Introduction to Java Beans

Before we can take a look at how JBuilder 2.0 supports the development of JavaBeans, it is preferred to look at some background theory of JavaBeans. This first part will focus on the JavaBeans definition. We will see what is necessary to define a Java class as a JavaBean. And because JavaBeans are a component model for Java, we will take a look at component technology in general, and the advantages of components.

Component technology

Suppose we want to build our own car. A car is made up of many components. For example the chassis, the motor block, and interior. And each of these components is made up of many more components. The interior consists of chairs, steering wheel, speed meter, air conditioning, and we can go on and on. When we build the car we have the following choice for each component we use: build the component from scratch on our own or buy the component from a manufacturer. The choice for building or buying will be decided on a couple of parameters. A few examples of parameters are: availability, money, time, personal choice. We can imagine that building a component on our own will take more time, than buying a component. But at the same time, if we have plenty of time, because of our retirement, this wouldn't be a problem, and we can save money by building it for ourselves. If the component is something never build before, we don't have a choice: we have to build it on our own. We conclude that there are good reasons for both buying and building components.

Now replace the car with a software program. The software program can also be divided into several components. For example a word processor, can be divided into a text component, spelling and grammatical components, formatting components, and many more. And if we want to build our own word processor we can ask ourselves the same questions for each component as we did when we wanted to build a car: buy or build. We can buy a spelling component, so our word processor will be able to check the spelling of a document. But we can also build the spelling component and use it in our word processor. And we can sell our spelling component to other application builders.

As a developer of applications we don't need to know everything, and we can focus on problems we can control, and use components on problems already solved by other developers. This can save a lot of time, and money.

These examples show that components are easy to re-use. But before a component is easy to re-use, standards need to be defined, to which the component needs to apply. A component seller could sell a lot more components if it applies to a software standard that a lot of component users use for their applications. ActiveX, formerly OCX, is a component standard on the Microsoft Windows platform. Microsoft has defined certain rules that everybody needs to follow. And because of these standards, there are a lot of ActiveX components available to buy and use in the application building process.

Borland Delphi developers are very familiar with the concept of software components, because the Visual Component Library (VCL) is a great example of a library of components.
The Java platform didn't have this kind of software component model. We could re-use our previous written classes in our own application building process. But interchanging our components with our developers, wasn't that easy, because of the missing standards. And this is where the JavaBeans APIs comes in.

The JavaBeans APIs define a software component model for Java. This way developers can create Java components, sell Java components, and compose Java components together into applications. So at last we can do what the Delphi developers already could: create and use software components.

**What is a Bean?**

Now we know that the JavaBeans APIs define a component standard, to which Bean developers need to obey. But what is a Java Bean? The JavaBeans specification contains the following definition of a Java Bean:

*A Java Bean is a reusable software component that can manipulated visually in a builder tool.*

We have seen in the previous section what a reusable software component is, but what about the visual manipulation in a builder tool. What is a builder tool?

Builder tools or application builders provide the environment to use the beans and manipulate them. Borland JBuilder is a good example of a builder tool, Borland Delphi is another builder tool. A builder tool is able to connect different beans together in a visual manner. So without writing any code, we should be able to manipulate a bean. Most builder tools contain a palette of components from which a developer can drag-and-drop components onto a form. A form can be a Java panel, window or canvas.

The functionality a bean supports will be different for every bean, but there are some features that distinguish a Java Bean as a bean:

- Support for *introspection*; a builder tool can analyze how a bean works.
- Support for *properties*; used for customization of the state of a bean.
- Support for *events*; a way of beans to tell the outside something is happening, or reacting to something that happened outside the bean.
- Support for *customization*; a user of the bean can customize the appearance and behavior of the bean within a builder tool.
- Support for *persistence*; a bean can be customized in a builder tool and have its state saved away and reloaded later.

We now know the features of a Java Bean, but how do we make a Java class to a bean? Is there perhaps a super Java Bean class we need to inherit from? The answer is no! A bean is not required to inherit from a particular class or to implement a specific interface. So to build a bean all we have to do is to write a Java class, and obey certain rules. Here are the main rules:

- The bean class must provide a zero-argument constructor.
- The bean must support persistence, by implementing the Serializable or Externalizable interface.
So almost every Java class is a Java Bean? Basically, yes. But there is more. The Java Beans framework introduces a lot of conventions and design patterns. The conventions and design patterns are all optional, we don't have to use them. But by using the conventions, we will create really useful beans, that can be used within a builder tool. A builder tool, like JBuilder, depends on these conventions and design patterns to introspect a bean. This way JBuilder is able to look for properties and events for example. But before we take a close look to the introspection mechanism, we first look at bean behavior during design time and run-time.

**Design time vs. Run-time**
Each Java Bean has to be able to run in two different environments. First of all the bean needs to be capable of running inside a builder tool. This is often referred to as the design environment. The bean must be able to provide the builder tool with design information. The user must be able to customize the bean visually in the builder tool. For this customization process a lot of extra baggage is carried around by the bean. Metadata, property editors, customizers, icons all need to be included with the Java Bean. The bean must also be able to be used during run-time within a generated application. During run-time there is much less need for customization of the behavior and appearance of the Java Bean. And therefore a run-time bean carries much less baggage around.

**Non-visual beans**
Most of the time a Java Bean will have a GUI representation of itself. For example a button or calendar bean, must have a GUI representation to be useful. But there are also beans that don't need a GUI representation. For example a spelling bean, that only will check a word for spelling errors, doesn't need a GUI representation. We will call these type of beans non-visual beans.

But the invisibility only applies to the run-time representation of the bean. The bean stills needs to be customized from within the builder tool. So the bean can show itself inside the builder tool. We can change properties, look at events, and do more, visually within the builder tool, but at run-time, the bean will not be seen by the end-user.

The visible bean is visible within both the builder tool as in the run-time application.

**Introspection**
We saw earlier that JBuilder, as a builder tool, is able to look inside a bean. JBuilder "knows", through a process called introspection, which properties, events, and methods a bean contains. JBuilder uses the Reflection API, for this purpose. We as bean builders can use the Java Beans design patterns or naming conventions to help JBuilder determine the characteristics of our bean. We can also get around these conventions by supplying our bean with a BeanInfo class. The class contains the information about the bean, we want the builder tool to be able to see. In this class we ourselves can decide what properties, events and methods to show to the outside world. Later on we will take a closer look at this BeanInfo class. For now we will focus on using the naming conventions. The following table shows the naming conventions for properties, events, and methods. These will be explained in the following sections.

<table>
<thead>
<tr>
<th>Simple property</th>
<th>public void setPropertyName(PropertyType value);</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>public PropertyType getPropertyName();</td>
</tr>
<tr>
<td>Property Type</td>
<td>Methods</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Boolean property</td>
<td>public void setPropertyName(boolean value); public boolean isPropertyName();</td>
</tr>
<tr>
<td>Indexed property</td>
<td>public void setPropertyName(int index, PropertyType value); public PropertyType getPropertyName(int index); and public void setPropertyName(PropertyType[] value); public PropertyType[] getPropertyName();</td>
</tr>
<tr>
<td>Multicast events</td>
<td>public void addEventListenerType(EventListenerType l); public void removeEventListenerType(EventListenerType l);</td>
</tr>
<tr>
<td>Unicast events</td>
<td>public void addEventListenerType(EventListenerType l) throws TooManyListenersException; public void removeEventListenerType(EventListenerType l);</td>
</tr>
<tr>
<td>Methods</td>
<td>public methods</td>
</tr>
</tbody>
</table>

**Properties**
A property is an attribute of the Java Bean that can affect its appearance or behavior. For example a button can have a property named `enabled`, which represents if the state of the button is enabled or disabled. A property can be read/write, read-only, or write-only. The read/write state of a property is determined by the availability of accessor methods. A property is read/write, if the property has both a get and a set method. A property is read-only, if only a get method for the property is defined. And for a write-only property, only the set method is defined.

The name of the property is derived from the accessor methods names. For example, the methods `getInprise` and `setInprise` indicate a property named `Inprise`.

**Simple properties**
A simple property represent a single value. By using an isPropertyName() getter method we flag to the introspector a single-valued boolean property.

```java
// member used to store property value
//
private String internalString;
// set and get method for property named String
//
public void setString(String newValue) { internalString = newValue; }
public String getString() { return internalString; }
// member used to store property value
//
private boolean connect;
// set and get method for boolean property name connected
//
public void setConnected(boolean newValue) { connect = newValue; }
public boolean isConnected() { return connect; }
```
Indexed properties
An indexed property represent a array of values. And we can use additional get and set methods for an indexed property. This get and set method take an integer index parameter. This index parameter indicates the place in the array where the new property value has to be put. If for example we want the first value from the indexed property we would use a get method like this: getPropertyName(1).

```
// member to store indexed property
//
private int[] numbers = {1, 2, 3, 4};
// set and get method for complete array
//
public void setNumbers(int[] newValue) { numbers = newValue; }
public int[] getNumbers() { return numbers; }
// set and get method with index to set one element of array
//
public void setNumbers(int index, int newValue) { numbers[index] = newValue; }
public int getNumbers(int index) { return numbers[index]; }
```

Bound properties
A bound property is a property that notifies other objects when its values changes. When the value changes a PropertyChangeEvent is fired. This event contains the property name, the old, and the new value. Other beans or objects can register themselves as listeners to this event. These objects will then receive the PropertyChangeEvent when it is fired from the bean.

```
// declare a property change object and instantiate it
//
private PropertyChangeSupport changes = new PropertyChangesSupport(this);
// private member used to store property value
//
private String internalString;
// set and get method for property named String
//
public void setString(String newValue) {
    // save current value of property
    String oldValue = internalString;
    // assign new value of property
    internalString = newValue;
    // send change event to listeners, because the property has changed
    changes.firePropertyChangeEvent("string", oldValue, newValue);
}
public String getString() { return internalString; }
// implement methods to add and remove listeners
//
public void addPropertyChangeListener(PropertyChangeListener l) {
    changes.addPropertyChangeListener(l);
}
public void removePropertyChangeListener(PropertyChangeListener l) {
    changes.removePropertyChangeListener(l);
}
```
Constrained properties
The last type of property is the constrained property. When the value of the property changes a VetoableChangeEvent is fired. Listeners of this event can veto the change of the property value. A listeners vetoes the change by throwing a PropertyVetoException. So the following happens when a constrained properties value is changed:

1. The bean fires a VetoableChangeEvent.
2. Any listener to this event, will receive this event.
3. According to some rule, this listener will accept or veto the change. When a veto occurs the listener throws a PropertyVetoException.
4. Next the bean will fire a PropertyChangeEvent to any listeners for this event, to tell them the property has changed.

```java
// declare a property change object and instantiate it
//
private PropertyChangeSupport changes = new PropertyChangesSupport(this);
// declare a vetoable change object and instantiate it
//
private VetoableChangeSupport vetos = new VetoableChangeSupport(this);
// member used to store property value
//
private String internalString;
// set and get method for property named String
// The set method throws PropertyVetoException
//
public void setString(String newValue) throws PropertyVetoException {
    // save current value of property
    String oldValue = internalString;
    // tell interested vetoable change listener about the change
    vetos.fireVetoableChange("string", oldValue, newValue);
    // assign new value of property
    internalString = newValue;
    // send change event to listeners, because the property has changed
    changes.firePropertyChangeEvent("string", oldValue, newValue);
}
public String getString() { return internalString; }
// implement methods to add and remove listeners for vetoable change events
//
public void addVetoableChangeListener(VetoableChangeListener l) {
    changes.addVetoableChangeListener(l);
}
public void removeVetoableChangeListener(VetoableChangeListener l) {
    changes.removeVetoableChangeListener(l);
}
// implement methods to add and remove listeners for property change events
//
public void addPropertyChangeListener(PropertyChangeListener l) {
    changes.addPropertyChangeListener(l);
}
public void removePropertyChangeListener(PropertyChangeListener l) {
    changes.removePropertyChangeListener(l);
}
```
**Events**
Java Beans support the Java 1.1 event model. The 1.1 event handling model is based on the concept of an event listener. An object interested in receiving events is an event listener. An object that generates events is called an event source. In this case our bean is an event source and the application implementing the bean is an event listener. The event source maintains a list of listeners and provides the methods to add and remove interested listeners to and off the list. So we need to add methods to add and remove interested listeners to our bean.

When the specific event happens the event source notifies all registered listeners. The event source notifies the listeners by invoking a method on it and passing it an event object. This event object is a subclass of the java.util.EventObject class. So we need to create our own Event class. For this scheme to work all listeners must implement the required method. This is achieved by requiring all listeners to implement the corresponding interface to an event. An event listener interface is implemented by extending the java.util.EventListener interface. This interface contains no methods and is only a marker, to clearly identify the interface as such.

These are the steps:

1. Create our own custom event class, named XXXEvent.
2. Create the XXXListener interface class, containing methods that listener object need to overwrite.
3. Add a property to the bean containing a list of listener objects.
4. Add an addXXXListener and removeXXXListener method, where XXX stands for the name of the event.
5. Add a method that will notify all registered listeners.

Most of the time we want our bean to sent a event to all registered listeners. But sometimes we just want to connect to one listener only. This can be achieved by throwing a TooManyListenersException at the addXXXListener method. When a listener is already connected to the bean, another listener trying to connect will be denied.

**Step 1.**

```java
// our event named OurEvent
//
public class OurEventObject extends java.util.EventObject {
    // only implementation of simple constructor
    //
    public OurEventObject(Object source) {
        super(source);
    }
}
```

**Step 2.**

```java
// event listener interface for OurEvent, so it is named OurEventListener
//
public interface OurEventListener extends java.util.EventListener {
```
Step 3.

// method that the listener needs to provide with a body
//
// eventRaised(OurEventObject e);

Step 4.

// maintains list of objects which have registered to receive our event
//
private Vector listeners = new Vector();

// method to add listeners to the bean
// this method is synchronized because multiple listeners can call this
// method at the same time
//
public synchronized addOurEventListener(OurEventListener l) {
    listeners.addElement(l);
}

// method to remove listeners from the bean
//
public synchronized removeOurEventListener(OurEventListener l) {
    listeners.removeElement(l);
}

Step 5.

// method that will notify all listeners of OurEvent
//
public void fireOurEvent() {
    Vector l;
    // make a workable copy of available listeners
    //
    synchronized (this) { l = (Vector)listeners.clone(); }
    // loop until all listeners have been reached and invoke the method
    // defined in OurEventListener interface
    //
    for (int i=0; i < l.size(); i++) {
        OurEventListener listener = (OurEventListener) l.elementAt(i);
        listener.eventRaised(new OurEventObject(this));
    }
}
**Methods**
By simply declaring a method public in our bean, the builder tool will know about the method.

```java
// by simply declaring a method public in a bean,
// will the method be exposed to the builder tool
// by way of introspection

public void myMethod() { /* empty, do nothing */ }
```

**Persistent storage**
To make a Java Bean persistent, the bean simply needs to implement the java.io.Serializable interface. By implementing this interface the fields of a bean will automatically be saved. We don't have to do anything else! To prevent certain fields of being saved we can use the keywords transient or static. A transient or static property is not saved.

```java
// implementing the java.io.Serializable interface makes a bean persistent

public class MyBean
    extends java.lang.Object
    implements java.io.Serializable {
```