

## *Building run-time bean classes*

In the last chapter, you learned how to construct beans through a few examples. These beans had simple properties and the custom events designed to connect them were not complicated. In real-world applications, however, not all beans can be that simple.

In this chapter, you'll learn about design and implementation of run-time bean classes for complex and large beans. We'll first discuss the issues involved in the design of properties, methods, and events. Then, we'll address implementation issues and describe how to implement bound and constrained properties and serialization.

To discuss the design and implementation of run-time bean classes, we'll use data visualization as the problem domain in our examples. Data visualization applications can be fairly large and complex. We'll expose you to a suite of beans that let you perform functions such as image viewing, manipulation, and plotting. The design and implementation details of these beans are available in part II of this book.

Before you start building beans, you need to decide if your problem is suitable for beans. Not all problems can be turned into beans. For example, if you plan to develop a library of data-compression algorithms, you would be better off building a class library. On the other hand, if your aim is to develop a JPEG compressor, building a JPEG bean is a better choice. Let's explore why.

### *4.1 Class libraries versus beans*

Reusability is an obvious goal for both beans and class libraries. However, designing for reusability is not an easy task, because it's hard to generalize requirements. Whatever your bean or class library has to offer may not exactly match your users' requirements.

In class libraries, you have to design APIs as interfaces to the outside world. With JavaBeans, on the other hand, you have to design properties, methods, and events to interface with execution environments and other beans.

A major difference between class libraries and beans is how you use them. You can use a class in the following ways:

- 1 Instantiate objects directly from the class.
- 2 Subclass to extend functionality and then construct objects from that subclass.
- 3 Directly use the static methods in the class.

Beans, on the other hand, are typically instantiated. While it is possible to extend the functionality of a bean by subclassing, you lose the benefits of JavaBeans.

As mentioned in the previous chapter, beans are building blocks. In other words, functionality is distributed over different beans. When you glue beans, functionality gets added. Therefore, if you need to extend the functionality of an existing bean, you would connect it to another bean that has what you want. However, adding functionality this way may not be as easy as it sounds.

From our prior discussion, it is obvious that you need to be a Java programmer to use class libraries. To use beans, you don't need to have Java programming skills if you intend to build applications visually.

Class libraries tend to be more general and often do not provide a complete solution to a problem. Typically, you can't run a class library on its own. Beans tend to be specific to a problem and typically provide a complete solution. While a JavaBean may not run on as a stand-alone application, it does run independently in builder tools. So, JavaBeans will have the code that is needed to run them. In case of class libraries, users provide the necessary code to execute them.

In view of the above, you choose beans over class libraries if the functionality can be completely defined and is clear-cut. In order for beans to be useful, their design may require much more thought and foresight than class libraries do.

## *4.2 Designing beans*

When your problem domain is large, you can create one large bean to provide a complete solution. Such a bean is not often reusable. Therefore, you often need to divide your problem into a number of subproblems, each of which can be a bean. While having many beans may be better from the reusability viewpoint, it involves additional overhead. Furthermore, you would have to design bean connection mechanisms to hook them.

The issues involved in designing run-time bean classes are slightly different from those of classes in libraries. While designing beans, you have to consider the bean's design-time operation as well. As mentioned before, design-time operations include customization and connection.

### *4.2.1 Designing properties*

We know that properties are used for configuring beans. Any entity that determines the structure of a bean has to be made into a property. In the case of visible beans, this may include color and size. While a bean's functionality is a major factor in deciding the properties a bean exposes, it is sometimes necessary to consider some design time factors as well. So, if you intend to develop special purpose customizers, you may need to design easily edited properties. To achieve this, you might have to combine or split properties.

In the Pie Chart bean, for instance, the position (which is a point) property is split into two values: an x and y increment. This makes it easier to customize the Pie Chart. By clicking an increment button, you can move the pie vertically or horizontally to the desired position.

Deciding a property's data type is often very important. Builder tools typically provide property editors for primitive data types and a few commonly used reference types such as font and color. If the property type is a class you developed, you'll have to provide the property editor. This means that you would have to write additional code. As you'll find out in the "Building property editors and customizers" chapter, developing a custom property editor is not always easy.

Properties often depend on each other. For example, the font size in a plotter bean depends on the size of the viewport. And sometimes, a property in one bean may depend on a property in another bean. We know from the JavaBean specs that such properties are categorized as bound or constrained. You need to identify such properties. In the case of a bound property, you have to anticipate whether properties in other beans can be dependent. Likewise, in the case of a constrained property, you have to anticipate whether other beans can veto the property change. We'll see how to implement such properties later in this chapter.

### *4.2.2 Designing methods*

As beans are typically developed independently of one another, a bean cannot directly invoke a method in another. As shown in chapter 2, adapters are used for this purpose. Builder tools construct adapters once the visual connection is made with an event. The adapter first registers with the source bean. When the event occurs, the source bean calls the listener method in the adapter which in turn calls an appropriate method in

the target bean. Methods in the target bean are discovered by the builder tools and exposed during bean connection.

You may have to design two types of methods: firing and receiving event methods. When a bean acts as an event source, you will have to provide event registration and firing methods. If the event is built-in and is fired by a component within the bean, then the event firing task can be delegated to that component. In that case, you need to define the event registration methods and not the event firing method (for example, a push button within a bean).

When a bean is a target, you may provide methods to receive events if you know the event that bean expects to receive. In the previous chapter we saw an example of methods that receive events. The `receivePulse(TimerEvent)` method in the Stopwatch bean and the `receiveMessage(MessageEvent)` method in the Message Display beans are examples of methods that receive events.

However, designing methods is not that straightforward in practice. Because of the independent nature of beans, you may not know what events other beans fire. Therefore, designing methods require a great deal of foresight.

Let's look at an example. In the Plotter beans, we developed a special event called `plot`. In order to receive this event, we designed methods such as `plotPieChart(PlotEvent pe)` and `plotBarChart(PlotEvent pe)`. However, a third-party bean may not fire a `plot` event. In order to accommodate such situations, we designed a common method `drawPlot(Object x, Object y)` for all the plotter beans. The formal parameter `x` is of type `Object` but the actual parameter can be an array of `int`, `long`, `short`, `double`, `float` and `String`. The `y` argument is similar. If your bean needs to connect to the plotter beans, you may have to provide the appropriate code to call the `drawPlot()` method in the adapter. This is quite easy in JBuilder as it provides an editor window to enter your code at connection time.

Here is a code snippet from the Pie Chart bean.

```
public void plotPieChart(PlotEvent e){
    y = e.getYValues();
    x = e.getXTextValues();
    reset();
    drawPlot(x,y);
}
```

The `drawPlot()` method is defined in the plotter beans' super class.

### 4.2.3 Designing connection strategies

As alluded to several times, connection between beans is facilitated by events. Typically, beans are developed independent of one another, so it's hard to anticipate how a bean can be connected to others.

Aside from notification of an occurrence, events can also send data from one bean to another. When designing a connection, you may need to consider the following strategies:

#### 1 Using existing events

If you only need to notify the event occurrence and not transfer of data, you can design the connection mechanism using the existing AWT /JFC events. (for example, pushing a button in one bean to trigger an action in another).

#### 2 Using bound properties

It is often necessary to notify the event occurrence as well as transfer data from the source to the target bean. If the source and the target share the same type of property as the data holder, you need not create special events for the sake of connection. You can declare such properties *bound* and use the `propertyChange` event to transfer data from the source bean to the target.

For example, the `message` property in the Stopwatch and Message Display beans is the same type. Even though we created a special `message` event to connect the Stopwatch to the Message Display bean, we didn't have to. We could easily have connected the two by binding the message property. We'll describe how to use the `propertyChange` event in the "Property change notification" section.

### 3 Using custom events

While creating a custom event involves additional coding, it is the surest way to send data from a source to a target. If you are not sure if the receiving bean has a compatible property, you would create special events for connection.

In the last chapter, we developed the timer and message events as examples of creating a special event. Creating an event to send images is another excellent example. The data visualization bean suite has the following two beans: Image Viewport and Image Loader. The Image Viewport bean displays and manipulates images. The Image Loader bean loads images.

Whenever an image is loaded by the Image Loader, it has to be sent to the Image Viewport for display. To achieve this, we created a new event called `imageLoaded`. The Image Loader bean acts as source for the `imageLoaded` event. The Image Viewport bean acts as a receiver of the `imageLoaded` event.

You can use the `imageLoaded` event to connect the Image Loader bean to the Image Viewport bean. Whenever image loading is complete, the Image Loader bean constructs the `imageLoaded` event and sends it to the registered listeners. When the Image Viewport is connected to the Image Loader, the adapter acts as a listener for the `imageLoaded` event and calls the appropriate method in the Image Viewport to send the image. The `ImageLoadedListener` code looks like this:

```
package vis.beans.imageloader;
public interface ImageLoadedListener extends java.util.EventListener{
    public void imageLoaded(ImageLoadedEvent e);
}
```

Below is the code listing for the `ImageLoadedEvent` class:

```
package vis.beans.imageloader;

public class ImageLoadedEvent extends java.util.EventObject{
    ScreenImage screenImage;
    public ImageLoadedEvent(Object obj, ScreenImage img){
        super(obj);
        screenImage = img;
    }
    public ScreenImage getScreenImage(){
        return screenImage;
    }
}
```

As you can see, the `ImageLoadedEvent` class has the `ScreenImage` object as an instance variable. The `ScreenImage` contains the `Image` object. Being a source for this event, the Image Loader bean creates the `ImageLoadedEvent` object. The `ImageCanvas` class in the Image Viewport bean can receive this event. It has the `displayImage()` method shown in the next example. As you can see, this method fetches the `ScreenImage` object from the `ImageLoadedEvent` and uses it for painting the image.

```
public void displayImage(ImageLoadedEvent e){
    ScreenImage img = (ScreenImage)e.getScreenImage();
    imagePaint(img);
}
```

### 4 Using InfoBus

InfoBus is not part of JavaBeans 1.0. At the time of this writing, an InfoBus preview has been released. InfoBus enables dynamic exchange of data. When you use bound properties or custom events for connec-

tion, you need to know the type of data being exchanged. But, when using InfoBus, you can exchange data between beans which are unaware of one another and do not need to know the data type in advance. However, to achieve dynamic data exchange, your bean implementation has to comply with certain InfoBus protocols. In other words, it has to implement certain interfaces that would make it a member of InfoBus. You can find more details in the “Advanced JavaBeans” chapter.

When a bean can receive many events, there may be several ways of connecting to it. Some connections might require special events and others might not. From a user’s point of view, fewer connections are certainly desirable.

#### 4.2.4 Class design

Once you’ve determined what properties, methods and events a bean should expose, your next step is to analyze and design object-oriented run-time classes. A detailed discussion on object-oriented design is beyond the scope of this book. However, we’ll provide a quick overview of the Plotter beans example. The design of these beans is explained in Part II of this book under “Plotting Beans.” Figure 4.1 illustrates the class hierarchy.

Let’s quickly review the class hierarchy. The root is the `Viewport` class which facilitates double buffering and painting of plots. The `PlotViewport` class, which is a subclass of `Viewport`, is the root of the plotting hierarchy. There are five beans which are divided into two categories: scientific and business plots. The XY Plot and Histogram beans belong to the scientific category and the Line Chart, Bar Chart, and Pie Chart beans belong to the business category. The `XYPlot` and `Histogram` classes are direct descendants of the `PlotViewport` class. The business beans, however, are subclasses of the `Chart` class which extends the `PlotViewport` class. The `PlotViewport` class has several common properties and methods.

This example illustrates an important point: many beans can share the same class hierarchy. Once we develop the hierarchy, the next step is to build the actual classes. When a bean is part of a hierarchy of classes, it inherits all the properties of its super classes. If beans have common properties, they can move up in the class hierarchy. In the prior example, properties such as `plot title`, `X label string`, and `Y label string` are defined in the root of the plot hierarchy (`PlotViewport` class). The target bean classes define only those properties specific to the bean.

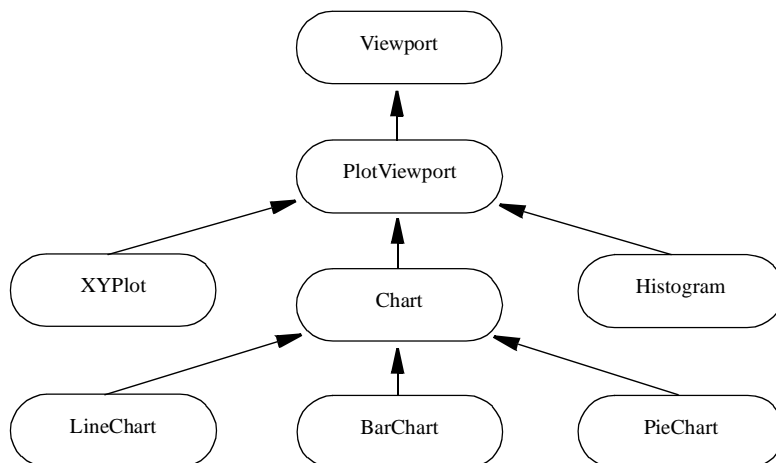


Figure 4.1 The Plotter bean class hierarchy

## 4.2.5 Package design

The JavaBeans specification does not define the relationship between packages and run-time bean classes. In theory, bean classes can even belong to the default package, that is, have no package defined for them. But in practice, bean-testing tools may not accept such beans. Sun's Beanbox certainly doesn't, so it's always advisable to put your bean in a package.

Typically, a bean goes into a single package. But, there is nothing to prevent you from putting many beans in one package. For example, all the plotter beans are in one package (`vis.beans.plotter`). There is also nothing to prevent you from spreading your run-time classes over many packages. Several beans can share common classes that are in entirely different package hierarchies. For example, the plotter beans use a number of utility classes from the `vis.util` package. The bean vendor is responsible for installing any packages the bean uses.

Since beans are typically distributed, you may want to follow Java's package naming conventions for your beans package. If you use this convention, all packages must start with `com.<your company name>`.

Some guidelines for designing your package follow:

- 1 Put all the bean-specific classes in one package.
- 2 If you are designing a bean suite (a number of related beans), create a package hierarchy. In our visualization bean suite, for instance, the root of the bean package hierarchy starts with `vis`. All the bean-related packages are in a subpackage called `beans`.
- 3 Put common classes in a separate package.  
In our example, we put all the common classes in the `vis.util` package. There are no bean-target classes in this package.
- 4 If you've used special events, put the event related classes in a separate package. In the visualization bean suite, custom event classes and interfaces are in the package `vis.beans.events`. Grouping events in this manner saves you the trouble of keeping track of where the events are defined. The same argument holds true for property editors.

*Note: For the sake of completion, we included the design-time classes. If you are not familiar with them, refer to this section once you have completed the designtime chapters.*

- 1 If your bean is associated with `BeanInfo` classes, they should go in the same package as the bean.
- 2 If you've developed custom property editors used by many beans, put them in a separate package. In our case, all the property editors are in the package `vis.beans.editors`.
- 3 Unlike property editors, customizers are specific to a bean, so they should typically go in the same package as the bean. However, several customizers can share the same class. You can put them in a separate package. In our case, we put some customizer classes in the `vis.beans.customizers` package.

## 4.3 Implementing run-time classes

Implementing run-time bean classes is more involved than implementing class libraries. There are several requirements you need to meet to build run-time bean classes. This includes providing a no-argument constructor, conforming to the design patterns, and making a bean class serializable. In addition to the code that implements the bean functionality, a typical run-time bean class will have the following:

- 1 A constructor with no arguments
- 2 Setter and getter methods for properties
- 3 Methods that register and fire events

- 4 Methods that receive events
- 5 Serialization code

We discussed some of these items in the previous chapter.

### 4.3.1 *Constructor with no arguments*

It's essential that bean-target classes have no-argument constructors. This is because a bean is expected to be instantiated by application builder tools. If constructors with arguments are allowed, the application builder tools have to provide arguments. If there are multiple constructors, they have to choose one. Since builder tools have no way of knowing which constructor to use or what its arguments are, you need the no-argument constructor. Once you define the no-argument constructor, nothing prevents you from providing as many constructors as you like.

Beans often need initialization parameters. How do you provide the initial values to such parameters? You can either hard code the default values or expose them as properties. If the initialization parameters are properties, you can set their values using the property editor or customizer. In the case of beans that are also applets, there is one more option: you can pass the initial parameters through the applet's HTML file.

### 4.3.2 *Conformance to design patterns*

As mentioned in the previous chapter, builder tools automatically deduce information about a bean's properties, methods, and events using design patterns in the bean-target class. So, when you develop your bean-target class, you need to conform to the design patterns.

The need often arises to use legacy classes as beans. If you don't want to change the names of the methods to conform to design patterns, you can still use them as beans by implementing certain descriptor methods in the `BeanInfo` class. You'll learn about them in the "Building design-time classes" chapter. If you are creating new bean classes, conforming to the design patterns saves you additional coding.

### 4.3.3 *Implementing multiple bean classes*

As we mentioned before, a bean is an object which is instantiated from a class.<sup>1</sup> This does not mean that a bean is comprised of a single run-time class. On the contrary, a typical bean may consist of several classes including superclasses. A bean class may delegate some of its tasks to other class(es).<sup>2</sup> These classes are also part of the bean. While the bean-target, its superclasses and bean-helper classes make up a bean, they need not be in the same jar file or in the same package as the bean-target class.

Because bean-target classes can be part of a hierarchy, they can inherit properties from their superclasses. For example, when a bean-target class is subclassed from the AWT Component class, properties like `name`, `font`, `background`, and `foreground` are also inherited. You need not reimplement them.

With the current JavaBeans model, properties have to be defined in the bean class hierarchy. They cannot be in the bean-helper classes. Often, it is unavoidable. In such cases, you need to define the properties in the bean-target class and call the property set and get methods in the bean-helper class as a workaround. An example follows:

`ImageCanvas` is the bean-target class of the Image Viewport bean. It delegates image loading tasks to the `ImageLoader` class, which maintains a list of images needed for the movie feature and controls the rate at which the images are displayed. Because this rate is a property, it has to be defined in the `ImageCanvas` class.

---

1. We called such a class a *beantarget* class  
2. We called such a class a *beanhelper* class

In the `ImageCanvas` class:

```
private ImageLoader loader;
protected double cineRate;

public void setCineRate(int rate){
    if(rate ==0) return ;
    if(loader != null) loader.setCineRate(rate);
    cineRate = rate;
}

public int getCineRate(){
    return cineRate;
}
```

The `ImageLoader` class has instance variables named `frameRate` and `delay`:

```
public void setCineRate(int rate){
    if(rate==0) return;
    frameRate = rate;
    delay = (int)((60.0/(double)frameRate)*1000);
    if(delay <10) delay =10;
}
```

Ideally, we should define the `cineRate` property in the `ImageLoader` class. But, until JavaBeans supports multiple collaborating objects, we may have to define all such indirect properties in the bean-target class.

The same logic applies to events and methods. A beantarget class has to hold all the event and method design patterns in order for the builder tool to expose them.

#### 4.3.4 *Implementing serialization*

It is very important that you make your run-time classes serializable. Many complications arise when you implement serialization. We dealt with some of them in previous chapters.

Let's go over an example that deals with images. In the `ImageViewport` bean, the image displayed on the viewport has to be saved and restored. Since images cannot be serialized, we used an alternate mechanism. In the `Image Viewport` case, we save the ID of the image that is displayed on the viewport. Upon deserialization, the images are loaded from the jar again and the saved ID is used to display the image on the viewport.

The following listing shows the serialization related code from the `ImageCanvas` class:

```
public class ImageCanvas extends Canvas implements Callback, Serializable {

    public ImageLoader loader=null; // helps in image loading
    private boolean loaded= false;
    private int loadedFrom = JAR;
    private int curImageId; //this is set when a new image displayed
    // Other variables
    ...

    //Other methods
    ...
    private void writeObject(ObjectOutputStream out)throws IOException{
        //Set loaded to be false; need to load the images again
        loaded = false;
        //Save all the nontransient, nonstatic fields
        out.defaultWriteObject();
    }
}
```

```

private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException{
    //Read all the default fields
    in.defaultReadObject();
    //Now load the images;
    if((loadedFrom == JAR) && (loader != null)){
        loader.loadImages();
        loaded = true;
    }
}

// Other methods
...
public void addNotify(){
    super.addNotify();
    if(loader == null)loader = new ImageLoader(this);
    if(Beans.isDesignTime()){
        //Load all the images from the jar at designtime only
        if(!loaded){
            loader.loadImages();
            loaded = true;
        }
        loader.loadCurrentImage(curImageId);
    }
}
}

```

The `writeObject()` method saves the non-transient and non-static fields, including `curImageId`, to the output stream. Notice that the `writeObject()` method sets the `loaded` variable to `false`. This allows the images to load again after deserialization. The `readObject()` method restores the values of the saved fields, including the value of `curImageId`.

The `addNotify()` method is invoked only after the bean is visible. This method is called whether an object is instantiated from the serialized prototype or class file. When invoked, `addNotify()` displays an image whose ID is held in `curImageId`.

### 4.3.5 *Implementing property change events*

A property can be bound and constrained. When a bound property changes, the properties in other beans tied to it also change. Similarly, when a constrained property changes, the other beans in the container may apply appropriate constraints and veto the change.

*Note: To refresh your memory, you may want to refer to the [Bound Properties section in the JavaBeans Model chapter](#) before proceeding.*

**Property change events** How does a bean tell other beans that it has bound and constrained properties? Through events. There are two types of events: `propertyChange` and `vetoableChange`. The classes and interfaces for these events are in the `java.beans` package.

A bean fires a `propertyChange` event whenever its bound property changes and a `vetoableChange` event whenever a constrained property changes. Each of these events has a corresponding listener interface. The `PropertyChangeListener` interface listens to bound properties and the `VetoableChangeListener` listens to constrained properties. Both the `propertyChange` and `vetoableChange` events are supported by an event state class called `PropertyChangeEvent` which passes the property related data from the source bean to the target.

The `PropertyChangeEvent` class constructor is as follows:

```
public PropertyChangeEvent(Object source,
                          String propertyName,
                          Object oldValue,
                          Object newValue)
```

The parameters `oldValue` and `newValue` refer to the old and new values of the property. As you can see, they are of the `Object` type. Therefore, before passing a parameter, primitive properties have to be converted to objects using the wrapper classes.

The methods in the `PropertyChangeEvent` class, `getPropertyName()`, `getOldValue()`, and `getNewValue()`, return the property's name, old value, and new value. In addition, the `PropertyChangeEvent` class has two more methods, `setPropagationId()` and `getPropagationId()`, which have no use in JavaBeans 1.0.

The `PropertyChangeListener` interface has just one method called `propertyChange(PropertyChangeEvent e)` which is invoked when a bound property is changed. The `VetoableChangeListener` interface also has a method named `vetoableChange(PropertyChangeEvent e)` which is called when a constrained property changes. It throws an exception called `PropertyVetoException`. The bean that receives the `VetoableChangeEvent` can throw an exception if it does not want the property to change.

Implementing a property change notification involves a lot of routine code including methods to add and remove listeners and firing events. The `java.beans` package has two utility classes called `PropertyChangeSupport` and `VetoableChangeSupport` which support event registration and firing.

### 4.3.6 *Implementing a bound property*

Whenever a bound property changes, the `propertyChange` event has to fire to the registered listeners. An appropriate place to fire a property change event is in the property's `setter` method. A bean that owns the bound properties has to implement the `<add/remove>PropertyChangeListener` method for registering the `propertyChange` event listeners. Further, it should have a method to fire the `propertyChange` event. Alternatively, a bean can utilize the helper `PropertyChangeSupport` class and delegate these event related tasks.

The `PropertyChangeSupport` constructor has one argument which is of the `Object` type. The bean that needs to implement the bound property typically passes itself as an argument to this constructor. The `PropertyChangeSupport` method has the following three methods:

```
1 addPropertyChangeListener(PropertyChangeListener pcl)
2 removePropertyChangeListener(PropertyChangeListener pcl)
3 firePropertyChange(String propertyName, Object oldValue, Object newValue)
```

When a bound property changes, this method fires the property change notification to all the registered listeners by invoking `propertyChange()`. The property related data is sent to the listeners through the `PropertyChangeEvent` object which is passed as an argument to the `propertyChange()` method. The `propertyChange()` event does not fire when the `newValue` is the same as the `oldValue`.

Since the `PropertyChangeSupport` class simplifies the code, we'll only use this class in our examples for event related tasks.

The steps to implement a bound property using the `PropertyChangeSupport` class appear in the next example.

- 1 In the source bean (that is, the bean that has the bound property), create a `PropertyChangeSupport` object. Typically, this can be done in the bean constructor or in the `init()` method:

```
PropertyChangeSupport pcNotifier= new PropertyChangeSupport(this);
```

- 2 Add the `propertyChange` event registration methods:

```

public void addPropertyChangeListener(PropertyChangeListener p){
    pcNotifier.addPropertyChangeListener(p);
}
public void removePropertyChangeListener(PropertyChangeListener p){
    pcNotifier.removePropertyChangeListener(p);
}

```

- 3 In the property's setter method, call the `firePropertyChange()` method of the `PropertyChangeSupport` class as shown below:

```

public void set<property>(aType newValue){
    aType oldValue = property;
    pcNotifier.firePropertyChange(propertyName, oldValue, newValue);
    property = newValue;
    // Other statements
}

```

The preceding steps are implemented in the source bean.

### 4.3.7 *Listening to bound properties*

The builder tool typically exposes the list of bound properties in the selected bean. You select the desired property from the list. Once you select a property, the builder tool lets you target a bean in which matching properties are found. Once you select the target bean, the builder tool displays the matching properties in the target. The properties are typically matched by the argument types.

When you select a matching property, the builder tool creates a hook-up class with an event adapter class for the `propertyChange` event. The adapter class implements the `PropertyChangeListener` interface. In other words, this class will implement `propertyChange()` which will invoke the setter method for the matching property in the target bean.

The builder tool creates a property change adapter object from this class and registers it with the source bean for `propertyChange` event notification. When the bound property changes, `propertyChange()` is invoked in the adapter. The `propertyChange()` method extracts the new value of the property from its argument, which is the `PropertyChangeEvent` state object. It then calls the property's setter method in the target bean to set it to this value.

How does the builder tool recognize that a property is bound? By using design patterns.

The property change notification is a design-time and run-time operation. The property change events are always fired no matter the bean mode.

### 4.3.8 *Using bound properties for connecting beans*

As we mentioned before, the `propertyChange` event can be used to connect beans. This often eliminates the need to create special events. Let's examine this with an example.

The Image Viewport bean allows images to be displayed and manipulated and has a number of related properties. Let's consider the magnification properties, `magOn` and `magFactor`. The `magOn` property turns magnification on or off. The `magFactor` property, which is of the `double` type, holds the magnification factor.

A client bean can control to the magnification of the image in the Image Viewport bean by invoking the setter methods in `magOn` and `magFactor`. One approach would be use AWT events. As an example, a check box in the client can be tied to the `magOn` property and a scrollbar can be tied to the `magFactor` property. The problem with this approach is that the Image Viewport should have methods to receive different types of AWT events since client bean can have any type of GUI.

The other approach would be to bind properties. The target bean can also have similar properties which can be bound to the Image Viewport bean at connection time. You can see the Image Viewport and Pan Zoom

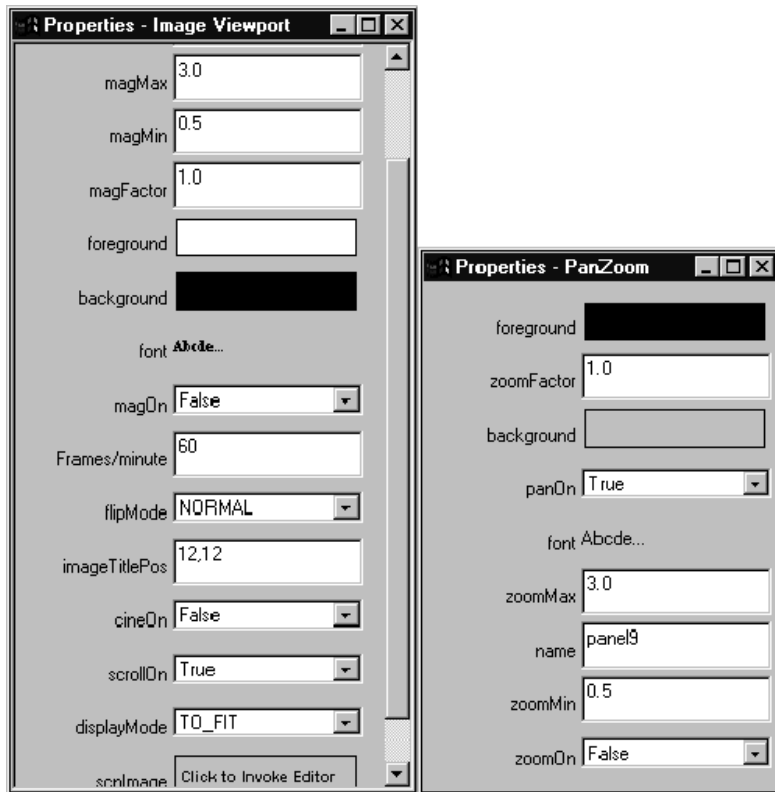


Figure 4.2 Image Viewport and Pan Zoom bean property sheets

beans in figure 4.3. The Pan Zoom bean has the GUI that controls the panning and zooming in the Image Viewport bean.

The screen shot in figure 4.2 shows the property sheets for the Image Viewport and Pan Zoom beans.

Next, we'll try to bind the properties of the Pan Zoom bean to the Image Viewport bean. The `zoomOn` property in the Pan Zoom bean should match `magOn` in the ImageViewport. Similarly, the `zoomFactor` property should match the `magFactor` property.

The Bind Property menu in the BeanBox shown in figure 4.3 enables binding of properties. This menu is only available when the active bean has bound properties. When the Bind Property menu is clicked, a `PropertyDialog` with a list of bound properties appears.

As you can see, the Pan Zoom bean has a number of bound properties. Let's bind the `zoomOn` property. When you click `zoomOn`, the BeanBox lets the you point a red line to a target bean. When you point that red line to the ImageViewport bean and click, the matching properties in the Image Viewport bean appear in a dialog box as shown in the left-hand screenshot in figure 4.4. As you can see, there are three matching properties

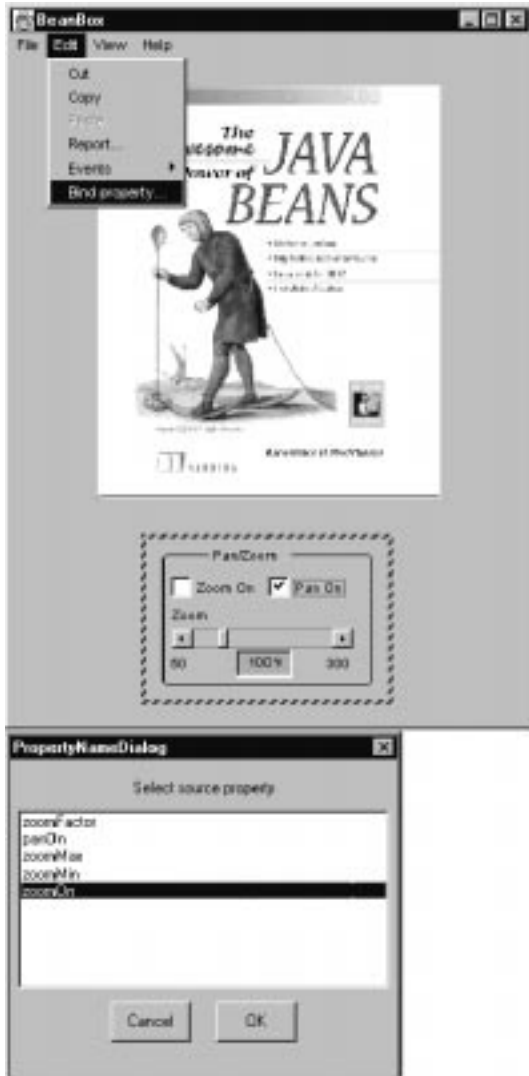


Figure 4.3 Connecting the Pan Zoom bean to the Image Viewport bean by binding properties

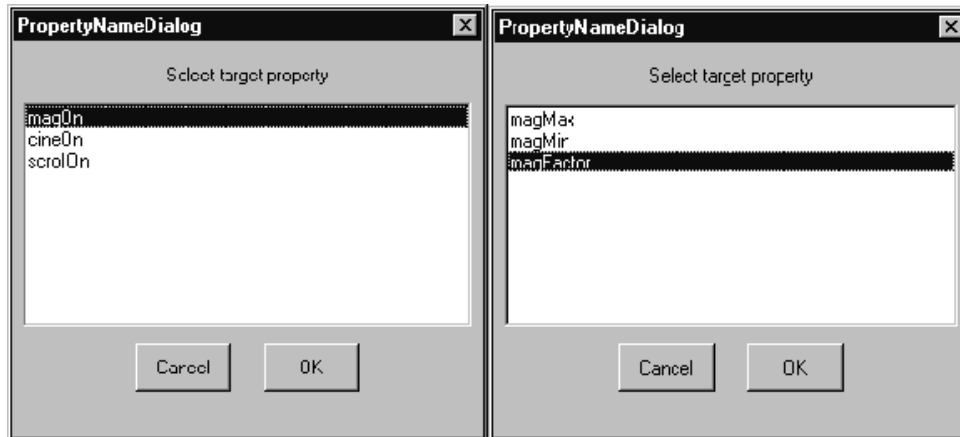


Figure 4.4 Matching properties in the Image Viewport bean.

which have Boolean arguments. When you click `magOn`, the `zoomOn` property in the Pan Zoom bean is bound to the `magOn` property in the Image Viewport.

In the same manner, we can bind the `zoomFactor` property in the Pan Zoom bean to the `magFactor` property. Figure 4.4 shows a list of matching properties in the Image Viewport bean.

Having bound the `zoomOn` to `magOn` and `zoomFactor` to `magFactor`, you can now turn on the magnification by setting the Zoom On check box and increment/decrement the zoom factor by sliding the scroll bar. You can see the image being magnified/scaled as the slider moves.

*Implementation* Next, we'll see how the `zoomOn` and `zoomFactor` properties' setter methods are implemented in the `PanZoom` class. Note that the `pcNotifier` is an object instantiated from the `PropertyChangeSupport` class. Refer to the preceding steps.

```
public void setZoomOn(boolean onOrOff){
    pcNotifier.firePropertyChange("zoomOn",
        new Boolean(zoomOn),
        new Boolean(onOrOff));

    zoomOn = onOrOff;
}
public boolean getZoomOn(){
    return zoomOn;
}

public void setZoomFactor(double zoomfac){
    pcNotifier.firePropertyChange("zoomFactor",
        new Double(zoomFactor),
        new Double(zoomfac));

    zoomFactor = zoomfac;
}
public double getZoomFactor(){
    return zoomFactor;
}
```

### 4.3.9 Implementing a constrained property

Implementing a constrained property is a little more complicated than implementing a bound property. Whenever a constrained property changes, the `vetoableChange` event has to be fired to the registered listeners. Typ-

ically, a constrained property is also a bound property. An appropriate place to fire the `vetoableChange` event is in the property's `setter` method.

A direct way to implement a constrained property is to implement the `VetoableChangeListener` interface. However, just like the `PropertyChangeSupport` class, you can use the helper class `VetoableChangeSupport` for `vetoableChange` event related tasks.

The constructor for the `VetoableChangeSupport` class has one argument which is of the `Object` type. A bean that needs to implement the `vetoableChange` property typically passes itself as an argument to this constructor. `VetoableChangeSupport` has the following three methods:

```
1 addVetoableChangeListener(VetoableChangeListener pcl)
2 removeVetoableChangeListener(VetoableChangeListener pcl)
3 public void fireVetoableChange(String propertyName,
                                Object oldValue,
                                Object newValue)
   throws PropertyVetoException
```

As you can see, this method throws the `PropertyVetoException`. When a constrained property changes, this method fires the property change notification to all the registered listeners by invoking `vetoableChange()` in the listeners. The listeners receive the property related data through the `PropertyChangeEvent` object which is passed as an argument to `vetoableChange()`.

Typically, target beans receive the new property value through a `vetoableChange` event adapter. A target bean can choose to veto the change if the new property value does not meet its criteria. This is accomplished by throwing a `PropertyVetoException`.

If the source bean catches such an exception, it won't modify the property. The situation gets more complicated when there is more than one listener. When one bean vetoes a property change, there is no way for the other beans to know about it. To solve this problem, the JavaBeans specification requires that a second `vetoableChange` event be fired with the original value. To comply with this specification, when the veto is received, `fireVetoableChange()` constructs a new `PropertyChangeEvent` by interchanging the old and new values. It then invokes the `vetoableChange()` methods in all the registered listeners.

The steps for implementing a constrained property using the `VetoableChangeSupport` class are given below. Here, both the source and the target beans are involved.

### *Source bean*

- 1 In the source bean (the bean with the constrained property), create a `VetoableChangeSupport` object. Typically, this can be done in the bean constructor or `init` methods:

```
VetoableChangeSupport vetoNotifier= new VetoableChangeSupport(this);
```

If the constrained property is also bound, which is typically the case, create the `PropertyChangeObject`:

```
PropertyChangeSupport pcNotifier= new PropertyChangeSupport(this);
```

- 2 Add the `vetoableChange` event registration methods:

```
public void addVetoableChangeListener(VetoableChangeListener p){
    vetoNotifier.addVetoableChangeListener(p);
}
public void removeVetoableChangeListener(VetoableChangeListener p){
    vetoNotifier.removeVetoableChangeListener(p);
}
```

- 3 In the constrained property's `setter` method, call the `fireVetoableChange()` method:

```
public void set<Property>(aType newValue){
```

```

aType oldValue = property;
try {
    vetoNotifier.fireVetoableChange(propertyName, oldValue, newValue);
} catch (PropertyVetoException e) {
    // Handle the property veto exception here itself
    System.out.println(e.getMessage());
    return;
}
property = newValue;
// Other statements
}

```

In the preceding code, the `PropertyVetoException` is handled in the setter method itself. The setter method can propagate the exception to its calling method, which is typically a method from the builder tool. When it catches this exception, the builder tool may issue a warning in a dialog box. The same setter method now changes to:

```

public void set<Property>(aType newValue) throw PropertyVetoException{
    aType oldValue = property;
    vetoNotifier.fireVetoableChange(propertyName, oldValue, newValue);
    property = newValue;
    // Other statements
}

```

- 4 If the constrained property is also bound, add the `propertyChange` event notification code after the `fireVetoableChange()` method as shown:

```

public void set<Property>(aType newValue) throw PropertyVetoException{
    aType oldValue = property;
    vetoNotifier.fireVetoableChange(propertyName, oldValue, newValue);
    //Bound property notification
    pcNotifier.firePropertyChange(propertyName, oldVaue, newValue);
    property = newValue;
    // Other statements
}

```

When the `PropertyVetoException` is handled locally, add this code after the catch block.

### *Target bean*

- 1 Create a method that has an argument that is of the `PropertyChangeEvent` type and throws `PropertyVetoException`.
- 2 Retrieve the new value of the property from the `PropertyChangeEvent` and apply the constraints to the new value. If the property does not satisfy the constraints, throw the `PropertyVetoException`.

Here is the pseudocode:

```

public void vetoSomePropertyChange(PropertyChangeEvent e)
    throws PropertyVetoException {
    Property type newValue = (PropertyType)e.getNewValue();
    if (constraintsFailed) // apply the constraints to newValue
        throw new PropertyVetoException("Constraints failed", e);
}
}

```

For every constrained property a bean needs to veto, there has to be a method as previously shown. There can be many beans that can veto a constrained property.

### *4.3.10 Listening to constrained properties*

We have seen the source and target constrained property implementation steps. But, how does a source bean hook up its constrained properties with other interested beans? Through the `vetoableChange` event. The pro-

cedure is the same as the one used for connecting beans through events. The source bean exposes the `vetoableChange` event. When it is linked to a target bean method that can receive the `vetoableChange` event, the builder tool generates a hook-up class with a `vetoableChange` event adapter and registers that adapter with the source bean. The `propertyChange()` method in the adapter invokes the connected target bean method. If the target bean method raises the `PropertyVetoException`, it is propagated all the way to the source bean.

Bound properties are relatively easy to connect because they are connected on a one-to-one basis. On the other hand, constrained properties are connected through events. On the source bean side, there may be several properties firing `vetoableChange` events. On the target bean side, there may be several veto methods (methods with the `PropertyChangeEvent` argument). The problem is how to make sure a given constrained property is connected to the appropriate veto method.

Let's take the same Pan Zoom bean example discussed in the bound properties section. As we saw, a scrollbar controls the `zoomFactor`. This property has a minimum and a maximum value and corresponds to the `zoomMin` and `zoomMax` properties. You can set this property to anything. Does that mean that when `zoomFactor` is bound to `magFactor`, the Image Viewport will magnify the image by any amount? No, the Image Viewport has a limit by which a image can be magnified/scaled. This limit depends on local machine capabilities like availability of primary memory. In order to enforce such constraints, the Image Viewport bean has two properties, `magMin` and `magMax`, which hold the minimum and maximum `magFactor` values.

Here are the setter methods for `zoomMax` in the PanZoom bean:

```
public void setZoomMax(double max){
    try{
        vetoNotifier.fireVetoableChange("zoomMax",
                                       new Double(zoomMax),
                                       new Double(max));
    } catch(PropertyVetoException e){
        System.out.println("Zoom max vetoed: "+ e.getMessage());
        return;
    }

    pcNotifier.firePropertyChange("zoomMax",
                                   new Double(zoomMax),
                                   new Double(max));

    zoomMax = max;
    maxLabel.setText(Integer.toString((int)(zoomMax*100)));
    zoomSlider.setMaximum((int)(zoomMax*100));
}
```

Since `zoomMax` is also a bound property, you can see the `propertyChange` event notifier code. If no veto is received, `zoomMax` is updated and all the registered `propertyChange` listener beans are notified. Once the property is updated, the new value has to be reflected on the GUI. The `zoomSlider`, which is a scrollbar, controls the `zoomFactor`. The minimum and maximum values of `zoomSlider` are percentages. The maximum value of the `zoomSlider` is accordingly set to the new percentage. The `maxLabel` displays the maximum zoom value, which is also set to the new value.

The corresponding target bean code is taken from the `ImageCanvas` class.

```
public void vetoMagMaxChange(PropertyChangeEvent pce)
    throws PropertyVetoException{
    double maxmag =((Double)(pce.getNewValue())).doubleValue();
    if ((maxmag > magMax ) || (maxmag <= magMin)){
        throw new PropertyVetoException("Max value out of range", pce);
    }
}
```

Similar code exists for `zoomMin` in both the Pan Zoom and Image Viewport beans.

What we've seen so far are simple cases. Things get complicated when there are many bound and constrained properties. It is hard to anticipate which properties are going to be constrained. Since beans can be used in a wide variety of applications, it is often not easy to arrive at commonly acceptable constraints.

## *4.4 Summary*

We discussed the difference between Java Beans and class libraries. Beans and class libraries don't compete but rather complement one another. A major difference is how you use them. An end user who builds applications with beans doesn't need to be proficient in Java. On the other hand, Java programming skills are a must for using class libraries.

Designing beans requires a lot of thought and foresight. We discussed the issues involved in designing beans for realworld applications. It was followed by some implementation details. We went over the implementation of bound and constrained properties in great detail and noted that bound properties can connect beans.