



## CHAPTER 6

# *JavaBeans*

The JavaBeans API provides a framework for defining reusable, embeddable, modular software components. The JavaBeans specification includes the following definition of a bean: “a reusable software component that can be manipulated visually in a builder tool.” As you can see, this is a rather loose definition; beans can take a variety of forms. The most common use of beans is for graphical user interface components, such as components of the `java.awt` and `javax.swing` packages, which are documented in *Java Foundation Classes in a Nutshell* (O’Reilly).<sup>\*</sup> Although all beans can be manipulated visually, this does not mean every bean has its own visual representation. For example, the `javax.sql.RowSet` class (documented in *Java Enterprise in a Nutshell* (O’Reilly)) is a JavaBeans component that represents the data resulting from a database query. There are no limits on the simplicity or complexity of a JavaBeans component. The simplest beans are typically basic graphical interface components, such as a `java.awt.Button` object. But even complex systems, such as an embeddable spreadsheet application, can function as individual beans.

One of the goals of the JavaBeans model is interoperability with similar component frameworks. So, for example, a native Windows program can, with an appropriate bridge or wrapper object, use a JavaBeans component as if it were a COM or ActiveX component. The details of this sort of interoperability are beyond the scope of this chapter, however.

The JavaBeans component model consists of the `java.beans` and `java.beans.beancontext` packages and a number of important naming and API conventions to which conforming beans and bean-manipulation tools must adhere. Because JavaBeans is a framework for generic components, the JavaBeans conventions are, in many ways, more important than the actual API.

---

<sup>\*</sup> JavaBeans are documented in this book instead of that one because the JavaBeans component model is not specific to AWT or Swing programming. Nevertheless, it is hardly possible to discuss beans without mentioning AWT and Swing components. You will probably get the most out of this chapter if you have some familiarity with GUI programming in Java using AWT or Swing.

Beans can be used at three levels by three different categories of programmers:

- If you are writing an application that uses beans developed by other programmers or using a beanbox tool\* to combine those beans into an application, you need to be familiar with general JavaBeans concepts and terminology. You also need to read the documentation for the individual beans you use in your application, but you do not need to understand the JavaBeans API. This chapter begins with an overview of JavaBeans concepts that should be sufficient for programmers using beans at this level.
- If you are writing beans, you need to understand and follow various JavaBeans naming and packaging conventions. After the introduction to general bean concepts and terminology, this chapter describes the basic bean conventions bean developers must follow. Although a JavaBeans component can be implemented without using the JavaBeans API, most beans are distributed with various auxiliary classes that make them easier to use within beanbox tools. These auxiliary classes rely heavily on the JavaBeans API so that they can interoperate with beanbox tools.
- If you are developing a GUI editor, application builder, or other beanbox tool, you use the JavaBeans API to help you manipulate beans within the tool. You also need to be intimately familiar with all the various JavaBeans programming conventions. Although this chapter describes the most important conventions, you should also refer to the primary source, the JavaBeans specification (see <http://java.sun.com/beans/>).

## *Bean Basics*

Any object that conforms to certain basic rules can be a bean; there is no Bean class all beans are required to subclass. Many beans are AWT components, but it is also quite possible, and often useful, to write “invisible” beans that do not have an onscreen appearance. (Just because a bean does not have an onscreen appearance in a finished application does not mean it cannot be visually manipulated by a beanbox tool, however.)

A bean is characterized by the properties, events, and methods it exports. It is these properties, events, and methods an application designer manipulates in a beanbox tool. A *property* is a piece of the bean’s internal state that can be programmatically set and/or queried, usually through a standard pair of get and set accessor methods.

A bean communicates with the application in which it is embedded and with other beans by generating *events*. The JavaBeans API uses the same event model AWT and Swing components use. This model is based on the `java.util.EventObject` class and the `java.util.EventListener` interface; it is described in detail in *Java Foundation Classes in a Nutshell* (O’Reilly). In brief, the event model works like this:

---

\* *beanbox* is the name of the sample bean-manipulation program provided by Sun in its Beans Development Kit (BDK). The term is a useful one, and I’ll use it to describe any kind of graphical design tool or application builder that manipulates beans.

- A bean defines an event if it provides add and remove methods for registering and deregistering listener objects for that event.
- An application that wants to be notified when an event of that type occurs uses these methods to register an event listener object of the appropriate type.
- When the event occurs, the bean notifies all registered listeners by passing an event object that describes the event to a method defined by the event listener interface.

A *unicast event* is a rare kind of event for which there can be only a single registered listener object. The add registration method for a unicast event throws a `TooManyListenersException` if an attempt is made to register more than a single listener.

The *methods* exported by a bean are simply any public methods defined by the bean, excluding those methods that get and set property values and register and remove event listeners.

In addition to the regular sort of properties described earlier, the JavaBeans API also supports several specialized property subtypes. An *indexed property* is a property that has an array value, as well as getter and setter methods that access both individual elements of the array and the entire array. A *bound property* is one that sends a `PropertyChangeEvent` to any interested `PropertyChangeListener` objects whenever the value of the property changes. A *constrained property* is one that can have any changes vetoed by any interested listener. When the value of a constrained property of a bean changes, the bean must send out a `PropertyChangeEvent` to the list of interested `VetoableChangeListener` objects. If any of these objects throws a `PropertyVetoException`, the property value is not changed, and the `PropertyVetoException` is propagated back to the property setter method.

Because Java allows dynamic loading of classes, beanbox programs can load arbitrary beans. The beanbox tool uses a process called *introspection* to determine the properties, events, and methods exported by a bean. The introspection mechanism is implemented by the `java.beans.Introspector` class; it relies on both the `java.lang.reflect` reflection mechanism and a number of JavaBeans naming conventions. `Introspector` can determine the list of properties supported by a bean, for example, by scanning the class for methods that have the right names and signatures to be get and set property accessor methods.

The introspection mechanism does not rely on the reflection capabilities of Java alone, however. Any bean can define an auxiliary `BeanInfo` class that provides additional information about the bean and its properties, events, and methods. The `Introspector` automatically attempts to locate and load the `BeanInfo` class of a bean.

The `BeanInfo` class provides additional information about the bean primarily in the form of `FeatureDescriptor` objects, each one describing a single feature of the bean. Each `FeatureDescriptor` provides a name and brief description of the feature it documents. The beanbox tool can display the name and description to the user, making the bean essentially self-documenting and easier to use. Specific bean features, such as properties, events, and methods, are described

by specific subclasses of `FeatureDescriptor`, such as `PropertyDescriptor`, `EventSetDescriptor`, and `MethodDescriptor`.

One of the primary tasks of a beanbox application is to allow the user to customize a bean by setting property values. A beanbox defines *property editors* for commonly used property types, such as numbers, strings, fonts, and colors. If a bean has a property of a more complicated type, however, it can define a `PropertyDescriptor` class that enables the beanbox to let the user set values for that property.

In addition, a complex bean may not be satisfied with the property-by-property customization mechanism provided by most beanbox tools. Such a bean can define a `Customizer` class to create a graphical interface that allows the user to configure a bean in some useful way. A particularly complex bean can even define customizers that serve as “wizards” that guide the user step by step through the customization process.

A *bean context* is a logical container (and often a visual container) for JavaBeans and, optionally, for other nested bean contexts. In practice, most JavaBeans are AWT or Swing components or containers. Beanbox tools recognize this and allow component beans to be nested within container beans. A bean context is a kind of heavyweight container that formalizes this nesting relationship. More importantly, however, a bean context can provide a set of services (e.g., printing services, debugging services, database connection services) to the beans it contains. Beans that are aware of their context can be written to query the context and take advantage of the services that are available. Bean contexts are implemented using the `java.beans.beancontext` API, which is new as of Java 1.2 and discussed in more detail later in this chapter.

Java 1.4 introduces a JavaBeans Persistence API, which allows a bean, or a tree of beans, to have their state written persistently to an XML file from which the bean or beans can later be restored. This is done with the `XMLEncoder` and `XMLDecoder` classes in `java.beans`. JavaBeans persistence is similar to the serialization mechanism of the `java.io` package, but is based entirely on the public API of the beans, and is thus more robust in the presence of multiple versions or multiple implementations of that public API. You’ll see examples of this new JavaBeans Persistence API along with examples of the Serialization API in Chapter 4. And you’ll find details about `XMLEncoder`, `XMLDecoder`, and related classes in Chapter 9.

## *JavaBeans Conventions*

The JavaBeans component model relies on a number of rules and conventions bean developers must follow. These conventions are not part of the JavaBeans API itself, but in many ways, they are more important to bean developers than the API itself. The conventions are sometimes referred to as *design patterns*; they specify such things as method names and signatures for property accessor methods defined by a bean.

The reason for these design patterns is interoperability between beans and the beanbox programs that manipulate them. As we’ve seen, beanbox programs may

rely on introspection to determine the list of properties, events, and methods a bean supports. In order for this to work, bean developers must use method names the beanbox can recognize. The JavaBeans framework facilitates this process by establishing naming conventions. One such convention, for example, is that the getter and setter accessor methods for a property should begin with get and set.

Not all the patterns are absolute requirements. If a bean has property accessor methods that do not follow the naming conventions, it is possible to use a `PropertyDescriptor` object (specified in a `BeanInfo` class) to indicate the accessor methods for the property. Although the `BeanInfo` class provides an alternative to the property-accessor-method naming convention, the property accessor method must still follow the conventions that specify the number and type of its parameters and its return value.

## ***Beans***

A bean itself must adhere to the following conventions:

### *Class name*

There are no restrictions on the class name of a bean.

### *Superclass*

A bean can extend any other class. Beans are often AWT or Swing components, but there are no restrictions.

### *Instantiation*

A bean must provide a no-parameter constructor or a file that contains a serialized instance the beanbox can deserialize for use as a prototype bean, so a beanbox can instantiate the bean. The file that contains the bean should have the same name as the bean, with an extension of `.ser`.

### *Bean name*

The name of a bean is the name of the class that implements it or the name of the file that holds the serialized instance of the bean (with the `.ser` extension removed and directory separator (`/`) characters converted to dot (`.`) characters).

## ***Properties***

A bean defines a property *p* of type *T* if it has accessor methods that follow these patterns (if *T* is boolean, a special form of getter method is allowed):

### *Getter*

```
public T getP()
```

### *Boolean getter*

```
public boolean isP()
```

#### *Setter*

```
public void setP(T)
```

#### *Exceptions*

Property accessor methods can throw any type of checked or unchecked exceptions

### ***Indexed Properties***

An indexed property is a property of array type that provides accessor methods that get and set the entire array, as well as methods that get and set individual elements of the array. A bean defines an indexed property  $p$  of type  $T[]$  if it defines the following accessor methods:

#### *Array getter*

```
public T[] getP()
```

#### *Element getter*

```
public T getP(int)
```

#### *Array setter*

```
public void setP(T[])
```

#### *Element setter*

```
public void setP(int,T)
```

#### *Exceptions*

Indexed property accessor methods can throw any type of checked or unchecked exceptions. In particular, they should throw an `ArrayIndexOutOfBoundsException` if the supplied index is out of bounds.

### ***Bound Properties***

A bound property is one that generates a `PropertyChangeEvent` when its value changes. Here are the conventions for a bound property:

#### *Accessor methods*

The getter and setter methods for a bound property follow the same conventions as a regular property.

#### *Introspection*

A beanbox cannot distinguish a bound property from a nonbound property through introspection alone. Therefore, you may want to implement a `BeanInfo` class that returns a `PropertyDescriptor` object for the property. The `isBound()` method of this `PropertyDescriptor` should return `true`.

#### *Listener registration*

A bean that defines one or more bound properties must define a pair of methods for the registration of listeners that are notified when any bound property value change. The methods must have these signatures:

```
public void addPropertyChangeListener(PropertyChangeListener)
public void removePropertyChangeListener(PropertyChangeListener)
```

### *Named property listener registration*

A bean can optionally provide additional methods that allow event listeners to be registered for changes to a single bound property value. These methods are passed the name of a property and have the following signatures:

```
public void addPropertyChangeListener(String, PropertyChangeListener)
public void removePropertyChangeListener(String, PropertyChangeListener)
```

### *Per-property listener registration*

A bean can optionally provide additional event listener registration methods that are specific to a single property. For a property *p*, these methods have the following signatures:

```
public void addPListener(PropertyChangeListener)
public void removePListener(PropertyChangeListener)
```

Methods of this type allow a beanbox to distinguish a bound property from a nonbound property.

### *Notification*

When the value of a bound property changes, the bean should update its internal state to reflect the change and then pass a `PropertyChangeEvent` to the `propertyChange()` method of every `PropertyChangeListener` object registered for the bean or the specific bound property.

### *Support*

`java.beans.PropertyChangeSupport` is a helpful class for implementing bound properties.

## ***Constrained Properties***

A constrained property is one for which any changes can be vetoed by registered listeners. Most constrained properties are also bound properties. Here are the conventions for a constrained property:

### *Getter*

The getter method for a constrained property is the same as the getter method for a regular property.

### *Setter*

The setter method of a constrained property throws a `PropertyVetoException` if the property change is vetoed. For a property *p* of type *T*, the signature looks like this:

```
public void setP(T) throws PropertyVetoException
```

### *Listener registration*

A bean that defines one or more constrained properties must define a pair of methods for the registration of listeners that are notified when any constrained property value changes. The methods must have these signatures:

```
public void addVetoableChangeListener(VetoableChangeListener)
public void removeVetoableChangeListener(VetoableChangeListener)
```

#### *Named property listener registration*

A bean can optionally provide additional methods that allow event listeners to be registered for changes to a single constrained property value. These methods are passed the name of a property and have the following signatures:

```
public void addVetoableChangeListener(String, VetoableChangeListener)
public void removeVetoableChangeListener(String, VetoableChangeListener)
```

#### *Per-property listener registration*

A bean can optionally provide additional listener registration methods that are specific to a single constrained property. For a property *p*, these methods have the following signatures:

```
public void addPListener(VetoableChangeListener)
public void removePListener(VetoableChangeListener)
```

#### *Notification*

When the setter method of a constrained property is invoked, the bean must generate a `PropertyChangeEvent` that describes the requested change and pass that event to the `vetoableChange()` method of every `VetoableChangeListener` object registered for the bean or the specific constrained property. If any listener vetoes the change by throwing a `PropertyVetoException`, the bean must send out another `PropertyChangeEvent` to revert the property to its original value, and then it should throw a `PropertyVetoException` itself. If, on the other hand, the property change is not vetoed, the bean should update its internal state to reflect the change. If the constrained property is also a bound property, the bean should notify `PropertyChangeListener` objects at this point.

#### *Support*

`java.beans.VetoableChangeSupport` is a helpful class for implementing constrained properties.

## **Events**

In addition to `PropertyChangeEvent` events generated when bound and constrained properties are changed, a bean can generate other types of events. An event named *E* should follow these conventions:

#### *Event class*

The event class should directly or indirectly extend `java.util.EventObject` and should be named *EEvent*.

#### *Listener interface*

The event must be associated with an event listener interface that extends `java.util.EventListener` and is named *EListener*.

#### *Listener methods*

The event listener interface can define any number of methods that take a single argument of type *EEvent* and return `void`.



### *Listener registration*

The bean must define a pair of methods for registering event listeners that want to be notified when an *E* event occurs. The methods should have the following signatures:

```
public void addEventListener(EventListener)
public void removeEventListener(EventListener)
```

### *Unicast events*

A unicast event allows only one listener object to be registered at a single time. If *E* is a unicast event, the listener registration method should have this signature:

```
public void addEventListener(EventListener) throws TooManyListenersException
```

## **Methods**

A beanbox can expose the methods of a bean to application designers. The only formal convention is that these methods must be declared `public`. The following guidelines are also useful, however:

### *Method name*

A method can have any name that does not conflict with the property- and event-naming conventions. The name should be as descriptive as possible.

### *Parameters*

A method can have any number and type of parameters. However, beanbox programs may work best with no-parameter methods or methods that have simple primitive parameters.

### *Excluding methods*

A bean can explicitly specify the list of methods it exports by providing a `BeanInfo` implementation.

### *Documentation*

A bean can provide user-friendly, human-readable localized names and descriptions for methods through `MethodDescriptor` objects returned by a `BeanInfo` implementation.

## **Auxiliary Classes**

A bean can provide the following auxiliary classes:

### *BeanInfo*

To provide additional information about a bean *B*, implement the `BeanInfo` interface in a class named *BBeanInfo*.

### *Property editor for a specific type*

To enable a beanbox to work with properties of type *T*, implement the `PropertyEditor` interface in a class named *TEditor*. The class must have a no-parameter constructor.

#### *Property editor for a specific property*

To customize the way a beanbox allows the user to enter values for a single property, define a class that implements the `PropertyEditor` interface and has a no-parameter constructor, and register that class with a `PropertyDescriptor` object returned by the `BeanInfo` class for the bean.

#### *Customizers*

To define a customizer, or wizard, for configuring a bean *B*, define an AWT or Swing component with a no-parameter constructor that does the customization. The class is commonly called *BCustomizer*, but this is not required. Register the class with the `BeanDescriptor` object returned by the `BeanInfo` class for the bean.

#### *Documentation*

Define default documentation for a bean *B* in HTML 2.0 format and store that documentation in a file named *B.html*. Define localized translations of the documentation in files by the same name in locale-specific directories.

### ***Bean Packaging and Distribution***

Beans are distributed in JAR archive files that have the following:

#### *Content*

The class or classes that implement a bean should be included in the JAR file, along with auxiliary classes such as `BeanInfo` and `PropertyEditor` implementations. If the bean is instantiated from a serialized instance, that instance should be included in the JAR archive with a filename ending in *.ser*. The JAR file can contain HTML documentation for the bean and should also contain any resource files, such as images, required by the bean and its auxiliary classes. A single JAR file can contain more than one bean.

#### *Java-Bean attribute*

The manifest of the JAR file must mark any *.class* and *.ser* files that define a bean with the attribute:

```
Java-Bean: true
```

#### *Depends-On attribute*

The manifest of a JAR file can use the `Depends-On` attribute to specify all other files in the JAR archive on which the bean depends. A beanbox application can use this information when generating applications or repackaging beans. Each bean can have zero or more `Depends-On` attributes, each of which can list zero or more space-separated filenames. Within a JAR file, `/` is always used as the directory separator.

#### *Design-Time-Only attribute*

The manifest of a JAR file can optionally use the `Design-Time-Only` attribute to specify auxiliary files, such as `BeanInfo` implementations, that are used by a beanbox, but not used by applications that use the bean. The beanbox can use this information when repackaging beans for use in an application.

## *Bean Contexts and Services*

The JavaBeans component model was introduced in Java 1.1. Java 1.2 extends that model by introducing a containment and services protocol, defined in the `java.beans.beancontext` package. A bean context is a `java.util.Collection` of beans that implements the `BeanContext` interface and provides a context for the beans it contains. Many bean contexts define one or more services, such as a printing service, that beans can query and use. These bean contexts implement the `BeanContextServices` interface. All bean contexts are also `BeanContextChild` implementations, so contexts can be nested within each other.

Many beans never need to know about the contexts that contain them. A bean that does want to take advantage of its context and the services it provides implements the `BeanContextChild` interface. When a bean context child is added to a bean context, the `setBeanContext()` method of the `BeanContextChild` interface is invoked by the bean context. The implementation of this method should store the reference to the bean context for future use. The `setBeanContext()` method is a bound and constrained property, so it must notify `VetoableChangeListener` and `PropertyChangeListener` objects. For this reason, many beans delegate these responsibilities to a `BeanContextChildSupport` object.

If a bean (or bean context) is nested within a bean context that implements `BeanContextServices`, the bean can use the services provided by the bean context. A service is identified by the Java class that defines it. So a printing service is identified by the `Class` object of the `java.awt.print.PrinterJob` class, for example, and the system clipboard service is represented by the `java.awt.datatransfer.Clipboard` class. A bean can call the `hasService()` method of its containing `BeanContextServices` object to determine whether a specified service is available. If so, it can use `getService()` to obtain an appropriate instance of the service class. If a bean context is nested within another context, it can pass these `hasService()` and `getService()` methods to its containing context.

In addition to `getService()` and `hasService()`, a `BeanContext` provides several other methods beans can rely on. `getResource()` and `getResourceAsStream()` replace the methods by the same name defined by `Class` and `ClassLoader`. The `isDesignTime()` method (from the `DesignMode` interface) allows a bean to determine whether it is being displayed within a beanbox or run in an application or applet. The `BeanContext` method is preferred to the static `Beans.isDesignTime()` method because it is context-specific rather than global.

`BeanContext` and `BeanContextServices` are large interfaces; implementations must adhere to fairly complex specifications that govern the ways they interact with the beans they contain and the contexts within which they are nested. For these reasons, bean developers do not often create custom bean contexts. Instead, they rely on the contexts provided by the vendor of their beanbox tool. Advanced bean developers who do need to create bean contexts can delegate many of their methods to the `BeanContextSupport` and `BeanContextServicesSupport` classes that implement the basic framework and protocols.