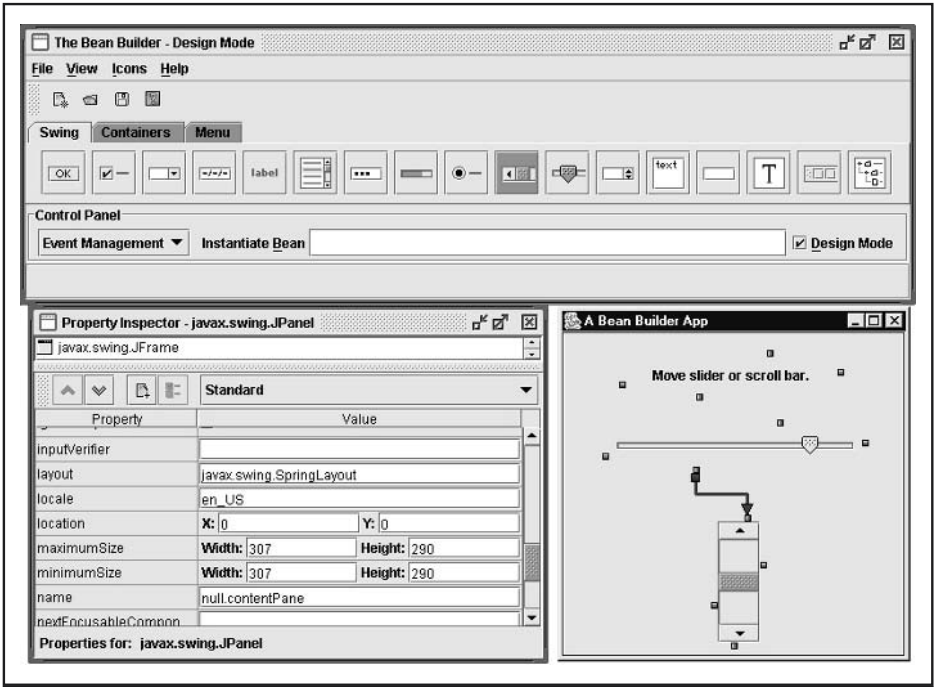

Figure 25-10. *Wiring a connection from the slider to the scroll bar*

Then, select **change** in the Event Sets list. In Event Methods, **stateChanged(ChangeEvent)** should already be selected. Your screen will look like Figure 25-11.

Press Next. You will now select the method on the target object (in this case, the scroll bar) that you want called when the source object changes. In this case, select the **JScrollbar** method **setValue(int)**. It sets the current position of the scroll bar. Your screen will look like Figure 25-12.

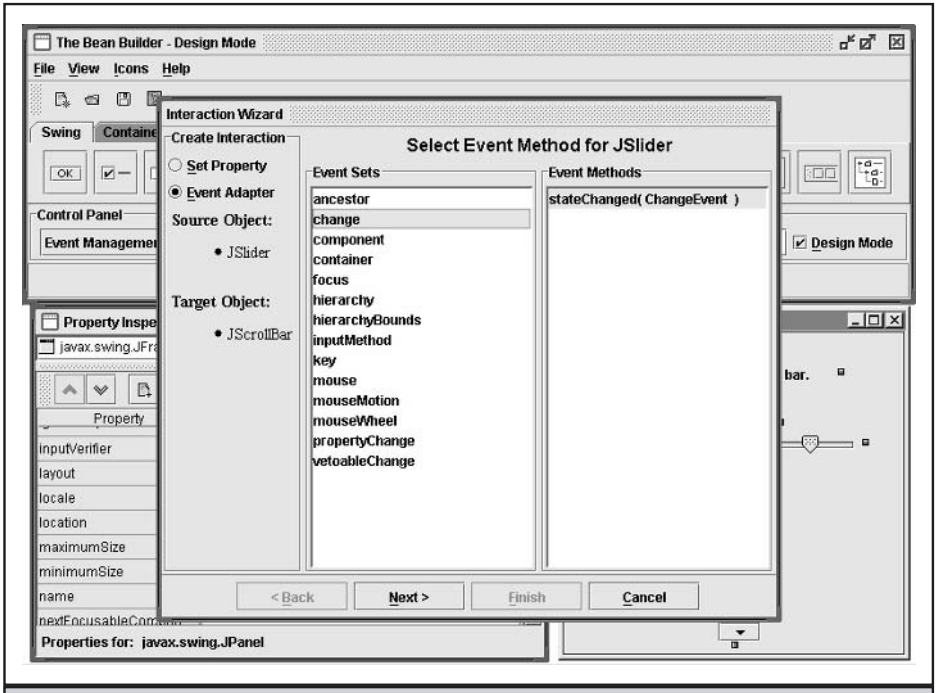

Figure 25-11. *The first page of the Interaction Wizard*

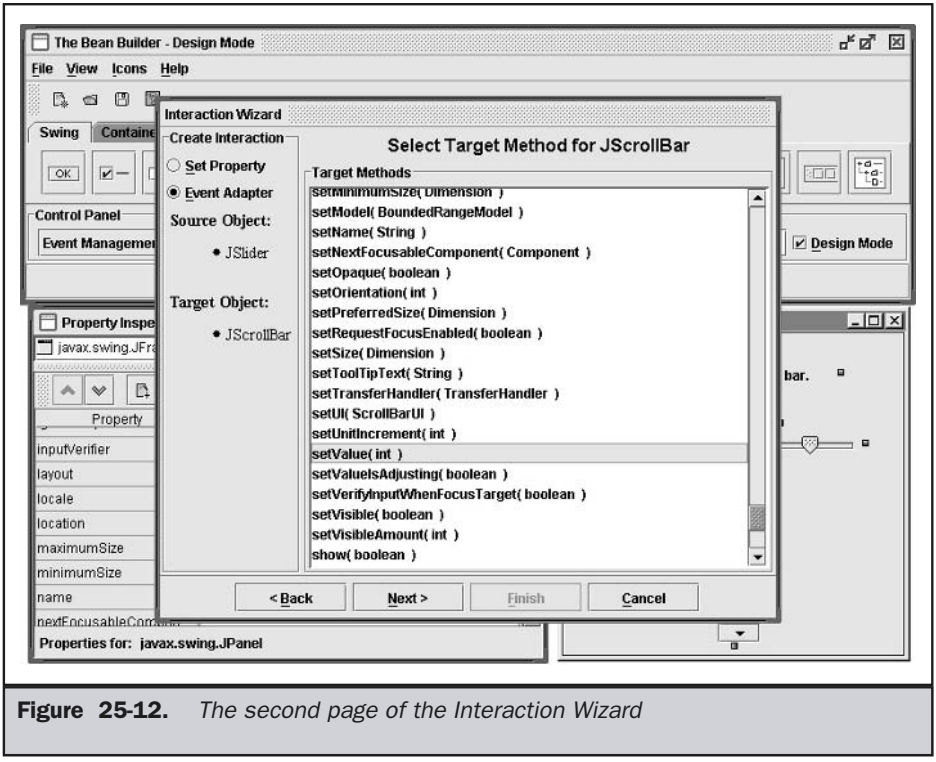

Figure 25-12. *The second page of the Interaction Wizard*

Press Next. Now, select the "getter" method that will supply the argument to **setValue( )**. In this case, it will be **JSlider**'s **getValue( )** method, which returns the current position of the slider. A "getter" is a method that uses the *get* design pattern. Your screen will look like Figure 25-13. Now, press finish. This completes the connection. Now, each time the slider changes, the **setValue( )** method of the scroll bar is called with an argument supplied by the **getValue( )** method of the slider. Thus, moving the slider also causes the scroll bar to move.

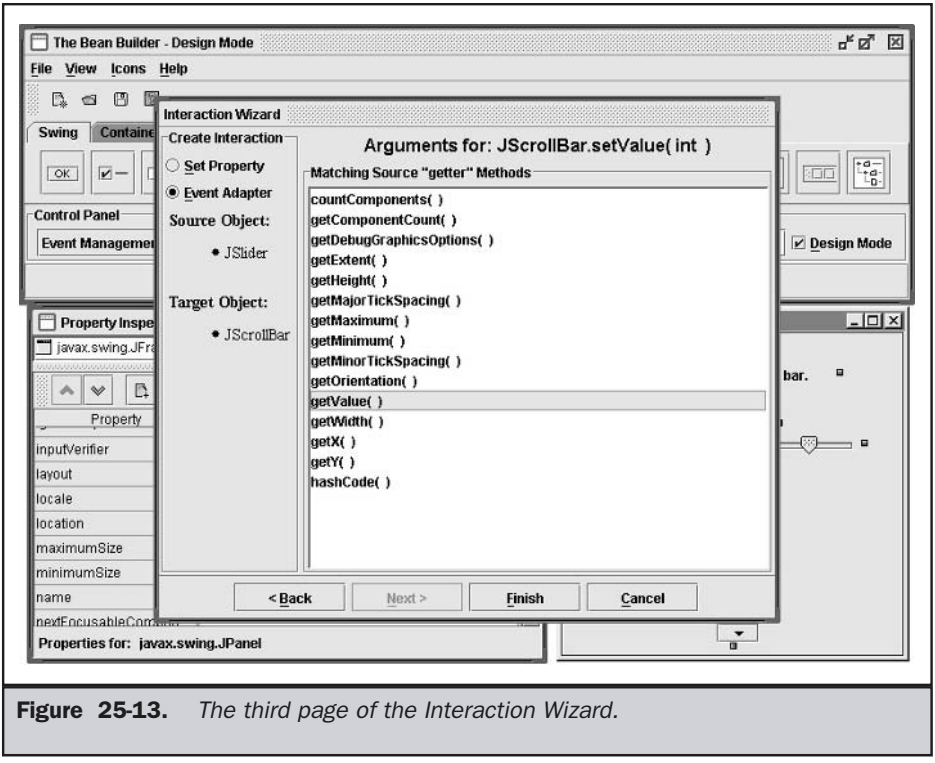

Figure 25-13. *The third page of the Interaction Wizard.*

Next, repeat this process, except this time, wire the connection from the scroll bar to the slider box.

Finally, test the application. To do this, uncheck the Design Mode check box. This causes the application to execute, as shown in Figure 25-14. Try moving the slider box. When it moves, the scroll bar automatically moves, too. This is because of the connection that we wired from the slider to the scroll bar. Assuming that you also wired the reverse connection, moving the scroll bar will cause the slider to move.

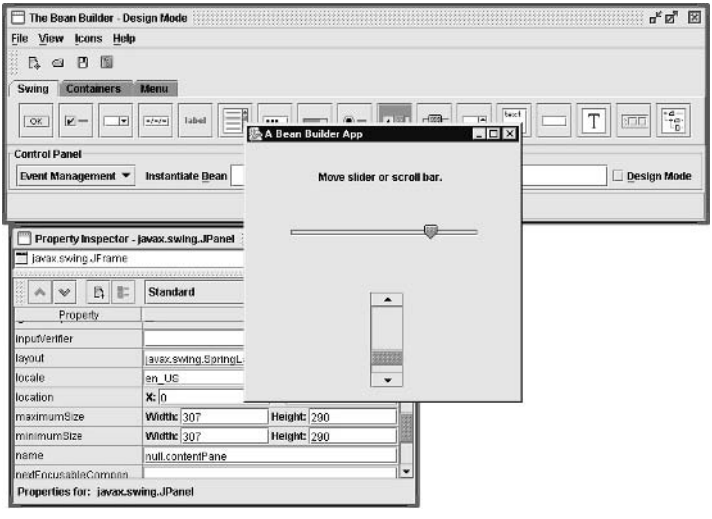

Figure 25-14. *The Bean Builder application executing*

You can save your application by selecting Save in the file menu.

The Bean Builder is a powerful, yet easy to use development tool. If Bean development is in your future, you will want to master its features. The best way to do this is to create a number of sample Bean applications. Also, try creating your own Beans and loading them into the palette. (To do so, create a JAR file containing your Beans, as described earlier.)