 ***Table of Contents***

JavaBeans Tutorial 

JavaBeans brings component technology to Java. JavaBeans lets your write Java classes, called Beans, that you can visually manipulate within application builder tools. The place to start learning about JavaBeans is the JavaBeans API Specification.

The software you'll need to understand and explore Beans is available free on the web. In addition to the Beans Development Kit (BDK) version 1.0, you will need the Java Development Kit (JDK) version 1.1.

This document describes what makes a Bean, and shows you how to use the BDK to write Beans.

 Introducing JavaBeans defines a Bean and Bean concepts, describes the BDK contents and the demonstration Beans, and discusses future Bean directions.

 BeanBox Basics describes the BeanBox: The BDK Bean reference container. You'll learn how to start the BeanBox, view events, generate property reports, serialize components, create a minimal Bean, save a Bean, and add a Bean to the ToolBox.

 Writing a Simple Bean describes how to incorporate properties, events, and persistence within your Bean, and how to generate an applet from the BeanBox.

 Writing Advanced Beans describes how to incorporate bound, constrained, and indexed properties into your Beans, how to use BeanInfo to advertise your Beans' capabilities to application builder tools, how to use your own Bean customizers, and how to convert your Beans into Active−X components.

Additional Documentation

The BDK's `beans/docs` directory contains documentation for

- The Beans API
- The BeanBox API
- The demo Beans
- The `java.util` API
- Java Archive (JAR) files and manifests
- Makefiles for gnumake (Unix) and nmake (Windows)

A good starting point is the file `beans/README.html`.

The JavaBeans Documentation page contains current JavaBeans API definitions, upcoming JavaBeans feature descriptions, and related Java documentation such as the Java Core Reflection API, Object Serialization, Remote Method Invocation (RMI), and Third−party JavaBeans books.

Table of Contents

 ***Table of Contents***

JavaBeans Overview

- JavaBeans Concepts briefly explains Bean fundamentals.
- BDK Contents is a roadmap to the BDK contents.
- BDK Demonstration Bean Descriptions describes the demo Beans shipped with the BDK.
- Future JavaBeans Features lets you know what is coming down the road for JavaBeans.

 ***Table of Contents***

JavaBeans Concepts

The JavaBeans API makes it possible to write component software in Java. Components are self−contained, reusable software units that can be visually composed into applets or applications using visual application builder tools.

JavaBeans is a core JDK1.1 capability: Any JDK1.1−compliant browser or tool implicitly supports JavaBeans.

JavaBean components are called Beans. A "Beans aware" builder tool maintains Beans in a palette or toolbox. You can select a particular Bean from the toolbox, drop it into a form, modify it's appearance and behavior, define its interaction with other Beans, and compose it and other Beans into an applet, application, or new Bean. All this can be done without writing a line of code.

The following list briefly describes key Bean concepts:

- Builder tools discover a Bean's properties, methods, and events by *introspection*. Beans support introspection in two ways:
  - By adhering to specific naming conventions, known as *design patterns*, when naming Bean features. Bean introspection relies on the *core reflection* API to discover Bean features via design patterns.
  - By explicity providing property, method, and event information with a related *Bean Information* class. A Bean information class implements the `BeanInfo` interface.

  See Chapter 8 of the JavaBeans API Specification for an introspection, design pattern, and `BeanInfo` discussion.
- *Properties* are a Bean's appearance and behavior attributes that can be changed at design time. Properties are exposed to builder tools by design patterns or a `BeanInfo` class. See Chapter 7 of the JavaBeans API Specification for a complete property discussion.
- Beans expose properties so that they can be *customized* at design time. Customization is supported in two ways: By using property editors, or by using more sophisticated Bean customizers. See Chapter 9 of the JavaBeans API Specification for a customization discussion.
- Beans use *events* to communicate with other Beans. A Bean that wants to receive events (a listener Bean) registers its interest with the Bean that fires the event (a source Bean). Builder tools can examine a Bean and determine which events that Bean can fire (send) and which it can handle (receive). See Chapter 6 of the JavaBeans API Specification for a complete event discussion.
- *Persistence* enables Beans to save their state, and restore that state later. JavaBeans uses Java Object Serialization to support persistence. See Chapter 5 of the JavaBeans API Specification for a complete persistence discussion.

- A Bean's *methods* are no different than Java methods, and can be called from other Beans or a scripting environment. By default a all public methods are exported.

  Beans can be used with both builder tools, or manually manipulated by text tools through programmatic interfaces. All key APIs, including support for events, properties, and persistence, have been designed to be easily read and understood by human programmers as well as by builder tools.

JavaBeans Overview

BDK Contents Described

THIS SECTION IS UNDER CONSTRUCTION

Here is a general description of the BDK files and directories:

- README.html contains an entry point to the BDK documentation
- LICENSE.html contains the BDK license agreement
- GNUmakefile and Makefile are Unix and Windows makefiles (.gmk and .mk suffixes) for building the demos and the BeanBox, and for running the BeanBox
- beans/apis contains
  - a java directory containing JavaBeans source files
  - a sun directory property editor source files
- beans/beanbox contains
  - makefiles for building the BeanBox
  - scripts for running the BeanBox
  - a classes directory containing the BeanBox class files
  - a lib directory containing a BeanBox support jar file used by MakeApplet's produced code
  - sun and sunw directories containing BeanBox source (.java) files
  - a tmp directory containing automatically generated event adapter source and class files, .ser files, and applet files automatically generated by MakeApplet
- beans/demos contains
  - makefiles for building the demo Beans
  - an html directory containing an applet wrapper demonstration that must be run in appletviewer, HotJava, or JDK1.1−compliant browsers
  - a sunw directory containing
    - a wrapper directory containing a bean applet wrapper
    - a demos directory containing demo source file
- beans/doc contains
  - demos documentation
  - a javadoc directory containing JavaBeans and JavaBeans−related class and interface documentation
  - miscellaneous documentation
- beans/jars contains jar files for demo Beans

 ***JavaBeans Overview***

The BDK Demonstration Beans

Description of the BDK demo Beans. UNDER CONSTRUCTION.

 ***JavaBeans Overview***


JavaBeans Overview

Future BDK Features

This file will contain info on future features. UNDER CONSTRUCTION


JavaBeans Overview


Table of Contents

BeanBox Basics

The BeanBox is an application for testing Beans, and serves as a simple demonstration for how more sophisticated Beans−aware builder tools should behave. The BeanBox is also a nice tool for learning about Beans. You will use the BeanBox to help you understand Bean characteristics.

Starting and Using the BeanBox explains how to start the BeanBox, and describes the ToolBox and Properties sheet.

Creating a Minimal Bean walks you through creating a rudimentary Bean, saving the Bean, adding the Bean to the ToolBox, placing the Bean in the BeanBox, inspecting the Bean's properties and events, and generating a Bean introspection report.

The BeanBox Menus explains each item in the BeanBox File, Edit, and View menus.


Table of Contents

Starting and Using the BeanBox

The `beans/beanbox.` directory contains Windows (run.bat) and Unix (run.sh) scripts that start the BeanBox. You can use these commands to start the BeanBox, or use make:

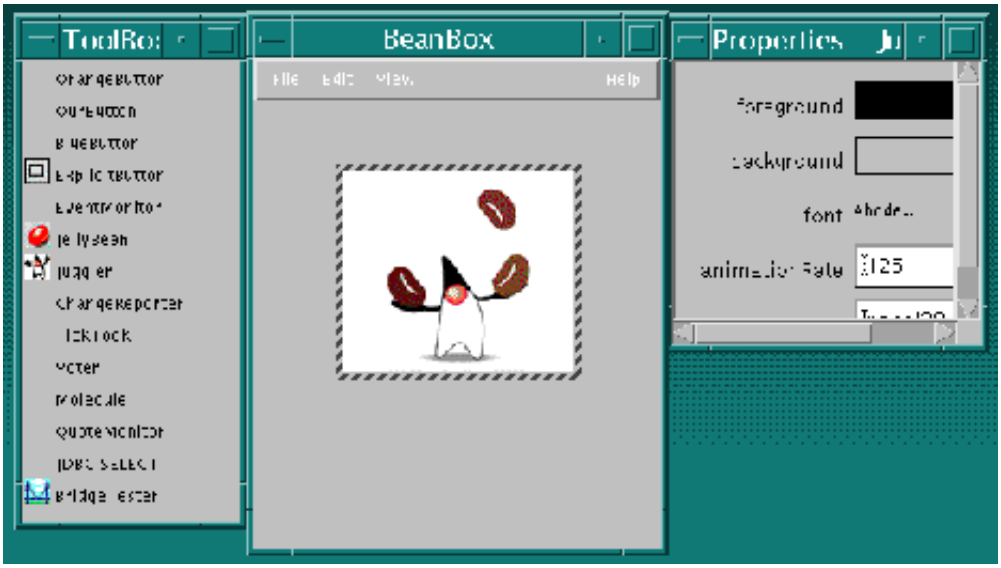
```
gnumake run
```

Or:

```
nmake run
```

The BDK files `beans/doc/makefiles.html` and `beans/doc/gnu.txt` for information about getting copies of these make tools.

When started, the BeanBox displays three windows: The BeanBox, ToolBox, and the Properties sheet. Here's how they look running under CDE (Common Desktop Environment):



This illustration shows the BeanBox with the Juggler demo Bean placed within it. The hatching around the Juggler is how the BeanBox indicates the selected Bean. Clicking on a Bean selects it. On Windows, it's a little tricky; you must click in the area where the hatched border *would appear* to select the Bean. On other systems clicking

anywhere on the Bean will select it.

Adding a Bean to the ToolBox

The ToolBox contains the Beans available for use by the BeanBox. When the BeanBox is started, it automatically loads all the Beans it finds within the JAR files contained in the `beans/jars` directory. Move your JAR files into that directory to have them automatically loaded at BeanBox startup. After the BeanBox is running you can load Beans Using the File|LoadJar... BeanBox menu item. Only Beans in the `beans/jars` directory will be loaded between startups though. Clicking on a Bean name within the ToolBox chooses that Bean for placement within the BeanBox. You will see the cursor change to a crosshair.

Dropping a Bean on the BeanBox

To add a `JellyBean` to the BeanBox

1. Click on the word JellyBean in the ToolBox. The cursor will change to a crosshair.
2. Click within the BeanBox. The JellyBean will appear and will be selected.

Note the change in the Properties sheet when you put the JellyBean in the BeanBox. Before you placed the JellyBean in the BeanBox, the BeanBox's properties were displayed; after placing the JellyBean in the BeanBox, the JellyBean's properties are displayed. If you missed the change, click within the BeanBox, away from the JellyBean. This will select the BeanBox rather than the JellyBean. The Properties sheet will then display the BeanBox's properties.

The Properties sheet displays the selected Bean's properties. After dropping a `JellyBean` instance on the BeanBox, the Properties sheet displays the `JellyBean` properties: color, foreground, priceInCents, background, and font.

Editing Bean Properties

The Properties sheet displays each property's name and its current value. Values are displayed in an editable text field (strings and numbers), a choice menu (booleans), or as painted values (colors and fonts). You can edit the property value by clicking on a property within the Properties sheet. Properties displayed in text fields or choice menus are edited within the Properties sheet. When you click on a color or font property a separate panel will pop up to do the editing. These property types use a *custom property editor*. Try clicking on each of the `JellyBean` properties.

Saving and Restoring Beans

The BeanBox uses Java Object Serialization to save and restore Beans and their state. The following steps demonstrate how to save and restore a Bean:

1. Drop a `JellyBean` on the BeanBox.
2. Change the `color` property to anything you want.

3. Select the File | Save BeanBox menu item. A file browser will pop up; use it to save the Bean to a file.
4. Select the File | Clear BeanBox menu item.
5. Select the File | Load BeanBox menu item. The file browser will again pop up; use it to retrieve the serialized Bean.

   In the next lesson you learn more BeanBox capabilities by creating a simple Bean.

BeanBox Basics

The BeanBox Menus

THIS SECTION IS UNDER CONSTRUCTION

This section explains each item in the BeanBox File, Edit, and View menus.

File Menu

Save

Saves the Beans in the BeanBox, including each Bean's size, position, and internal state. The saved file can be loaded via File | Load.

SerializeComponent...

Saves the Beans in the BeanBox to a serialized (*.ser*) file. This file must be put in a `.jar` file to be useable.

MakeApplet...

Generates an applet from the BeanBox contents.

Load...

Loads Saved files into the BeanBox. Will not load `.ser` files.

LoadJar...

Loads a Jar file's contents into the ToolBox.

Print

Prints the BeanBox contents.

Clear

Removes the BeanBox contents.

Exit

Quits the BeanBox *without offering to save.*

Edit Menu

Cut

Removes the Bean selected in the BeanBox. The cut Bean is serialized, and can then be pasted.

Copy

Copies the Bean selected in the BeanBox. The copied Bean is serialized, and can then be pasted.

Paste

Drops the last cut or copied Bean into the BeanBox.

Report...

Generates an introspection report on the selected Bean.

Events

Lists the event−firing methods, grouped by interface.

Bind property...

Lists all the bound property methods of the selected Bean.

View Menu

Disable Design Mode

Removes the ToolBox and the Properties sheet from the screen. Eliminates all

beanBox design and test behavior (selected Bean, etc.), and makes the BeanBox behave like an application.

Hide Invisible Beans

Hides invisible Beans.

BeanBox Basics

Creating a Minimal Bean

In this section you will continue learning some BeanBox fundamentals by

- Creating a minimal Bean
- Compiling and saving the Bean into a Java Archive (JAR) file (by using a makefile)
- Loading the Bean into the ToolBox
- Dropping a Bean instance into the BeanBox
- Inspecting the Bean's properties, methods, and events
- Generating an introspection report

Your example Bean will be named `SimpleBean`, and you will create it in the `beans/demo/sunw/demo/simple` directory:

1. **Create the directory** `beans/demo/sunw/demo/simple`. This will put SimpleBean in with the other BDK demo Beans.

2. **Write the SimpleBean code**. Since you're new to Beans, we'll give it to you:

```
package sunw.demo.simple;

import java.awt.*;
import java.io.Serializable;

public class SimpleBean extends Canvas
                      implements Serializable{

  //Constructor sets inherited properties
  public SimpleBean(){
   setSize(60,40);
   setBackground(Color.red);
  }

}
```

`SimpleBean` extends the `java.awt.Canvas` component. `SimpleBean` also implements the java.io.Serializable interface, a requirement for all Beans. Setting the background color and component size is all that `SimpleBean` does.

3. **Write the makefile**. Makefiles for Unix (gnumake) and Windows (nmake) are listed at the end of this section. These makefiles compile `SimpleBean`, and generate a JAR file in `beans/jars` (where the other demo jars reside). Save one of these makefiles as simple.gmk (Unix) or simple.mk (Windows), and put it in `beans/demo`.

4. **Run make** from the beans/demos directory. For Unix, run

```
gnumake simple.gmk
```

   For Windows, run

```
nmake -f simple.mk
```

   This will compile SimpleBean, and create a JAR file in the `beans/jars` directory. The BeanBox looks in that directory for JAR files.

5. **Load the JAR file** into the ToolBox. Pull down the File|LoadJar... menu item. This will bring up a file browser. Navigate to `beans/jars` and select simple.jar. SimpleBean will appear at the bottom of the ToolBox. Note that when the BeanBox is started, all Beans in JAR files in the `beans/jars` directory are automatically loaded into the ToolBox.

6. **Drop an instance of SimpleBean into the BeanBox**. Click on the word SimpleBean in the ToolBox. The cursor will change to crosshairs. Move the cursor to a spot within the BeanBox and click. SimpleBean will appear as a painted rectangle with hatched border. This border means that `SimpleBean` is selected. The `SimpleBean` properties will appear in the Properties sheet.

   You can resize `SimpleBean` (because it inherits from Canvas) by dragging a corner. You will see the cursor change to a right angle when over a corner. You can reposition SimpleBean within the BeanBox by dragging on any non-corner portion of the hatched border. You will see the cursor change to crossed arrows when in position to move the Bean.

Inspecting SimpleBean Properties and Events

The Properties sheet displays the selected Bean's properties. With `SimpleBean` selected, the Properties sheet displays four propeties: `foreground`, `background`, `font`, and `name`. We declared no properties in SimpleBean (you will see how later), so these are properties inherited from Canvas. Clicking on each property brings up a property editor. The BeanBox provides default property editors for the primitive types, plus font and color types. You can find the sources for these property editors in `beans/apis/sun/beans/editors`.

Beans communicate with other Beans by sending and receiving event notifications. To see which events SimpleBean can send, choose the Edit|Events BeanBox menu item. A list of events, grouped by interface, will be displayed. Under each interface group is a list of event methods. These are all inherited from Canvas.

You will learn more about properties and events in upcoming sections.

Generating Bean Introspection Reports

Introspection is the process of discovering a Bean's design−time features by one of two methods:

- Low−level reflection, which uses design patterns to discover your Bean's features
- By examining an associated *bean information* class that explicitly describes your Bean's features.

You can generate a Bean introspection report by choosing the the Edit|Report menu item. The report lists Bean events, properties, and methods, and their characteristics.

By default Bean reports are sent to the java interpreter's standard output, which is the window where you started the BeanBox. You can redirect the report to a file by changing the java interpreter command in beanbox/run.sh or run.bat to:

```
java sun.beanbox.BeanBoxFrame > beanreport.txt
```

SimpleBean Makefiles

Here are two makefiles (Unix and Windows) set up to create SimpleBean. The makefiles compile the `.java` files, and create a JAR with the resulting `.class` file and a manifest.

```
# gnumake file

CLASSFILES= \
        sunw/demo/simple/SimpleBean.class

JARFILE= ../jars/SimpleBean.jar

all: $(JARFILE)

# Create a JAR file with a suitable manifest.
$(JARFILE): $(CLASSFILES) $(DATAFILES)
        echo "Name: sunw/demo/simple/SimpleBean.class" >> manifest.tmp
        echo "Java-Bean: True" >> manifest.tmp
        jar cfm $(JARFILE) manifest.tmp sunw/demo/simple/*.class
        @/bin/rm manifest.tmp

# Compile the sources
%.class: %.java
        export CLASSPATH; CLASSPATH=. ; \
        javac $<

# make clean
clean:
        /bin/rm -f sunw/demo/simple/*.class
        /bin/rm -f $(JARFILE)
```

Here is the Windows nmake version:

```
# nmake file
CLASSFILES= \
        sunw\demo\simple\simplebean.class \

JARFILE= ..\jars\simplebean.jar

all: $(JARFILE)

# Create a JAR file with a suitable manifest.

$(JARFILE): $(CLASSFILES) $(DATAFILES)
        jar cfm $(JARFILE) << manifest.tmp sunw\demo\simple\*.class
Name: sunw/demo/simple/SimpleBean.class
Java-Bean: True
<<

.SUFFIXES: .java .class

{sunw\demo\simple}.java{sunw\demo\simple}.class :
        set CLASSPATH=.
        javac $<

clean:
        -del sunw\demo\simple\*.class
        -del $(JARFILE)
```

You can use these makefiles as templates for creating your own Bean makefiles.

The JavaSoft website contains documentation on JAR files.

BeanBox Basics

 ***Table of Contents***

Writing a Simple Bean

In the following sections you will learn how to implement and use the basic Bean attributes: properties, events, and persistence. You will also learn how to use the BeanBox's applet generating capability.

Now is a good time to read or review the JavaBeans API Specification. Chapter 7 describes properties, Chapter 6 describes events, and Chapter 5 describes persistence.

- Implementing Properties explains how to give your Beans properties: Bean appearance and behavior characteristics customizable at design time.
- Manipulating Events in the BeanBox describes the BeanBox's event capabilities. You'll need a sound understanding of the JDK 1.1 event mechanism going in.
- Implementing Persistence describes how to make your Beans, and their customized state, saveable and restoreable.
- Using the BeanBox to Generate Applets shows you how to easily generate applets from the BeanBox contents.

 ***Table of Contents***

Implementing Properties

To get the most out of this section, first read Chapters 7, *Properties*, and Chapter 8, *Introspection*, of the JavaBeans API Specification.

If you create a Bean class, give it an instance variable named `color`, and access `color` through a *getter* method named `getColor` and a *setter* method named `setColor`, then you have created a property.

Properties are aspects of a Bean's appearance and behavior that are changeable at design time.

JavaBean properties follow specific rules, called *design patterns*, when naming getter and setter method names. This lets Beans−aware builder tools (and the BeanBox) discover, display (usually in a property sheet), and alter those properties at design time.

For example, a builder tool, in introspecting your Bean, discovers two methods, `getColor()` and `setColor()`, infers that a property named color exists, and displays that property in a property sheet where it can be edited.

Adding a Color Property to SimpleBean

Make the following changes to SimpleBean.java to add a color property:

1. Create and initialize a private instance variable.

```
private Color color = Color.green;
```

2. Write a getter method.

```
public Color getColor(){
  return color;
}
```

3. Write a setter method.

```
 public void setColor(Color newColor){
  color = newColor;
```

```
 repaint();
}
```

4. Override the inherited `paint()` method. This is a requirement for all subclasses of Canvas.

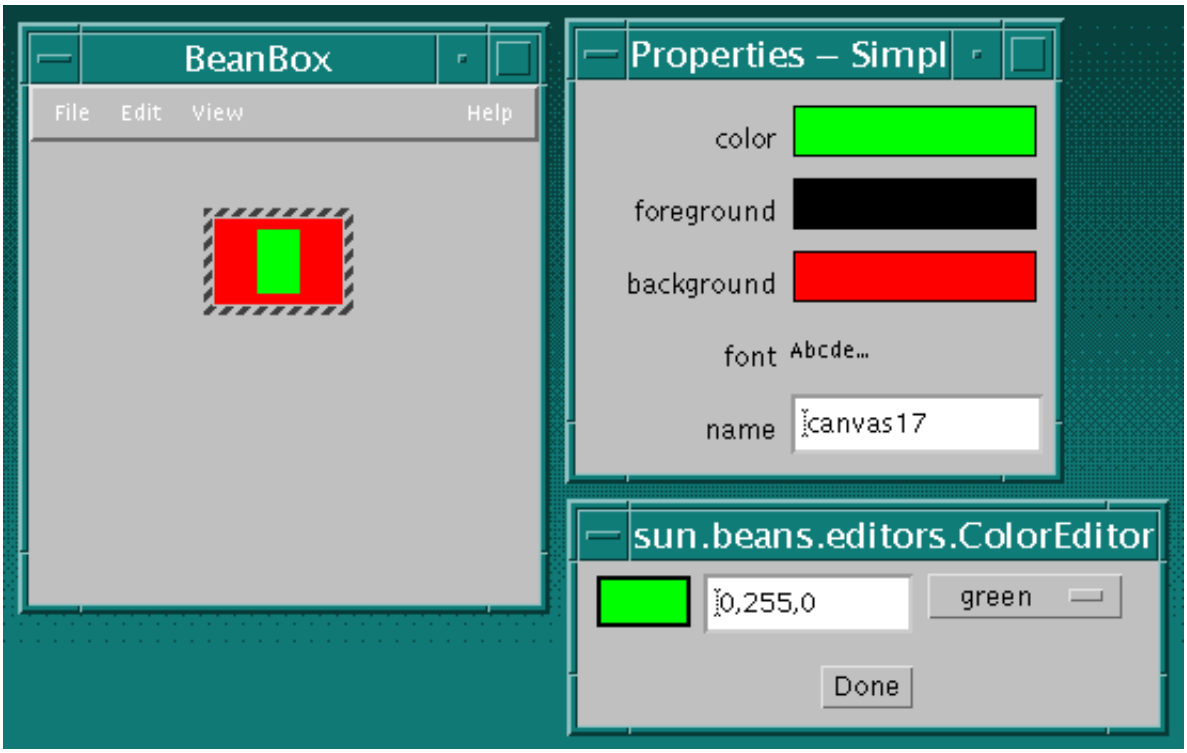
```
public void paint(Graphics g) {
 g.setColor(color);
 g.fillRect(20, 5, 20, 30);
}
```

5. Compile the Bean, load it in the ToolBox, and create an instance in the BeanBox.

   The results are:

- SimpleBean will be displayed with a green centered rectangle.
- The Properties sheet will contain a new color property. The introspection mechanism will also search for a color property editor. A Color property editor is one of the default editors supplied with the BeanBox. This editor is assigned as SimpleBean's color property editor. Click on the color property in the Properties sheet to run this editor. See section 9.2 of the JavaBeans API Specification for more information about property editors, particulary how the BeanBox searches for a property editor.

  Here is a BeanBox illustration showing the revised SimpleBean instance within the BeanBox, SimpleBean's new color property within the Properties sheet, and the Color property editor shipped with the BeanBox. Remember, clicking on the `color` property entry in the Properties sheet displays this editor.



You can change the color property by menu, or by RGB value. Try changing colors.

Here is the complete SimpleBean source code, revised to add a `color` property.

```
package sunw.demo.simple;

import java.awt.*;
import java.io.Serializable;

public class SimpleBean extends Canvas
                    implements Serializable{

  private Color color = Color.green;

  //property getter method
  public Color getColor(){
     return color;
  }

  //property setter method. Sets new SimpleBean
  //color and repaints.
  public void setColor(Color newColor){
     color = newColor;
     repaint();
  }

  public void paint(Graphics g) {
   g.setColor(color);
   g.fillRect(20, 5, 20, 30);
  }
```

```
  //Constructor sets inherited properties
  public SimpleBean(){
   setSize(60,40);
   setBackground(Color.red);
  }
}
```

You can learn how to implement bound, constrained, and indexed properties in the advanced Beans section of this document.

In the next lesson, you'll learn about events, and how to manipulate them in the BeanBox.

Writing a Simple Bean

Manipulating Events in the BeanBox

You'll need a good understanding of the JDK 1.1 event mechanism before reading this section. Here's the list of recommended readings:

- The Java AWT 1.1 event handling mechanism
- Chapter 6 of the JavaBeans API Specification
- Section 8.4, *Design Patterns for Events* of the JavaBeans API Specification.

Beans use the new event mechanism implemented in JDK 1.1, so implementing Bean events is the same as implementing events in any JDK 1.1 component. This section describes how this event mechanism is used by Beans and the BeanBox.

How the BeanBox Discovers a Beans Event Capabilities

The BeanBox uses either design pattern introspection or a `BeanInfo` class to discover what events a Bean can fire.

Using Introspection to Discover the Events A Bean Fires

JavaBeans provides event−oriented design patterns to give introspecting tools the ability to discover what events a Bean can fire. For a Bean to be the source of an event, it must implement methods that add and remove listener objects for that type of event. The design patterns for these methods are

```
public void add<EventListenerType>(<EventListenerType> a)
public void remove<EventListenerType>(<EventListenerType> a)
```

These methods let a source Bean know where to fire events. The source Bean then fires events at those listener Beans using the methods for those particular interfaces. For example, if a source Bean registers `ActionListener` objects, it will fire events at those objects by calling the `actionPerformed()` method on those listeners.

To see events discovered using design patterns, drop an instance of `OurButton` into the BeanBox and pull down the Edit|Events menu. This displays a list of event interfaces to which `OurButton` can fire events. Note that `OurButton` itself only adds and removes two of these interfaces; the rest are inherited from the base class.

Using BeanInfo to Define the Events a Bean Fires

You can explicitly "advertise" the events a Bean fires by using a class that implements the BeanInfo interface. The ExplicitButton demo Bean subclasses OurButton, and provides an associated ExplicitButtonBeanInfo class. ExplicitButtonBeanInfo implements the following method to explicitly define interfaces to which ExplicitButton fires events.

```
public EventSetDescriptor[] getEventSetDescriptors() {
    try {
        EventSetDescriptor push = new EventSetDescriptor(beanClass,
                    "actionPerformed",
                    java.awt.event.ActionListener.class,
                    "actionPerformed");

        EventSetDescriptor changed = new EventSetDescriptor(beanClass,
                    "propertyChange",
                    java.beans.PropertyChangeListener.class,
                    "propertyChange");

        push.setDisplayName("button push");
        changed.setDisplayName("bound property change");

        EventSetDescriptor[] rv = { push, changed};
        return rv;
    } catch (IntrospectionException e) {
        throw new Error(e.toString());
    }
}
```

Drop an ExplicitButton instance in the BeanBox, and pull down the Edit | Events menu. Notice that only those interfaces explicitly exposed in the ExplicitButtonBeanInfo class are listed. No inherited capabilities are exposed. See the Using BeanInfo section for more information on the BeanInfo interface.

Viewing a Bean's Events in the BeanBox

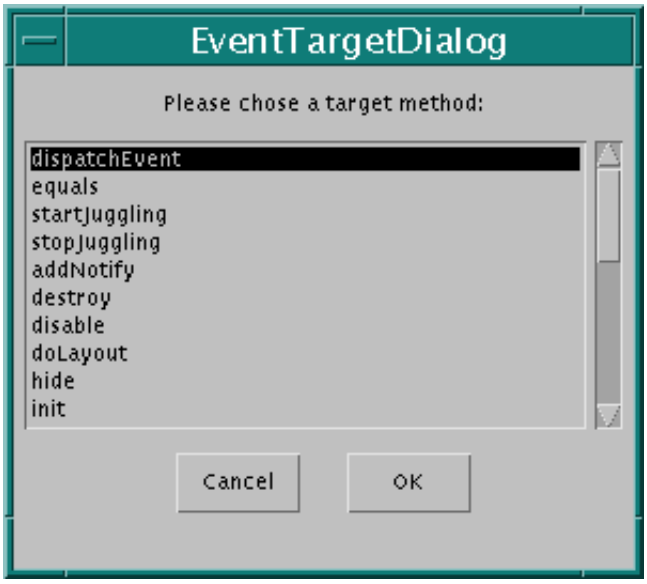
If you select an OurButton Bean in the BeanBox, then pull down the Edit | Events menu, you will see a list of interfaces that OurButton can fire events at. Each interface item will, when selected, display the methods that fire different events at those interfaces. These correspond to all the events that OurButton can fire.

Hooking Up Events in the BeanBox

In this example you will use two OurButton bean instances to stop and start an instance of the animated Juggler bean. You will label the buttons "start" and "stop"; make the start button, when pressed, invoke the Juggler bean's start method; and make the stop button, when pressed, invoke the Juggler bean's stop method.

1. Start the BeanBox.
2. Drop a Juggler bean and two OurButton bean instances into the BeanBox.

3. Select an OurButton instance. In the Properties sheet, change the label property to "start". Select the second OurButton instance and change its label to "stop".
4. Select the start button. Choose the Edit | Events | action | actionPerformed menu item. This causes a rubber band line to track between the start button and the cursor. Click on the Juggler instance. This brings up the EventTargetDialog:



   This list contains Juggler methods that take no arguments, or arguments of type actionPerformed.
5. Select the *start* method and press OK. You will see a message that the BeanBox is generating adapter classes.
6. Do the above two steps on the stop button, except choose the *stop* method in the EventTargetDialog.

   Clicking on the start and stop buttons will now start and stop the Juggler. Here is a general description of what happened:

- The start and stop buttons, are *event sources*. Event sources *fire events* at *event targets* . In this example the Juggler bean is the event target.
- When you select the start button and choose an event method (via the Edit | Event menu item), you are choosing the type of event the event source will fire.
- When you connect the rubber band line to another bean, you are selecting the event target Bean.
- The `EventTargetDialog` lists methods that can accept that type of event or that take no parameters. When you choose a method in the `EventTargetDialog`, you are specifying the method that will receive the fired event, and act on it.

   You will use this juggler example later when you learn how to generate applets, so use the File | Save menu item to save this example to a file of your choice.

Event Adapter Classes

The BeanBox generates an adapter class the interposes between the source and the target. The adapter class implements the appropriate event listener interface (and so is the actual listener, not the target Juggler Bean), catches the event fired by the button, and then calls the selected target method. Here is the BeanBox−generated adapter class that interposes between the start button and the `JugglerBean`:

```
// Automatically generated event hookup file.

package tmp.sunw.beanbox;
import sunw.demo.juggler.Juggler;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class ___Hookup_1474c0159e implements
           java.awt.event.ActionListener, java.io.Serializable {

    public void setTarget(sunw.demo.juggler.Juggler t) {
        target = t;
    }

    public void actionPerformed(java.awt.event.ActionEvent arg0) {
        target.startJuggling(arg0);
    }

    private sunw.demo.juggler.Juggler target;
}
```

The adapter implements the `ActionListener` interface that you selected in the BeanBox's Edit | Events menu. `ActionListener` declares one method, `actionPerformed()`, which is implemented by the adapter to call the target Bean method that you selected. The `setTarget()` method is called by the BeanBox to set the target Bean.

The `EventMonitor` Demo Bean

The `EventMonitor` Bean (`beans/demo/sunw/demo/encapsulatedEvents`) prints out source Bean event reports, as they occur, in a scrolling listbox. To see how this works, take the following steps:

1. Drop `OurButton` and `EventMonitor` instances in the BeanBox. You might want to resize the `EventMonitor` (and the BeanBox) to accommodate viewing the event reports.
2. Select the `OurButton` instance, and choose any event method in the Edit | Events menu.
3. Connect the rubber band line to the `EventMonitor`, and choose its `initiateEventSourcMonitoring` in the `EventTargetDialog`.
4. Select the `OurButton` Bean. You will begin seeing event reports in the `EventMonitor`

When the first event is delivered, `EventMonitor` analyzes the source Bean to discover all the events it fires, creates and registers an event listener for each event type, and then reports whenever any event is fired. This is useful for debugging. Try connecting

other demo Beans to `EventMonitor` to observer their events.

Events for Bound and Constrained Properties

The bound and constrained properties sections describe two specific event listener interfaces.

Writing a Simple Bean

 ***Writing a Simple Bean***

Implementing Persistence

> For complete information on persistence and serialization, see the Object Serialization web site.

A Bean persists by having its properties, fields, and state information saved and restored to and from storage. The mechanism that makes persistence possible is called *serialization*. When a Bean instance is serialized, it is converted into a data stream and written to storage. Any applet, application, or tool that uses that Bean can then "reconstitute" it by *deserialization*. JavaBeans uses the JDK's Object Serialization API for its serialization needs.

As long as one class in a class's inheritance hierarchy implements Serializable or Externalizable, that class is serializable.

All Beans must persist. To persist, your Beans must support serialization by implementing either the java.io.Serializable interface, or the java.io.Externalizable interface. These interfaces offer you the choice between automatic serialization, and "roll your own".

Controlling Serialization

You can control the level of serialization that your Beans undergo:

- Automatic: implement `Serializable`. Everything gets serialized.
- Selectively exclude fields you do not want serialized by marking with the `transient` (or `static`) modifier
- Complete control: implement `Externalizable`, and its two methods.

Default Serialization: The `Serializable` Interface

The `Serializable` interface provides automatic serialization by using the Java Object Serialization tools. `Serializable` declares no methods; it acts as a marker, telling the Object Serialization tools that your Bean class is serializable. Marking your class with `Serializable` means you are telling the JVM that you have made sure your class will work with default serialization. Here are some important points about working with the `Serializable` interface:

- Classes that implement `Serializable` must have a *no-argument constructor*. This constructor will be called when an object is "reconstituted" from a .ser file.

- There is no need to implement `Serializable` in your class if if it is already implemented in a superclass.
- All fields *but static and transient* are serialized. Use the `transient` modifier to specify fields you do not want serialized, and to specify classes that are not serializable.

The BeanBox writes serialized Beans to a file with a *.ser* extension.

The OurButton demo Bean uses default serialization to make its properties persist. OurButton only added `Serializable` to its class definition to make use of default serialization:

```
public class OurButton extends Component implements Serializable,...
```

If you drop an `OurButton` instance into the BeanBox, the properties sheet displays OurButton's properties. To ascertain that serialization is working

1. Change some OurButton properties. For example change the font size and colors.
2. Serialize the changed OurButton instance by selecting the File|SerializeComponent... BeanBox menu item. A file browser will pop up.
3. Put the .ser file in a JAR file with a suitable manifest (need to explain how to do this).
4. Clear the BeanBox form by selecting the File|Clear menu item.
5. Reload the serialized instance by selecting the File|LoadJar menu item.
   The `OurButton` instance will appear in the BeanBox with your property changes intact. By implementing `Serializable` in your class, simple, primitive properties and fields can be serialized. For more complex class members, different techniques must be used.

Selective Serialization Using the `transient` Keyword

To exclude fields from serialization in a `Serializable` object from serialization, mark the fields with the `transient` modifier.

```
transient int Status;
```

Default serialization will not serialize `transient` and `static` fields.

Selective Serialization: `writeObject` and `readObject()`

If your serializable class contains either of the following two methods (the signatures must be exact), then the default serialization will not take place.

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

You can control how more complex objects are serialized, by writing your own implementations of the `writeObject()` and `readObject()` methods. Implement

`writeObject` when you need to exercise greater control over what gets serialized, when you need to serialize objects that default serialization cannot handle, or when you need to add data to the serialization stream that is not an object data member. Implement `readObject()` to reconstruct the data stream you wrote with `writeObject()`.

Example: The `Molecule` Demo Bean

The `Molecule` demo keeps a version number in a static field. Since static fields are not serialized by default, `writeObject()` and `readObject()` are implemented to serialize this field. Here is the `writeObject()` and `readObject()` implementations in `Molecule.java`:

```
private void writeObject(java.io.ObjectOutputStream s)
                        throws java.io.IOException {
        s.writeInt(ourVersion);
        s.writeObject(moleculeName);
    }

private void readObject(java.io.ObjectInputStream s)
                        throws java.lang.ClassNotFoundException,
                               java.io.IOException {
        // Compensate for missing constructor.
        reset();
        if (s.readInt() != ourVersion) {
            throw new IOException("Molecule.readObject: version mismatch");
        }
        moleculeName = (String) s.readObject();
    }
```

These implementations limit the fields serialized to `ourVersion` and `moleculeName`. Any other data in the class will not be serialized.

It is best to use the `ObjectInputStream`'s `defaultWriteObject()` and `defaultReadObject` before doing your own specific stream writing. For example:

```
private void writeObject(java.io.ObjectOutputStream s)
                        throws java.io.IOException {
        //First write out defaults
        s.defaultWriteObject();
        //...
    }

private void readObject(java.io.ObjectInputStream s)
                        throws java.lang.ClassNotFoundException,
                               java.io.IOException {
        //First read in defaults
        s.defaultReadObject();
        //...
    }
```

Roll Your Own Serialization: The `Externalizable` Interface

Use the `Externalizable` interface to serialize (write) your Beans in a specific file format. `Externalizable` gives you complete flexibility in specifying the serialized format. You need to implement two methods: `readExternal()` and `writeExternal()`. `Externalizable` classes must also have a *no-argument constructor*.

Example: The `BlueButton` and `OrangeButton` Demo Beans

When you run the BeanBox, you will see two Beans named `BlueButton` and `OrangeButton` in the ToolBox. These two Beans are actually serialized instances of the `ExternalizableButton` class.

`ExternalizableButton` implements the `Externalizable` interface. This means it does all its own serialization, by implementing `Externalizable.readExternal()` and `Externalizable.writeExternal()`. The `BlueButtonWriter` program is used by the buttons makefile to create an `ExternalizableButton` instance, change its `background` property to blue, and write the Bean out to the file `BlueButton.ser`. `OrangeButton` is created the same way, using `OrangeButtonWriter`. The button makefile then puts these `.ser` files in `buttons.jar`, where the ToolBox can find and reconstitute them.

Tips on what should not be serialized

Here is a short list of things you don't want to serialize:

- Event adaptors
- References to other objects
- Images
- File handles

A good strategy is to mark fields or objects that cannot be made persistent using the default offered by the `Serialization` interface. Then write your own implementations of the `writeObject()` and `readObject()` methods to serialize and deserialize those transient fields.

Writing a Simple Bean

Using the BeanBox to Generate Applets

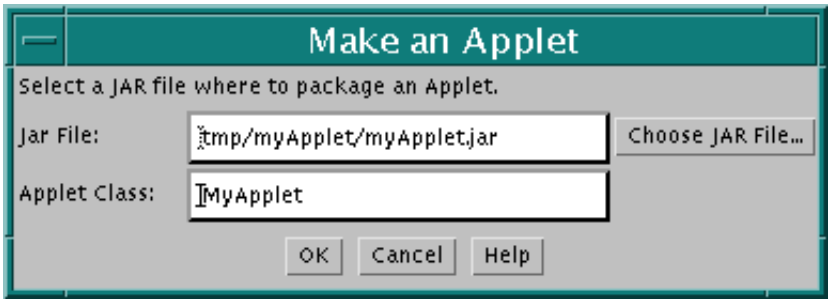
You can use the BeanBox's File|MakeApplet... menu item to generate an applet from the BeanBox contents. Making an applet from the BeanBox creates a JAR file containing

- Serialized data and class files
- A test HTML file that uses the JAR file (and any other JAR file needed)
- A subdirectory with Java sources and makefile
- A readme file with complete information about the generated applet and all files involved.

Take the following steps to generate an applet from the BeanBox:

1. Use the Juggler example that you made in the events section. If you saved that example to a file, load it in the BeanBox using the File|Load menu item. If you didn't save it, follow the steps in the events section to build the example. The generated applet will have the same size as the BeanBox frame, so you might want to adjust the BeanBox size to the size of the applet you want.

2. Choose File|Make Applet to bring up the MakeApplet dialog:

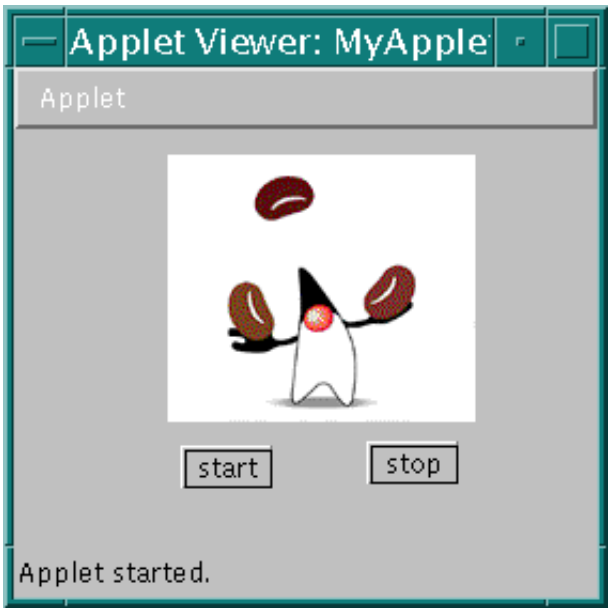


Use the default JAR file and applet name for this example. Press the OK button.

That's it. The generated files were placed in the `beanbox/tmp/myApplet` directory. You can inspect your handiwork by bringing up appletviewer in the following way:

```
appletviewer <BDKInstallation>/beanbox/tmp/myApplet.html.
```

Here is what you will see:



Don't forget to look at the generated myApplet_readme file, and the other files generated.

Generated applets can be used in any JDK 1.1−compliant browser. The appletviewer is a good test platform. Another fully compliant browser is the HotJava browser The preview2 Internet Explorer 4.0 release does not yet support JAR files, and you will have to expand the JAR and HTML files that are generated. Also, a deserialization bug causes components to not listen to mouse events. See the generated readme file for more information. The generated applet will not work in Netscape Communicator versions 4.0 and 4.01; versions with full JDK 1.1 support are expected later this year.

Writing a Simple Bean


Table of Contents

Writing Advanced Beans

This section contains lessons that illustrate more complex bean capabilities.

- Implementing Bound Properties describes how to cause a property change in one Bean to affect other Beans.
- Implementing Constrained Properties tells you how to let other outside Beans exert veto power on your Bean's property changes.
- Implementing Indexed Properties introduces properties that are best represented as arrays.
- Customizing Beans introduces you to property editors and the `Customizer` interface.
- Using BeanInfo describes the details of Bean information classes: Separate classes used to provide Bean information to builder tools.
- Converting Beans into ActiveX Components


Table of Contents

Implementing Bound Properties

Here is a list of background reading to prepare you for learning bound properties:

- Chapter 7 of the JavaBeans API Specification
- `PropertyChangeListener` interface
- `PropertyChangeSupport` class
- `PropertyChangeEvent` class

The Beans API provides an event mechanism for alerting *listeners* when Bean properties change. Listeners can be other objects, Beans, components, applications, applets or scripting environments.

There are three parts to a bound property implementation:

- A *source* Bean that contains the bound properties.
- *Listener* objects that register their interest in receiving property change events from a source Bean.
- Property change events that are fired *from* source Beans *to* interested listeners.

There is nothing new here; these are just specific interfaces and classes based on the AWT1.1 event mechanism.

Listeners must implement the `PropertyChangeListener` interface. Implementing this interface means that

- The listener can be added to the list maintained by the property change event source, and
- That the listener implements the `propertyChange()` method.

Beans that are the source of property change events must maintain a `PropertyChangeListener` list, and fire events at each object in that list when a bound property changes. The utility class `PropertyChangeSupport` implements this functionality. Your Bean can inherit from this class, or use it as an inner class, as you will see.

The `PropertyChangeEvent` class encapsulates property change information, and is sent from the property change event source to each object in the property change listener list.

Implementing Bound Property Support Within a Source Bean

A Bean containing bound properties must

- Allow listeners to register and unregister their interest in receiving property change events.
- Fire property change events at interested listeners when a bound property changes.

The `java.beans` package provides a utility class, named `PropertyChangeSupport`, which implements two methods that add and remove `PropertyChangeListener` objects to a listener list, and a method that fires property change events at each listener in that list when a bound property changes. Your Bean can either inherit from `PropertyChangeSupport`, or use it as an inner class.

To give your source Bean the ability to broadcast property changes events, take the following steps:

1. **Import the** `java.beans` **package**. This gives you access to the `PropertyChangeSupport` class.

2. **Instantiate a PropertyChangeSupport object**:

```
private PropertyChangeSupport changes = new PropertyChangeSupport(this);
```

This object manages the registered listener list, and property change event notification broadcast.

3. **Implement methods to maintain the property change listener list**. Since we instantiated a `PropertyChangeSupport` object in this example (rather inheriting from it), these methods merely wrap calls to the property−change support object's methods:

```
public void addPropertyChangeListener(PropertyChangeListener l) {
    changes.addPropertyChangeListener(l);
   }
public void removePropertyChangeListener(PropertyChangeListener l) {
    changes.removePropertyChangeListener(l);
   }
```

4. **Modify a property's setter method to fire a property change event when the property is changed**. `OurButton`'s `setLabel()` method looks like this:

```
public void setLabel(String newLabel) {
    String oldLabel = label;
    label = newLabel;
    sizeToFit();
    changes.firePropertyChange("label", oldLabel, newLabel);
  }
```

Note that `setLabel()` stores the old `label` value, because both the old and new labels must be passed to `firePropertyChange()`.

```
public void firePropertyChange(String propertyName,
                               Object oldValue, Object newValue)
```

These values are then bundled into the `PropertyChangeEvent` object sent to each registered listener. The old and new values are treated as `Object` values, so if they are primitive types such as `int`, you must use the object version such as `java.lang.Integer`. Also note that the property change events are fired *after* the property has changed.

When the BeanBox recognizes the design patterns for bound properties within your Bean, you will see a `propertyChange` interface item when you pull down the Edit | Events menu.

Now that you have given your Bean the ability to broadcast events when a bound property has changed, the next step is to create a listener.

Implementing Bound Property Listeners

To listen for property change events, your listener Bean must implement the `PropertyChangeListener` interface. This interface contains one method:

```
public abstract void propertyChange(PropertyChangeEvent evt)
```

Implementing the `PropertyChangeListener` interface in your class means implementing `propertyChange()`.

So to make your class able to listen and respond to property change events, you must

1. **Implement the** `PropertyChangeListener` **interface**.

```
public class MyClass implements java.beans.PropertyChangeListener,
                                                java.io.Serializable {
```

This interface contains one method, `propertyChange()`. This is the method that the property change event source calls when one of its bound propertys changes.

2. **Implement the** `propertyChange()` **method in the listener**. This method needs to contain the code that handles what you need to do when the listener receives property change event. Very often this is a call to a setter method in the listener class: a property change in the source bean propagates a change to a property in the listener.

To register interest in receiving notification about a Bean property change, call the listener registration method on the source Bean. For example:

```
button.addPropertyChangeListener(aButtonListener);
```

Or, you can use an adapter class to catch the property change event, and subsequently call the correct method within the listener object. Here is an example taken from comments in the `beans/demo/sunw/demo/misc/ChangeReporter.java` file.

```
OurButton button = new OurButton();
...
PropertyChangeAdapter adapter = new PropertyChangeAdapter();
...
button.addPropertyChangeListener(adapter);
...
class PropertyChangeAdapter implements PropertyChangeListener
{
  public void propertyChange(PropertyChangeEvent e)
  {
    reporter.reportChange(e);
  }
}
```

Bound Properties in the BeanBox

The BeanBox handles bound properties by using an event hookup adapter class. The `OurButton` and `ChangeReporter` Beans can be used to illustrate this technique. To see how this works, take the following steps:

1. Drop `OurButton` and `ChangeReporter` instances on the BeanBox.
2. Select the `OurButton` instance and choose the Edit | Events | propertyChange | propertyChange menu item.
3. Connect the rubber band line to the ChangeReporter instance. The EventTargetDialog will be displayed.
4. Choose `reportChange` from the EventTargetDialog. The event hookup adapter source will be generated and compiled
5. Select OurButton and change some of it's properties. You will see change reports in ChangeReporter.

Behind the scenes the BeanBox generated the event hookup adapter. This adapter implements the `PropertyChangeListener` interface, and also generates a `propertyChange()` method implementation that calls the `ChangeReporter.reportChange()` method. Here's the generated adapter source code:

```
// Automatically generated event hookup file.

package tmp.sunw.beanbox;
import sunw.demo.misc.ChangeReporter;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeEvent;

public class ___Hookup_14636f1560 implements
            java.beans.PropertyChangeListener, java.io.Serializable {

    public void setTarget(sunw.demo.misc.ChangeReporter t) {
        target = t;
    }
```

```
    public void propertyChange(java.beans.PropertyChangeEvent arg0) {
        target.reportChange(arg0);
    }

    private sunw.demo.misc.ChangeReporter target;
}
```

The `ChangeReporter` Bean need not implement the `PropertyChangeListener` interface; instead, the BeanBox generated adapter class implements `PropertyChangeListener`, and the adapter's `propertyChange()` method calls the appropriate method in the target object (`ChangeReporter`).

The BeanBox puts the event adapter classes in the `beans/beanbox/tmp/sunw/beanbox` directory. When an adapter class is generated, you can view the adapter source in that directory.

Writing Advanced Beans

Implementing Constrained Properties

To get the most out of this section, first read the following:

- Chapter 7 of the JavaBeans API Specification
- `VetoableChangeListener` interface
- `VetoableChangeSupport` class
- `PropertyChangeEvent` class
- `PropertyVetoException` class

A Bean property is constrained when any change to that property can be vetoed. Usually an outside object exercises the right to veto, but the Bean itself can also veto a property change.

The Beans API provides an event mechanism, very similar to the bound property mechanism, that allows objects to veto a Bean's property changes.

There are three parts to constrained property implementations:

- A *source* Bean containing one or more constrained properties.
- *Listener* objects that implement the `VetoableChangeListener` interface. This object accepts or rejects proposed changes to a constrained property in the source Bean.
- A `PropertyChangeEvent` object containing the property name, and its old and new values. This is the same class used for bound properties.

Implementing Constrained Property Support Within a Source Bean

A Bean containing constrained properties must

- Allow `VetoableChangeListener` objects to register and unregister their interest in receiving notification that a property change is proposed.
- Fire property change events at those interested listeners when a property change is proposed. The event should be fired *before* the actual property change takes place. This gives each listener a chance to veto the proposed change. The `PropertyChangeEvent` is fired by a call to each listeners `vetoableChange()` method.
- If a listener vetoes, then make sure that any other listeners can revert to the old value. This means reissuing the `vetoableChange()` call to all the listeners, with a `PropertyChangeEvent` containing the old value.

The `VetoableChangeSupport` utility class is provided to implement these

capabilities. This class implements methods to add and remove `VetoableChangeListener` objects to a listener list, and a method that fires property change events at each listener in that list when a property change is proposed. This method will also catch any vetoes, and resend the property change event with the original property value. Your Bean can either inherit from `VetoableChangeSupport`, or use an instance of it.

Note that, in general, constrained properties should also be bound properties. When a constrained property change does occur, a `PropertyChangeEvent` can be sent via `PropertyChangeListener.propertyChange()` to signal all `VetoableChangeListener` Beans that the change has taken effect.

The `JellyBean` demo Bean has a constrained property. We will its code to illustrate the steps in implementing constrained properties. Here's the steps to implement constrained properties in your Beans:

1. **Import the `java.beans` package**. This gives you access to the `VetoableChangeSupport` class.

2. **Instantiate a `VetoableChangeSupport` object within your Bean**:

```
private VetoableChangeSupport vetos =
          new VetoableChangeSupport(this);
```

`VetoableChangeSupport` manages a list of `VetoableChangeListener` objects, and fires property change events at each object in the list when a change occurs to a constrained property.

3. **Implement methods to maintain the property−change listener list**. These merely wrap calls to the `VetoableChangeSupport` object's methods:

```
public void addVetoableChangeListener(VetoableChangeListener l) {
 vetos.addVetoableChangeListener(l);
}
public void removeVetoableChangeListener(VetoableChangeListener l) {
 vetos.removeVetoableChangeListener(l);
}
```

4. **Write a property's setter method to fire a property change event when the property is changed. This includes adding a `throws` clause to the setter method's signature**. JellyBean's `setPriceInCents()` method looks like this:

```
public void setPriceInCents(int newPriceInCents)
                      throws PropertyVetoException {
  int oldPriceInCents = ourPriceInCents;

  // First tell the vetoers about the change.  If anyone objects, we
  // don't catch the exception but just let if pass on to our caller.
  vetos.fireVetoableChange("priceInCents",
                           new Integer(oldPriceInCents),
                           new Integer(newPriceInCents));
```

```
        // No-one vetoed, so go ahead and make the change.
        ourPriceInCents = newPriceInCents;
        changes.firePropertyChange("priceInCents",
                                   new Integer(oldPriceInCents),
                                   new Integer(newPriceInCents));
    }
```

Note that `setPriceInCents()` stores the old `price`, because both the old and new prices must be passed to `fireVetoableChange()`. Also note that the primitive `int` prices are converted to `Integer` objects.

```
public void fireVetoableChange(String propertyName,
                               Object oldValue,
                               Object newValue)
                        throws PropertyVetoException
```

These values are then bundled into a `PropertyChangeEvent` object sent to each listener. The old and new values are treated as `Object` values, so if they are primitive types such as `int`, you must use the object version such as `java.lang.Integer`.

Now you need to implement a Bean that listens for constrained property changes.

Implementing Constrained Property Listeners

To listen for property change events, your listener Bean must implement the `VetoableChangeListener` interface. This interface contains one method:

```
void vetoableChange(PropertyChangeEvent evt)
                        throws PropertyVetoException;
```

So to make your class able to listen and respond to property change events, your listener class must

1. **Implement the** `VetoableChangeListener` **interface**.

2. **Implement the** `vetoableChange()` **method**. This is the method that will be called by the source Bean on each object in the listener list (maintained by the `VetoableChangeSupport` object). This is also the method that exercises veto power. A property change is vetoed by throwing the `PropertyVetoException`.

Note that the `VetoableChangeListener` object is often an adapter class. The adapter class implements the `VetoableChangeListener` interface and the `vetoableChange()` method. This adapter is added to the constrained Bean's listener list, intercepts the `vetoableChange()` call, and calls the target Bean method that exercises veto power. You'll see an example of this in the next section.

Constrained Properties in the BeanBox

When the BeanBox recognizes the design patterns for constrained properties within your Bean, you will see a `vetoableChange` interface item when you pull down the Edit | Events menu.

As with any event hookup, the BeanBox generates an adapter class when you hook up a Bean with a constrained property to another Bean. To see how this works, take the following steps:

1. Drop `Voter` and `JellyBean` instances into the BeanBox.
2. Select the `JellyBean` instance, and choose the Edit | Events | vetoableChange | vetoableChange menu item.
3. Connect the rubber band line to the `Voter` Bean. This brings up the `EventTargetDialog` panel.
4. Choose the `Voter` Bean's `vetoableChange` method, and push the OK button. This generates an event adapter. You can view this adapter in the `beans/beanbox/tmp/sunw/beanbox` directory.
5. Test the constrained property. Select the `JellyBean` and edit its `priceInCents` property in the Properties sheet. A `PropertyVetoException` is thrown, and an error dialog pops up.

Behind the scenes the BeanBox generated the event hookup adapter. This adapter implements the `VetoableChangeListener` interface, and also generates a `vetoableChange()` method implementation that calls the `Voter.vetoableChange()` method. Here's the generated adapter source code:

```
// Automatically generated event hookup file.

package tmp.sunw.beanbox;
import sunw.demo.misc.Voter;
import java.beans.VetoableChangeListener;
import java.beans.PropertyChangeEvent;

public class ___Hookup_1475dd3cb5 implements
        java.beans.VetoableChangeListener, java.io.Serializable {

    public void setTarget(sunw.demo.misc.Voter t) {
        target = t;
    }

    public void vetoableChange(java.beans.PropertyChangeEvent arg0)
                            throws java.beans.PropertyVeto Exception {
        target.vetoableChange(arg0);
    }

    private sunw.demo.misc.Voter target;
}
```

The `Voter` Bean need not implement the `VetoableChangeListener` interface; instead, the generated adapter class implements `VetoableChangeListener`. The adapter's `vetoableChange()` method calls the appropriate method in the target

object (`Voter`).

Per Property Constraint

Like bound property support, there is design pattern support for adding and removing `VetoableChangeListener` objects that are tied to a specific named property:

```
void addVetoableChangeListener(String propertyName,
                        VetoableChangeListener listener);
void removeVetoableChangeListener(String propertyName,
                        VetoableChangeListener listener);
```

As an alternative, for each constrained property a Bean can provide methods with the following signature to register and unregister vetoable change listeners on a per property basis:

```
void add<PropertyName>Listener(VetoableChangeListener p);
void remove<PropertyName>Listener(VetoableChangeListener p);
```

Writing Advanced Beans

Implementing Indexed Properties

To get the most out of this section, read the following documentation:

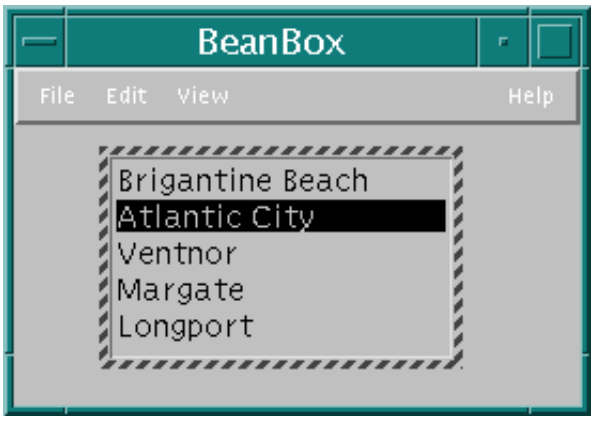
- Section 7.2 of the JavaBeans API Specification.
- IndexedPropertyDescriptor class
- PropertyEditor interface
- PropertyEditorSupport class

Indexed properties represent collections of values accessed, like an array, by index. The indexed property design patterns are

```
//Methods to access the entire indexed property array
public <PropertyType>[] get();
public void set<PropertyName>([] value);

//Methods to access individual values
public <PropertyType> get(int index);
public void set<PropertyName>(int index,  value);
```

Conforming to these patterns lets builder tools know that your Bean contains an indexed property.

The `OurListBox` demo Bean illustrates how to use an indexed property. OurListBox extends the `List` class to provide a Bean that presents the user a list of choices: Choices that you can provide and change at design time. Here's an illustration of an `OurListBox` instance:



`OurListBox` exposes the `item` indexed property with the following accessor methods:

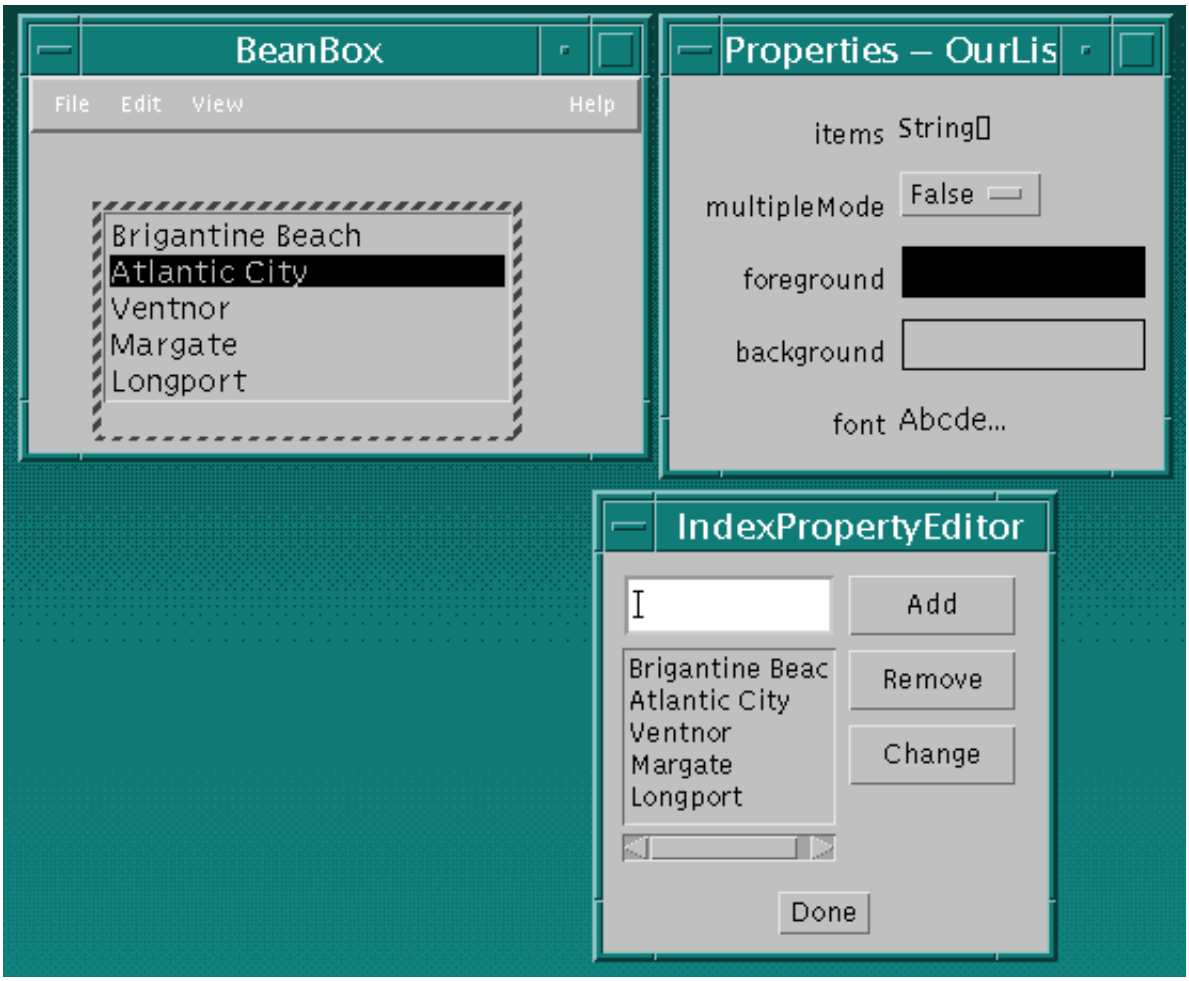
```
public void setItems(String[] indexprop) {
  String[] oldValue=fieldIndexprop;
  fieldIndexprop=indexprop;
  populateListBox();
  support.firePropertyChange("items",oldValue, indexprop);
}

public void setItems(int index, String indexprop) {
  String[] oldValue=fieldIndexprop;
  fieldIndexprop[index]=indexprop;
  populateListBox();
  support.firePropertyChange("Items",oldValue, fieldIndexprop);
}

public String[] getItems() {
  return fieldIndexprop;
}

public String getItems(int index) {
  return getItems()[index];
}
```

When an item is set by one of the `setItems()` methods, `OurListBox` is populated with the contents of a `String` array.

Indexed properties are almost as easily exposed as simple properties. Writing an indexed property editor, though, requires writing a custom property editor.

Indexed Property Editors

The `OurListBox` demo Bean provides an associated `IndexPropertyEditor` which is a good example of how to implement an indexed property editor. The following illustration shows an `OurListBox` instance in the BeanBox, the Properties sheet which contains an entry for the indexed property `items`, and the `IndexPropertyEditor` which pops up when the `items` property entry is clicked:



Implementing `IndexPropertyEditor` is the same as implementing any custom property editor:

1. Implement the `PropertyEditor` interface:

```
public class IndexPropertyEditor extends Panel
             implements PropertyEditor, Action Listener {
```

You can use the `PropertyEditorSupport` class, either by subclassing or as an inner class.

2. Denote the custom editor in a related BeanInfo class. `OurListBox` has a related `OurListBoxBeanInfo` class that contains the following code:

```
itemsprop.setPropertyEditorClass(IndexPropertyEditor.class);
```

3. Make the property editor a source for bound property events. The property editor will register property listeners, and fire property change events at those listeners. This is how the property changes are propagated back to the Bean (via the property sheet). So `IndexPropertyEditor` instantiates an inner `PropertyChangeSupport` class:

```
    private PropertyChangeSupport support =
         new PropertyChangeSupport(this);
```

Provides the ability for objects to register their interest in being notified when a property is edited:

```
public void addPropertyChangeListener(PropertyChangeListener l) {
 support.addPropertyChangeListener(l);
}

public void removePropertyChangeListener(PropertyChangeListener l) {
 support.removePropertyChangeListener(l);
}
```

And fires property change events at those listeners:

```
   public void actionPerformed(ActionEvent evt) {
 if (evt.getSource() == addButton) {
     listBox.addItem(textBox.getText());
     textBox.setText("");
     support.firePropertyChange("", null, null);
 }
 else if (evt.getSource()== textBox) {
     listBox.addItem(textBox.getText());
     textBox.setText("");
     support.firePropertyChange("",null,null);
 }
 ...
}
```

`IndexPropertyEditor` maintains `listbox` as a proxy for OurListBox. When a change is made to `listbox`, a property change event is fired to all listeners.

When the Properties sheet, which is registered as an `IndexPropertyEditor` listener, receives a property change event from `IndexPropertyEditor`, the Properties sheet calls `IndexPropertyEditor.getValue()` to retrieve the new or changed items and update the Bean.

Writing Advanced Beans

Customizing Beans

To prepare yourself for learing about property editors and customizers, read the following documentation:

- `PropertyEditor` api interface
- `PropertyEditorSupport` api class
- `PropertyEditorManager` api class
- `Customizer` api interface
- `BeanInfo` api interface

A Bean's appearance and behavior can be customized at design time within Beans−compliant builder tools. Typically there are two ways to customize a Bean:

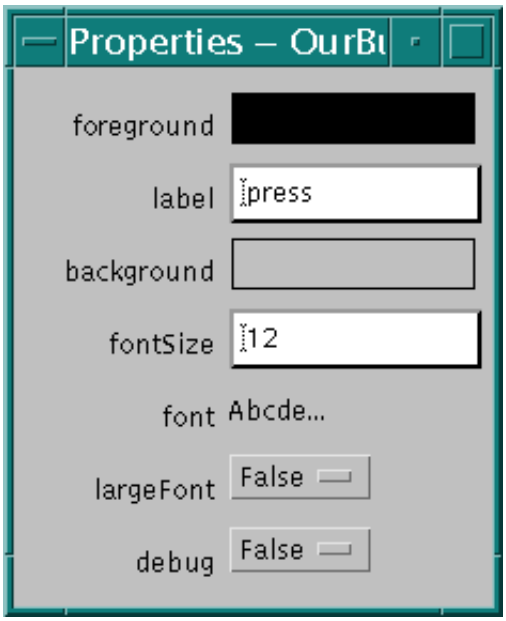
- By using a *property editor*. Each Bean property has its own property editor. A builder tool usually displays a Bean's property editors in a *property sheet*. A property editor is associated with, and edits a particular property type.
- By using *customizers*. Customizers give you complete GUI control over Bean customization. Customizers are used where property editors are not practical or applicable. Unlike a property editor, which is associated with a property, a customizer is associated with a Bean.

Property Editors

A property editor is a tool for customizing a particular property *type*. Property editors are typically displayed in, or activated from property sheets. A property sheet will determine a property's type, look around for a relevant property editor, and display the property's current value in a relevant way.

Property editors must implement the `PropertyEditor` interface. `PropertyEditor` provides methods that specify how a property should be displayed in a property sheet.

Here is the BeanBox's Properties sheet containing `OurButton` properties:



You begin the process of editing these properties by clicking on the property entry in the sheet.

- The `label` and `fontSize` properties are displayed in an editable text box. Changes can be made in place.
- The `largeFont` and `debug` properties are selection boxes with discrete choices.
- Clicking on the `foreground`, `background`, and `font` entries brings up separate panels.

How each of these is displayed depends on which `PropertyEditor` methods you implement to return non−null (or equivalent) values.

For example, the `int` property editor implements the `setAsText()` method. This indicates to the property sheet that the property can be displayed as a `String`, hence an editable text box will do.

The `Color` and `Font` property editors use a separate panel, and merely use the property sheet to display the current property value. The editor is displayed by clicking on that value. To display the current property value "sample" within the property sheet you need to override `isPaintable()` to return `true`, and override `paintValue()` to paint the current property value on a rectangle in the property sheet. Here's how `ColorEditor` implements `paintValue()`:

```
public void paintValue(java.awt.Graphics gfx, java.awt.Rectangle box) {
        Color oldColor = gfx.getColor();
        gfx.setColor(Color.black);
        gfx.drawRect(box.x, box.y, box.width−3, box.height−3);
        gfx.setColor(color);
        gfx.fillRect(box.x+1, box.y+1, box.width−4, box.height−4);
        gfx.setColor(oldColor);
}
```

To support the custom property editor, you need to override two more methods: Override `supportsCustomEditor()` to return true, and override `getCustomEditor()` to return a custom editor instance. `ColorEditor.getCustomEditor()` returns `this`.

Additionally, the `PropertyEditorSupport` class maintains a `PropertyChangeListener` list, and fires property change event notifications to those listeners when a bound property is changed. For example

How Property Editors are Associated with Properties

Property editors are discovered and associated with a given property by

- Explicit association via a `BeanInfo` object.

The `Molecule` demo Bean uses this technique. Within the `MoleculeBeanInfo` class, the `Molecule` Bean's property editor is set with the following line of code:

```
pd.setPropertyEditorClass(MoleculeNameEditor.class);
```

- Explicit registration via `java.Beans.PropertyEditorManager.registerEditor()`. This method takes a pair of arguments: The class type, and the editor to be associated with that.

- Name search. If a class has no explicitly associated property editor, then the `PropertyEditorManager` searchs for a that class's property editor by:
  - Appending "Editor" to the fully qualified class name. For example, for the `java.beans.ComplexNumber` class, the property editor manager would search for the `java.beans.ComplexNumberEditor` class.
  - Appending "Editor" to the class name and searching a class search path. The default class path for the BeanBox is `sun.beans.editors`.

The BDK Property Editors

The BDK provides property editors for the primitive data types like `intbool`, and `float`, and `Color` and `Font` class types. The source code for these property editors is in `beans/apis/sun/beans/editors`. These sources make a good starting point for writing your own property editors. Some things to note about the BDK property editors:

- All the "number" properties are represented as `String` objects. The `IntEditor` overrides `PropertyEditorSupport.setAsText()`.
- The `bool` property editor is a menu of discrete choices. By overriding the `PropertyEditorSupport.getTags()` method to return a `String[]` containing "True" and "False":

```
public String[] getTags() {
   String result[] = { "True", "False" };
   return result;
```

```
}
```

- The `Color` and `Font` property editors implement custom property editors. Because these objects require a more sophisticated interface to be easily edited a separate component pops up to do the property editing. Overriding `supportsCustomEditor()` to return true signals the property sheet that this property's editor is a custom component. The `isPaintable()` and `paintValue()` methods are also overridden to provide color and font painting in the editors property sheet sample areas.

  The source code for these property editors is in `beans/apis/sun/beans/editors`.

  Note that if no property editor is found for a property, the BeanBox will not display that property in the Properties sheet.

Customizers

When you use a Bean *Customizer*, you get complete control over how to configure or edit a Bean. A Customizer is like an application that specifically targets a Bean's customization. Sometimes properties are insufficient for representing a Bean's configurable attributes. Customizers are used where sophisticated instructions would be needed to change a Bean, and where property editors are too primitive to achieve Bean customization.

All customizers must:

- Extend `java.awt.Component` or one of its subclasses.
- Implement the `java.beans.Customizer` interface This means implementing methods to register `PropertyChangeListener` objects, and firing property change events at those listeners when a change to the target Bean has occurred.
- Implement a default constructor.
- Associate the customizer with its target class via `BeanInfo.getBeanDescriptor()`.

If a Bean that has an associated Customizer is dropped into the BeanBox, you will notice a "Customize..." item on the Edit menu.

BDK Customizers

UNDER CONSTRUCTION

The `OurButtonCustomizer` serves as an example that demonstrates the mechanics of building a customizer. `OurButtonCustomizer`:

- Extends `java.awt.Panel` (a `Component` subclass).
- Implements the `Customizer` interface, and uses a `PropertyChangeSupport` object to manage `PropertyChangeListener` registration and notification. See the bound property section for a `PropertyChangeSupport` description.
- Implements a default constructor:

```
public OurButtonCustomizer() {
   setLayout(null);
}
```

- Is associated with its target class, `ExplicitButton`, by `ExplicitButtonBeanInfo`. Here's the `ExplicitButtonBeanInfo` statements that do the association:

```
public BeanDescriptor getBeanDescriptor() {
   return new BeanDescriptor(beanClass, customizerClass);
}
...
private final static Class customizerClass =
                          OurButtonCustomizer.class;
```

The `BridgeTester` and `JDBC Select` demo Beans also have customizers.

Writing Advanced Beans

Using BeanInfo

The following documentation will help you learn about the `BeanInfo` class:

- BeanInfo interface
- class
- Introspector class
- FeatureDescriptor class
- BeanDescriptor class
- EventSetDescriptor class
- PropertyDescriptor class
- IndexedPropertyDescriptor class
- MethodDescriptor class
- ParameterDescriptor class
- Java Core Reflection API
- JavaBeans API Specification section 8.6

A Bean can *implicitly* expose its properties, events, and methods by conforming to design patterns which are recognized by low−level reflection. Alternatively, a Bean can *explicitly* expose features in a separate, associated class that implements the `BeanInfo` interface. You can use the `BeanInfo` interface (or its convenience class `SimpleBeanInfo`) to explicitly expose design time information for Beans aware builder tools.

By using an associated `BeanInfo` class you can

- Expose only those features you want to expose.
- Rely on `BeanInfo` to expose some Bean features while relying on low−level reflection to expose others.
- Associate an icon with the target Bean.
- Specify a customizer class.
- Segregate features into normal and expert categories.
- Provide a more descriptive display name, or additional information about a Bean feature.

`BeanInfo` defines methods that return descriptors for each property, method, or event that is to be exposed in a builder tool. Here's the prototypes for these methods:

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

Each of these methods returns an array of *descriptors* for each item that is to be exposed in a builder tool.

Creating a `BeanInfo` Class

We'll use the `ExplicitButtonBeanInfo` to illustrate `BeanInfo` class creation. Here are the general steps to make a `BeanInfo` class:

1. Name your `BeanInfo` class. You must append the string "BeanInfo" to the target class name. If the target class name is `ExplicitButton`, then its associated Bean information class must be named `ExplicitButtonBeanInfo`
2. Subclass the `SimpleBeanInfo` class. This is a convenience class the implements `BeanInfo` methods to return null, or an equivalent noop value.

```
public class ExplicitButtonBeanInfo extends SimpleBeanInfo {
```

Using this class saves you from implementing all the `BeanInfo` methods; you only have to override those methods you need.

3.
4. Override the methods you need to specify and return the properties, methods, or events that you want. `ExplicitButtonBeanInfo` overrides the `getPropertyDescriptors()` method to return four properties:

```
public PropertyDescriptor[] getPropertyDescriptors() {
 try {
  PropertyDescriptor background =
    new PropertyDescriptor("background", beanClass);
  PropertyDescriptor foreground =
    new PropertyDescriptor("foreground", beanClass);
  PropertyDescriptor font =
    new PropertyDescriptor("font", beanClass);
  PropertyDescriptor label =
    new PropertyDescriptor("label", beanClass);

  background.setBound(true);
  foreground.setBound(true);
  font.setBound(true);
  label.setBound(true);

  PropertyDescriptor rv[] =
    {background, foreground, font, label};
  return rv;
 } catch (IntrospectionException e) {
    throw new Error(e.toString());
   }
}
```

There are two important things to note here:

- If you leave a descriptor out, that property, event or method not described will not be exposed. In other words, you can selectively expose properties, events, or methods by leaving out those you don't want exposed.
- If a feature's getter method returns null, low−level reflection is used for that

feature. So you can explicitly specify properties, for example, and let low−level reflection discover the methods. If you don't override the `SimpleBeanInfo` default method, which returns null, low−level reflection will be used for that feature.

- Optionally associate an icon with the target Bean.

```
public java.awt.Image getIcon(int iconKind) {
  if (iconKind == BeanInfo.ICON_MONO_16x16 ||
      iconKind == BeanInfo.ICON_COLOR_16x16 ) {
      java.awt.Image img = loadImage("ExplicitButtonIcon16.gif");
      return img;
  }
  if (iconKind == BeanInfo.ICON_MONO_32x32 ||
      iconKind == BeanInfo.ICON_COLOR_32x32 ) {
      java.awt.Image img = loadImage("ExplicitButtonIcon32.gif");
      return img;
  }
  return null;
}
```

The BeanBox displays this icon next to the Bean name in the ToolBox. You can expect builder tools to do similar.

- Specify the target Bean `class`, and, if the Bean has a customizer, specify it:

```
public BeanDescriptor getBeanDescriptor() {
   return new BeanDescriptor(beanClass, customizerClass);
}
...
private final static Class beanClass = ExplicitButton.class;
private final static Class customizerClass = OurButtonCustomizer.class;
```

Keep the `BeanInfo` class in the same directory as it's target class. The BeanBox first looks for a Bean's `BeanInfo` class in the target Bean's package path. If no `BeanInfo` is found, then the Bean information package search path (maintained by the `Introspector`) is searched. The default Bean information search path is `sun.beans.infos`. If no `BeanInfo` class is found, then low−level reflection is used to discover a Bean's features.

Using `BeanInfo` to Control What Features are Exposed

If you rely on low−level reflection to discover your Bean's features, all those properties, methods, and events that conform to the appropriate design patterns will be exposed in a builder tool. This includes any features in all base classes. If the BeanBox finds an associated `BeanInformation` class, then that information is used instead, and no more base classes are examined using reflection. In other words, `BeanInfo` information overrides low−level reflection information, and prevents base class examination.

By using a `BeanInfo` class, you can expose subsets of a particular Bean feature. For example, by not returning a method descriptor for a particular method, that method

will not be exposed in a builder tool.

When you use a `BeanInfo` class

- Base class features will *not* be exposed. You can retrieve base class features by using the `BeanInfo.getAdditionalBeanInfo()` method.
- Properties, events, or methods that have no descriptor will *not* be exposed. (Omitting a property, event, or method from the list returned by the equivalent `BeanInfo` getter method eliminates it from exposure) For a particular feature, only those items returned in the descriptor array will be exposed. For example, if you return descriptors for all your Bean methods except `foo()`, then `foo()` will not be exposed.
- Low−level reflection will be used for features with getter methods returning null. (Omitting a particular feature, by returning null with its getter method, causes low−level reflection to be used to discover elements of that feature in the current class and all base classes.) If you return null for a particular feature's getter method, low−level reflection will be used to discover that particular feature of the target Bean.

Feature Descriptors

`BeanInfo` classes contain *descriptors* that precisely describe the target Bean's features. The BDK implements the following descriptor classes:

- `FeatureDescriptor` is the base class for the other descriptor classes. It declares the aspects common to all descriptor types.
- `BeanDescriptor` describes the target Bean's class type and name, and describes the target Bean's customizer class if it exists.
- `PropertyDescriptor` describe the target Bean's properties.
- `IndexedPropertyDescriptor` is a subclass of `PropertyDescriptor`, and describes the target Bean's indexed properties.
- `EventSetDescriptor` describes the events the target Bean fires.
- `MethodDescriptor` describes the target Bean's methods.
- `ParameterDescriptor` describes method parameters.

The `BeanInfo` interface declares methods that return arrays of the above descriptors.

Using `BeanInfo` to Control What Features are Exposed

For base classes: low−level reflection will yield all features; `BeanInfo` will yield no base class features unless explicitly told to do so.

For properties, events, and methods, any feature left out will not show up. For example, if you have four methods in your Bean but only list three `MethodDescriptor` objects in your `BeanInfo`, the fourth method will not be accessible.

Writing Advanced Beans

Converting Beans into ActiveX Components

ActiveX Bridge section *UNDER CONSTRUCTION*
