

JAVABEANS™

WHAT MAKES A CLASS TO BE A COMPONENT

DETAILED VIEW



JAVABEANS™

JavaBeans™ makes it easy to reuse software components

Developers can use software components written by others without having to understand their inner workings.

JavaBeans are classes encapsulating many objects into a single object (the bean).

They are serializable, have a zero-argument constructor, and allow access to properties using getter and setter methods.

JAVABEANS™

Design of JavaBeans components is easy.

No need a special tool or to implement any interfaces.

Writing beans is a matter of following certain coding conventions.

All you have to do is to make your class look like a bean

JAVABEANS™

A bean is a Java class with **method** names that **follow** the **JavaBeans guidelines**.

A bean builder tool uses **introspection** to **examine** the **bean class**.

Based on this inspection, the bean builder tool can figure out the bean's **properties**, **methods**, and **events**.

JAVABEANS™

There are guidelines for properties, methods, and events.

EXAMPLE BEAN

1. Simple bean properties

```
public class FaceBean {  
    private int mMouthWidth = 90;  
  
    public int getMouthWidth() {  
        return mMouthWidth;  
    }  
  
    public void setMouthWidth(int mw) {  
        mMouthWidth = mw;  
    }  
  
}
```

2. Boolean has an exception

```
public boolean isRunning() {  
    // ...  
}
```

BOUND PROPERTIES

A bound property notifies **listeners** when its **value changes**.

This has two implications:

- The bean class includes `addPropertyChangeListener()` and `removePropertyChangeListener()` methods for managing the bean's listeners.
- When a **bound property is changed**, the bean sends a `PropertyChangeEvent` to its registered **listeners**.

```
public class FaceBean {  
    private int mMouthWidth = 90;  
    private PropertyChangeSupport mPcs =  
        new PropertyChangeSupport(this);  
}
```

BOUND PROPERTIES

```
import java.beans.*;

public class FaceBean {
    private int mMouthWidth = 90;
    private PropertyChangeSupport pcs =
        new PropertyChangeSupport(this);

    public int getMouthWidth() {
        return mMouthWidth;
    }

    public void setMouthWidth(int mw) {
        int oldMouthWidth = mMouthWidth;
        mMouthWidth = mw;
        pcs.firePropertyChange("mouthWidth",
                                oldMouthWidth, mw);
    }
}
```

```
..
public void addPropertyChangeListener (
    PropertyChangeListener listener) {
    pcs.addPropertyChangeListener(listener);
}

public void removePropertyChangeListener (
    PropertyChangeListener listener) {
    pcs.removePropertyChangeListener(listener);
}
}
```


CONSTRAINED PROPERTIES



A **constrained property** is a special kind of bound property.

For a constrained property, the bean keeps track of a set of **veto** listeners.

When a constrained property is **about to change**, the listeners are **consulted** about the change.

Any one of the listeners has a chance to **veto** the change, in which case the property **remains unchanged**.

```
public class FaceBean {  
    private int mMouthWidth = 90;  
    private PropertyChangeSupport mPcs =  
        new PropertyChangeSupport(this);  
    private VetoableChangeSupport mVcs =  
        new VetoableChangeSupport(this);  
}
```

CONSTRAINED PROPERTIES

```
import java.beans.*;

public class FaceBean {
    private int mMouthWidth = 90;
    private PropertyChangeSupport pcs =
        new PropertyChangeSupport(this);
    private VetoableChangeSupport vcs =
        new VetoableChangeSupport(this);

    public int getMouthWidth() {...}

    public void setMouthWidth(int mw)
        throws PropertyVetoException {
        int oldMouthWidth = mMouthWidth;
        vcs.fireVetoableChange("mouthWidth",
            oldMouthWidth, mw);
        mMouthWidth = mw;
        pcs.firePropertyChange("mouthWidth",
            oldMouthWidth, mw);
    }
}
```

```
..

public void addPropertyChangeListener (
    PropertyChangeListener listener) {
    pcs.addPropertyChangeListener(listener);
}

public void removePropertyChangeListener (
    PropertyChangeListener listener) {
    pcs.removePropertyChangeListener(listener);
}

public void addVetoableChangeListener (
    VetoableChangeListener listener) {
    vcs.addVetoableChangeListener(listener);
}

public void removeVetoableChangeListener (
    VetoableChangeListener listener) {
    vcs.removePropertyChangeListener(listener);
}
}
```

METHODS

A bean's methods are the things it can do.

Any public method that is not part of a property definition is a bean method.

EVENTS

A bean class can fire off any type of event, including custom events.

As with properties, events are identified by a specific pattern of method names.

```
public void add<Event>Listener(<Event>Listener l);  
public void remove<Event>Listener(<Event>Listener l);
```

The listener type must be a descendant of `java.util.EventListener`.

```
public void addActionListener(ActionListener l);  
public void removeActionListener(ActionListener l);
```

EVENTS

EXAMPLE LISTENER

```
//... initialization occurs:
double amount;

JFormattedTextField amountField;
...
amountField.addPropertyChangeListener("value",
                                     new FormattedTextFieldListener());
...

class FormattedTextFieldListener implements PropertyChangeListener {
    public void propertyChanged(PropertyChangeEvent e) {
        Object source = e.getSource();
        if (source == amountField) {
            amount = ((Number) amountField.getValue()).doubleValue();
            ...
        }
        ...//re-compute payment and update field...
    }
}
```

BEAN PERSISTENCE

A bean has the property of **persistence** when its **properties**, **fields**, and **state** information are **saved** to and **retrieved from storage**.

Component models provide a mechanism for persistence that enables the state of components to be stored in a non-volatile place for later retrieval.

The mechanism that makes persistence possible is called **serialization**. Object serialization means converting an object into a data stream and writing it to storage.

Any application that uses that bean can then "reconstitute" it by **deserialization**. The object is then restored to its original state.

All beans must persist. To persist, your beans must support serialization by implementing either the `java.io.Serializable`

CONTROLLING SERIALIZATION

You can control the level of serialization that your beans undergo. Three ways to control serialization are:

1. Automatic serialization, implemented by the `Serializable` interface. The Java serialization software serializes the entire object, **except transient and static fields**.
2. Customized serialization. **Selectively exclude** fields you do not want serialized by marking with the **transient** (or **static**) modifier.
3. Customized file format, implemented by the `Externalizable` interface and its two methods. Beans are written in a specific file format.

```
private void writeObject(java.io.ObjectOutputStream out)
    throws IOException;

private void readObject(java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```

LONG TERM PERSISTENCE

Long-term persistence is a model that enables beans to be saved in XML.

The **XMLEncoder** class is assigned to write output files for textual representation of Serializable objects.

Writing a Java bean and its Properties in XML format:

The **XMLDecoder** class reads an XML document that was created with XMLEncoder:

```
XMLEncoder encoder = new XMLEncoder(  
    new BufferedOutputStream(  
        new FileOutputStream("Beanarchive.xml")));  
encoder.writeObject(object);  
encoder.close();
```

```
XMLDecoder decoder = new XMLDecoder(  
    new BufferedInputStream(  
        new FileInputStream("Beanarchive.xml")));  
Object object = decoder.readObject();  
decoder.close();
```