

Using the Beans Development Kit 1.0

November 1997

A Tutorial

Alden DeSoto



2550 Garcia Avenue
Mountain View, CA 94043 U.S.A.
408-343-1400

Nov 97

Contents

1. Getting Started	1-1
Beans Development Kit (BDK)	1-1
Example Beans and Online Documentation	1-1
The BeanBox	1-2
Testing Sample Beans	1-2
Creating and Testing the Simplest Bean	1-5
Using the BeanBox to Create an Applet	1-7
2. Properties	2-1
Simple Properties	2-1
Indexed Properties	2-2
Bound Properties	2-3
Constrained Properties	2-6
Example Beans and Properties	2-7
3. Events	3-1
WaterEventObject	3-2

WaterSource	3-2
Valve	3-3
Pipe	3-5
Testing WaterSource, Valve, and Pipe	3-6
Example Beans and Events	3-8
4. Customization	4-1
Customizer Interface	4-1
PropertyEditor Interface	4-2
BeanInfo Interface	4-2
Example Beans and Customization	4-3
5. Persistence	5-1
What to Save	5-1
Changes and Versioning	5-1
6. Packaging	6-1
MANIFEST file	6-1
Example	6-1
Additional Jar and Manifest File Information	6-2

Getting Started



JavaBeans is a portable, platform-independent software component model written in Java. It enables developers to write reusable components once and run them anywhere - benefiting from the platform-independent power of Java.

Beans are Java classes that can be manipulated in a visual builder tool and composed together into applications. Any Java class that adheres to certain property and event interface conventions can be a Bean. This short tutorial provides simple examples of how to program to these conventions.

Beans Development Kit (BDK)

The Beans Development Kit (BDK) is a pure Java application whose only dependency is the Java Development Kit (JDK) 1.1. The BDK provides support for the JavaBeans APIs, a test container (the “BeanBox” to test Bean behavior), sample Beans complete with their source code, the JavaBeans Specification, and this Tutorial.

Example Beans and Online Documentation

Extensive online documentation for the sample Beans is available from `beans\doc\examples.html` in the distribution. The online documentation is an important complement to this book and provides descriptions, suggestions for experimentation, and in some cases annotated code for each sample Bean. The online documentation also provides information on the BeanBox.

The BeanBox

The BeanBox is a sample container for testing Beans. The BeanBox handles visible Beans, those Beans that have a visual element, and invisible Beans, those Beans that are purely computational objects.

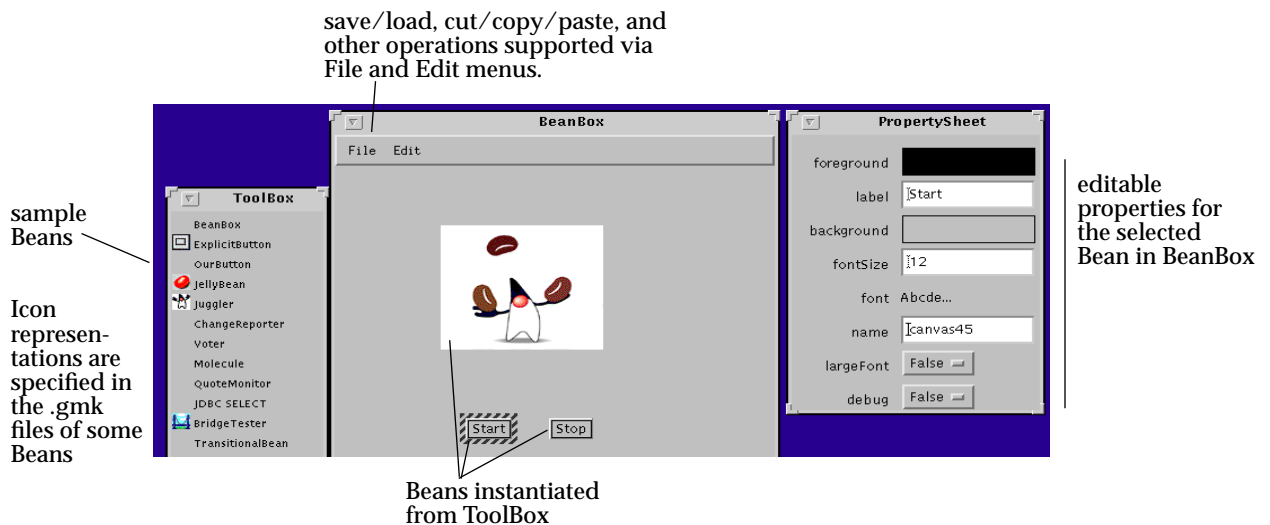
When you start the BeanBox, a ToolBox of sample Beans is displayed. Source code for these Beans is provided in the `demo\sunw\demo\` subdirectory of the distribution.

Testing Sample Beans

Start the BeanBox with the following commands:

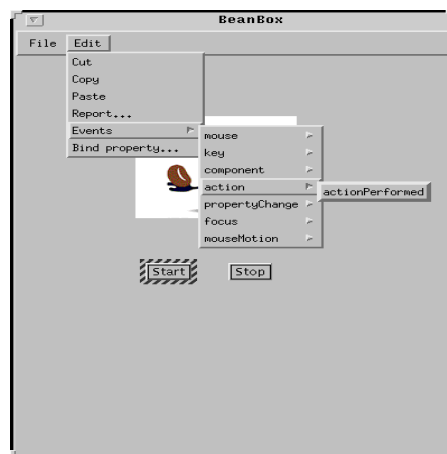
```
C:>cd beanbox  
C:>nmake run
```

The BeanBox, ToolBox, and PropertySheet appear on the screen. To instantiate a Bean in the BeanBox, click on the desired Bean in the ToolBox and then click in the BeanBox area. In the example below, a Juggler and two OurButtons have been instantiated in the BeanBox. The buttons have been labeled “Start” and “Stop” by editing the label property in the PropertySheet.



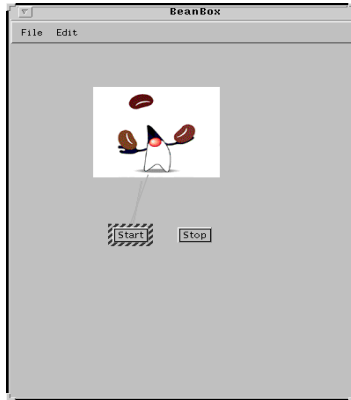
To test the OurButton and Juggler sample Beans:

1. Instantiate two OurButtons and a Juggler in the BeanBox as shown above. Label one button “start” and the other “stop” in the PropertySheet.
2. Select the “start” button.



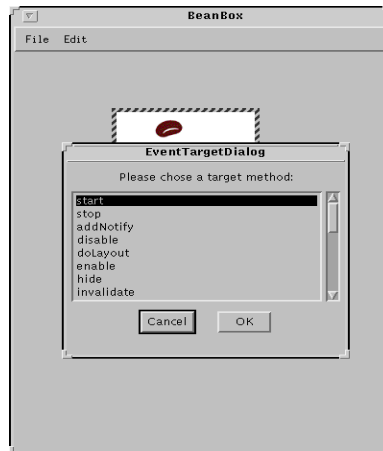
3. Select the **Edit-->Events-->action-->actionPerformed** pulldown menu as shown above.

The BeanBox positions a line under your mouse arrow which you can use to connect “start” to the Juggler.



4. Connect the line to the Juggler and click the mouse button.

The BeanBox responds with an Event Target Dialog as shown below. Juggler methods which either take no argument or which take an argument of type actionPerformed are listed in this dialog.



5. Select the start method and press “OK”.

The BeanBox will generate an adapter class. Once the BeanBox has generated this code, press the “start” button in the BeanBox and the Juggler will start juggling.

6. Connect the “stop” button to the Juggler stop method in the same fashion.

Test by pressing the “stop” button.

Creating and Testing the Simplest Bean

1. Create a SimplestBean.java source file as shown below.

Create a simplest directory under demo\sunw\demo\ and create a SimplestBean.java within it.

```

constructor
sets a visible _____
attribute

getMinimumSize —
assures that Bean
will be big enough
to see in BeanBox

package sunw.demo.simplest;
import java.awt.*;

public class SimplestBean extends Canvas{

    public SimplestBean(){
        setBackground(Color.red);
    }

    public Dimension getMinimumSize(){
        return new Dimension(50,50);
    }

}

```

2. Create a SimplestBean.mk file as shown below.

Create this file in the demo\ directory. Refer to the sample Bean .mk files provided in demo\ for additional examples.

```
CLASSFILES= \
    sunw\demo\simplest\SimplestBean.class

JARFILE= ..\jars\SimplestBean.jar

list of compiled class files — .SUFFIXES: .java .class
Beans in this location will be found by the BeanBox — all: $(JARFILE)
package classes “Java-Bean: True” causes class to appear in Toolbox | $(JARFILE): $(CLASSFILES) $(DATAFILES)
                                                                    jar cfm $(JARFILE) <<manifest.tmp sunw\demo\simplest\*.class
                                                                    $(DATAFILES)
                                                                    Name: sunw/demo/simplest/SimplestBean.class
                                                                    Java-Bean: True
                                                                    <<
                                                                    # Rule for compiling a normal .java file
                                                                    {sunw\demo\simplest}.java{sunw\demo\simplest}.class :
                                                                    set CLASSPATH=.
                                                                    javac $<

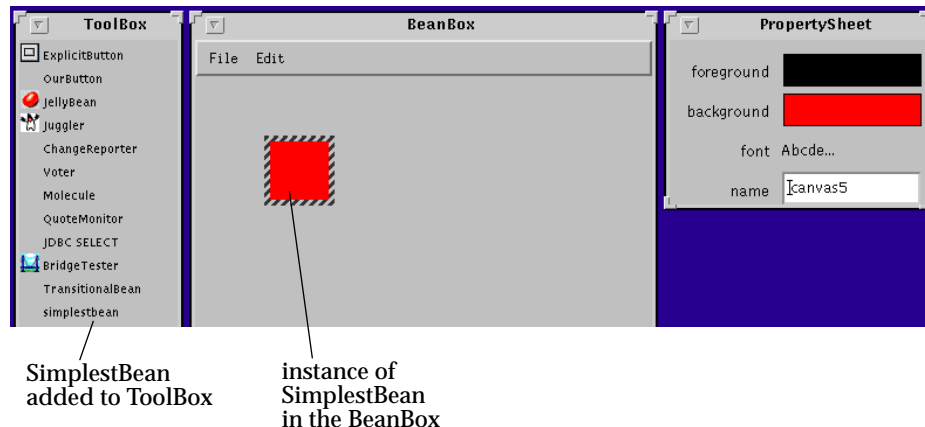
clean:
    -del sunw\demo\simplest\SimplestBean.class
    -del $(JARFILE)
```

3. Build the example

```
C:>nmake -f SimplestBean.mk
```

4. Run the BeanBox and create an instance of your SimplestBean.

Your simplestbean will automatically appear in the ToolBox at startup.



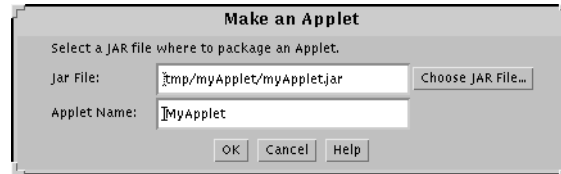
Using the BeanBox to Create an Applet

You can use the BeanBox's File | MakeApplet... menu selection to create an applet from the BeanBox contents. The resulting applet uses Java Object Serialization to record the state of the Beans.

The File | MakeApplet... menu item 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, and a readme file with complete information about the generated applet and all files involved. This generated readme file contains much useful information.

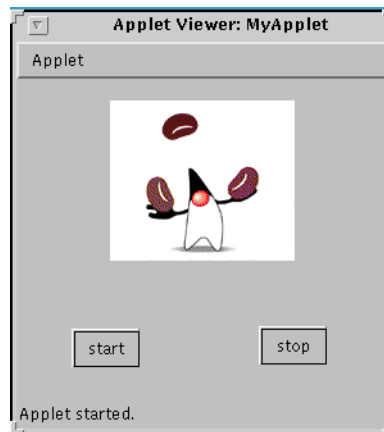
The generated applet can be used in any JDK 1.1-compliant browser. A good test platform is the JDK 1.1 appletviewer (see <http://java.sun.com/jdk/1.1/>). Another fully compliant browser is the HotJava browser (see <http://java.sun.com/products/hotjava>). At the time of writing (November '97) the current versions of Navigator and Internet Explorer have some limitations in their support for JDK1.1. The failures in IE4.0 included lack of support for the `Class.getResource()` call. In particular this means that our Juggler Bean will not show. Navigator 4.04 is throwing `SecurityExceptions` when `getResourceAsStream()` is invoked, we expect this to be fixed soon.

Here's a snapshot of the BeanBox's File | Make Applet dialog:



To see how Make Applet works, instantiate the Juggler Bean and two buttons, and connect them like you did at the beginning of this chapter.

1. The generated applet will have the same size as the BeanBox frame, so you may want to start by adjusting the BeanBox size to the size of the applet you want.
2. Choose File | Make Applet to bring up the above dialog. Use the default JAR file and applet name for this example.
3. Press the OK button. You can inspect your handiwork by moving to the `beanbox/tmp/myApplet` directory of your JDK installation.
4. Bring up `appletviewer` in the following way:
`appletviewer <BDKInstallation>/beanbox/tmp/myApplet.html.`
Here is what you should see:



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

Properties



A property is a single public attribute. Properties can be read/write, read-only or write-only. There are several types of properties: simple, indexed, bound, and constrained.

Simple Properties

A simple property represents a single value and can be defined with a pair of get/set methods. A property's name is derived from the method names. For example the method names `setX` and `getX` indicate a property named "X". A method name `isX` by convention indicates that "X" is a boolean property.

```
property will be called — ourString  
ourString  
public class alden2 extends Canvas {  
    String ourString="Hello";  
  
    public alden2(){  
        setBackground(Color.red);  
        setForeground(Color.blue);  
    }  
  
    "set" property — public void setString(String newString){  
        ourString = newString;  
    }  
  
    "get" property — public String getString() {  
        return ourString;  
    }  
  
    public Dimension getMinimumSize(){  
        return new Dimension(50,50);  
    }  
}
```

Indexed Properties

An indexed property represents an array of values. Property element get/set methods take an integer index parameter. The property may also support getting and setting the entire array at once.

The BDK 1.0 BeanBox does not support indexed properties.


```
public class alden3 extends Canvas {
    int[] dataSet={1,2,3,4,5,6};

    public alden3(){
        setBackground(Color.red);
        setForeground(Color.blue);
    }

    public void setDataSet(int[] x){
        dataSet=x;
    }

    public void setDataSet(int index, int x) {
        dataSet[index]=x;
    }

    public int[] getDataSet() {
        return dataSet;
    }

    public int getDataSet(int x) {
        return dataSet[x];
    }

    public Dimension getMinimumSize(){
        return new Dimension(50,50);
    }
}
```

dataSet is an indexed property —

set entire array —

set one element of array —

get entire array —

get one element of array —

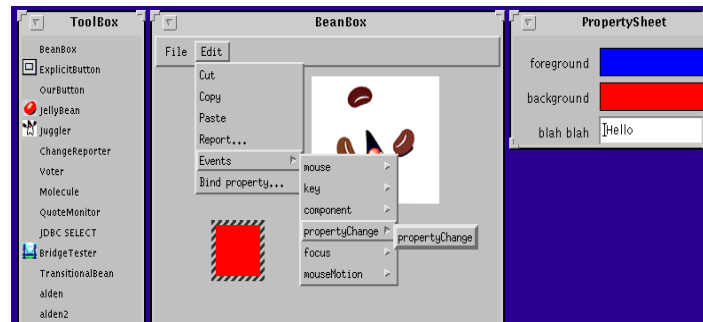
Bound Properties

A bound property notifies other objects when its value changes. Each time its value is changed, the property fires a `PropertyChange` event which contains the property name, old, and new values. Notification granularity is per bean, not per property.

<p>declare and instantiate a property change object</p>	<p>—</p>	<pre> public class alden5 extends Canvas { String ourString="Hello"; private PropertyChangeSupport changes = new PropertyChangeSupport(this); public alden5() { setBackground(Color.red); setForeground(Color.blue); } public void setString(String newString){ String oldString = ourString; ourString = newString; changes.firePropertyChange("string",oldString,newString); } public String getString() { return ourString; } public Dimension getMinimumSize() { return new Dimension(50,50); } public void addPropertyChangeListener(PropertyChangeListener l) { changes.addPropertyChangeListener(l); } public void removePropertyChangeListener(PropertyChangeListener l) { changes.removePropertyChangeListener(l); } } </pre>
<p>send change event to listeners when property is changed</p>	<p>—</p>	
<p>implement methods to add and remove listeners. The BeanBox will call these methods when a connection is made.</p>	<p> </p>	

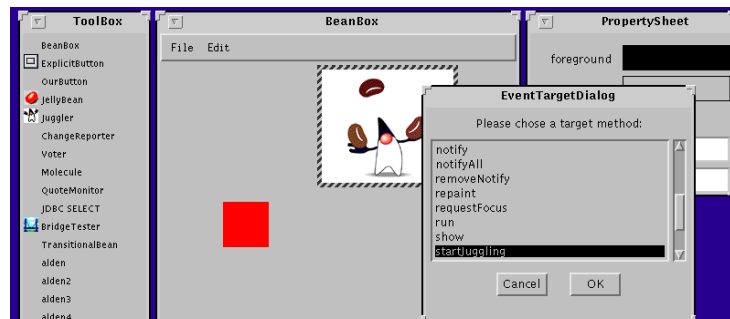
You can test bound properties in the BeanBox as follows.

- 1. Instantiate a Bean with bound properties and any other Bean in the Beanbox. Select the Bean with bound properties.**
- 2. Select the Edit-->Events-->propertyChange-->propertyChange pulldown menu as shown below.**



3. Connect the Bean with bound properties to the second Bean and select a target method.

The BeanBox will add the second bean to the bound property Bean's list of listeners.



4. When the BeanBox has finished generating code, change the bound property value in the PropertySheet.

The selected method on the listener bean will be invoked.

the source notifies any registered `propertyChange` listeners that the change has completed. If any vetoable change listener rejects the change then a new `vetoableChange` event will be delivered reverting to the previous value.

This allows a property watcher to either:

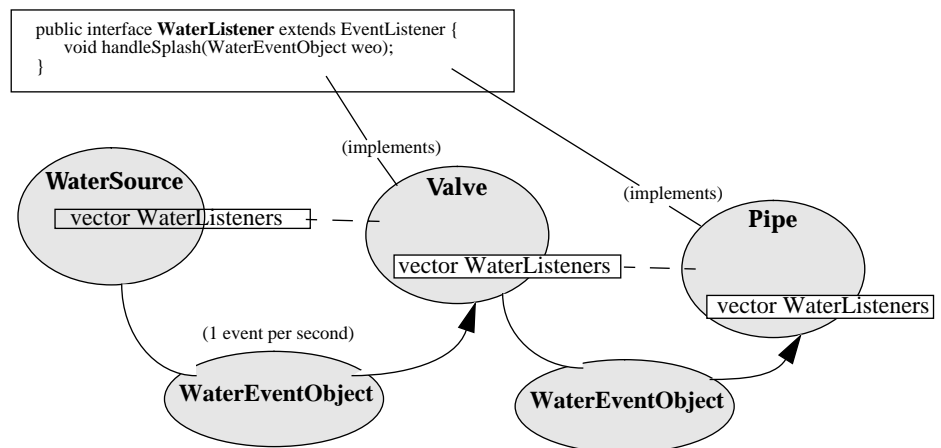
- treat constrained/bound property updates in a "two phase" fashion by registering both a `VetoableChangeListener` and a `PropertyChangeListener`. The watcher ignores the `vetoableChange` event unless it wants to veto the change. At `propertyChange` event time it acts on the new value, as it knows that this new value has successfully passed the `vetoableChange` phase.
- register only a `vetoableChange` listener. In this case, the watcher will be notified about proposed changes and will also get subsequently notified if the proposed change is vetoed. This approach means that the watcher is deliberately choosing to assume that vetoable changes will "pass" and is prepared to act on information that may be subsequently vetoed.

Example Beans and Properties

Several of the sample Beans illustrate properties. Refer to `beans\doc\examples.html`.

Events

This chapter uses three example Beans to explain Events: WaterSource, Valve, and Pipe. A WaterSource drips one WaterEventObject per second to its list of WaterListeners. The list of WaterListeners may include any number and/or combination of Valves and Pipes. An open Valve passes on WaterEventObjects that it receives to its own list of WaterListeners. A closed Valve does not pass on any WaterEventObjects. A Pipe behaves in the same way as an open Valve.



WaterEventObject

WaterListeners
check timeOfEvent
to determine
whether it is more
than 2 seconds old.

```
public class WaterEventObject extends EventObject {  
    long timeOfEvent;  
  
    public WaterEventObject(Object o) {  
        super(o);  
        timeOfEvent = System.currentTimeMillis();  
    }  
  
    public long getTimeOfEvent() {  
        return timeOfEvent;  
    }  
}
```

WaterSource

maintain a list of
objects which have
registered to receive
water events

```
public class WaterSource extends Canvas implements Runnable {  
  
    private Vector waterListeners = new Vector();  
    Thread thread;  
  
    public WaterSource() {  
        setBackground(Color.blue);  
        thread = new Thread(this);  
        thread.start();  
    }  
  
    public Dimension getMinimumSize()  
    {  
        return new Dimension(15,15);  
    }  
  
    public void run() {  
        while(true) {  
            splash();  
            try {  
                thread.sleep(1000);  
            } catch (Exception e) {}  
        }  
    }  
}
```


BeanBox will call these methods to add and remove registered listeners

```
public synchronized void addWaterListener(WaterListener l) {
    waterListeners.addElement(l);
}

public synchronized void removeWaterListener(WaterListener l) {
    waterListeners.removeElement(l);
}
```

send a water event to registered listeners

```
private void splash() {
    Vector l;
    WaterEventObject weo = new WaterEventObject(this);
```

you must copy the vector before sending the event in order to avoid a timing race

```
    synchronized(this) {
        l = (Vector)waterListeners.clone();
    }

    for (int i = 0; i < l.size(); i++) {
        WaterListener wl = (WaterListener) l.elementAt(i);
        wl.handleSplash(weo);
    }
}
```

Valve

```
public class Valve extends Canvas implements WaterListener,
    Runnable {
```

list of listeners
last water event received
open/close valve property

```
    private Vector waterListeners = new Vector();
    private WaterEventObject lastWaterEvent;
    private boolean open = true;
    Thread thread;
```

property get and set methods

```
    public Valve() {
        setBackground(Color.white);
        thread = new Thread(this);
        thread.start();
    }

    public boolean isOpen() {
        return open;
    }

    public void setOpen(boolean x) {
        open = x;
    }
}
```

this method is specified in the WaterListener interface (which this class implements).

```
public Dimension getMinimumSize() {
    return new Dimension(20,30);
}
```

```
public void handleSplash(WaterEventObject e) {
    lastWaterEvent = e;
    if (isOpen()) {
        setBackground(Color.blue);
        repaint();
        splash();
    }
}
```

make the valve white if a WaterEventObject has not been recieved in the last 2 seconds or if the valve is closed

```
public void run() {
    while(true) {
        try {
            thread.sleep(1000);
        } catch (Exception e) {}

        if (lastWaterEvent != null) {
            long dt = System.currentTimeMillis() -
                lastWaterEvent.getTimeOfEvent();
            if ((dt > 2000) || (!isOpen())) {
                setBackground(Color.white);
                repaint();
            }
        }
    }
}
```

BeanBox will call these methods to add and remove registered listeners

```
public synchronized void addWaterListener(WaterListener l) {
    waterListeners.addElement(l);
}

public synchronized void removeWaterListener(WaterListener l) {
    waterListeners.removeElement(l);
}
```

send a water event to registered listeners

```
void splash() {
    Vector l;
    WaterEventObject weo = new WaterEventObject(this);

    synchronized(this) {
        l = (Vector)waterListeners.clone();
    }
}
```

... method continued on next page

send a water event to registered listeners

... method continued from previous page

```

    for (int i = 0; i < l.size(); i++) {
        WaterListener wl = (WaterListener) l.elementAt(i);
        wl.handleSplash(weo);
    }
}

```

Pipe

```

public class Pipe extends Canvas implements WaterListener,
                                             Runnable {

```

list of listeners ——— private Vector waterListeners = new Vector();
last water event received — private WaterEventObject lastWaterEvent;
Thread thread;

```

    public Pipe() {
        setBackground(Color.white);
        thread = new Thread(this);
        thread.start();
    }

```

```

    public Dimension getMinimumSize() {
        return new Dimension(150,10);
    }

```

This method is specified in the WaterListener interface (which this object implements)

```

    public void handleSplash(WaterEventObject e) {
        lastWaterEvent = e;
        setBackground(Color.blue);
        repaint();
        splash();
    }

```

```

    public void run() {
        while(true) {
            try {
                thread.sleep(1000);
            } catch (Exception e) {}
        }
    }

```

make the pipe white if a water event has not been received in the last 2 seconds

```

        if (lastWaterEvent != null) {
            long dt = System.currentTimeMillis() -
                lastWaterEvent.getTimeOfEvent();
            if (dt > 2000) {
                setBackground(Color.white);
                repaint();
            }
        }
    }

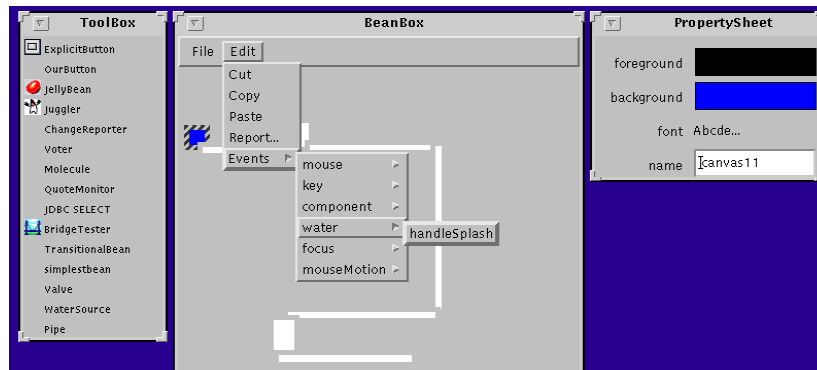
```

BeanBox will call these methods to add and remove registered listeners

```
    }  
    }  
    public synchronized void addWaterListener(WaterListener l) {  
        waterListeners.addElement(l);  
    }  
    public synchronized void removeWaterListener(WaterListener l) {  
        waterListeners.removeElement(l);  
    }  
    void splash() {  
        WaterEventObject weo = new WaterEventObject(this);  
        for (int i = 0; i < waterListeners.size(); i++) {  
            WaterListener wl =  
                (WaterListener)waterListeners.elementAt(i);  
            wl.handleSplash(weo);  
        }  
    }  
}
```

Testing WaterSource, Valve, and Pipe

- 1. Instantiate a collection of WaterSources, Valves, and Pipes in the BeanBox.**
- 2. Select a WaterSource Bean and invoke the Edit-->Events-->water-->handleSplash pulldown as shown in the picture below.**

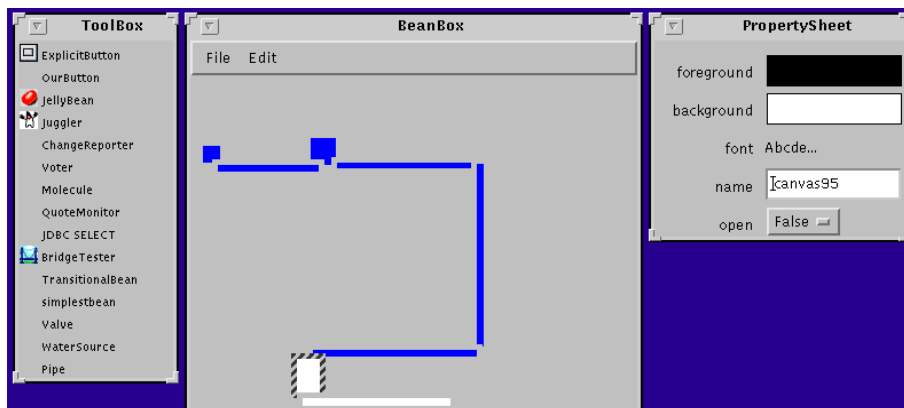


3. Connect the WaterSource to a Pipe or Valve and select the handleSplash method in the EventTargetDialog.

The BeanBox will generate an adaptor class.

4. Continue to connect water event producers to water event consumers as desired.

You can manipulate the water flow by turning valves on and off as illustrated in the example below.



Example Beans and Events

Several of the sample Beans illustrate events. Refer to `beans\doc\examples.html`.

Customization



You can customize how a Bean appears and behaves within a builder environment by using the Customizer, PropertyEditor, and BeanInfo interfaces as described in this chapter.

Customizer Interface

Implement the `java.beans.Customizer` interface to provide your own GUI implementation of the property sheet. For example, the `OurButton` bean in `demo\sunw\demo\buttons\` is packaged with a custom property sheet:

```
public OurButtonCustomizer extends Panel implements Customizer {
```

When implementing a custom property sheet such as `OurButtonCustomizer`, be sure to implement `addPropertyChangeListener` and `removePropertyChangeListener`. These will allow the `BeanBox` or other builder environment to add property event listeners for the Bean as required.

```

private PropertyChangeSupport support =
    new PropertyChangeSupport(this);

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

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

```

PropertyEditor Interface

Implement the `PropertyEditor` interface to create a custom editor for a specific property. The `MoleculeNameEditor` class in `demo\sun\demo\molecule\` of the distribution provides a good example of this.

If you provide a custom property editor class, you must refer to this class with a call to `PropertyDescriptor.setPropertyEditorClass` in a `BeanInfo` class (see next section).

`PropertyEditorSupport` is a basic implementation of the `PropertyEditor` interface

```

public class MoleculeNameEditor
    extends java.beans.PropertyEditorSupport {

    public String[] getTags() {
        String result[] = {
            "HyaluronicAcid",
            "benzene",
            "buckminsterfullerine",
            "cyclohexane",
            "ethane",
            "water"};
        return result;
    }
}

```

BeanInfo Interface

Each Bean class may have a `BeanInfo` class which customizes how the Bean is to appear in a builder. The `BeanInfo` can define properties, methods, events, with display names and short help.

The example shown below is from `MoleculeBeanInfo.java` in `demo\sunw\demo\molecule\` of the distribution.

`SimpleBeanInfo` is a basic implementation of the `BeanInfo` interface

Point to custom property editor

```
public class MoleculeBeanInfo extends SimpleBeanInfo {
    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            PropertyDescriptor pd = new PropertyDescriptor(
                "moleculeName", Molecule.class);
            pd.setPropertyEditorClass(MoleculeNameEditor.class);
            PropertyDescriptor result[] = { pd };
            return result;
        } catch (Exception ex) {
            System.err.println("MoleculeBeanInfo:
                unexpected exeption: " + ex);
            return null;
        }
    }
}
```

The `ExplicitButtonBean` in `demo\sunw\demo\buttons\` also illustrates the use of a `BeanInfo` class. `ExplicitButtonBeanInfo` defines four property descriptors, rather than just one as in `MoleculeBeanInfo`. Note that properties are displayed in the order they are listed in the `PropertyDescriptor`.

`ExplicitButtonBean` also illustrates the use of `EventSetDescriptor` and `BeanDescriptor`. `EventSetDescriptor` allows you to specify the text labels used in event dialogs and pulldowns. `BeanDescriptor` allows you to graphic image files to represent the Bean.

Example Beans and Customization

Refer to `beans\doc\examples.html` for additional discussion of sample Beans and customization.

To make fields in a Bean class persistent, simply define the class as implementing `java.io.Serializable`.

```
public class Button implements java.io.Serializable {  
}
```

The fields in any instance of a Bean which implements `Serializable` will automatically be saved. You need do nothing else. You can prevent selected fields from being saved by marking them `transient` or `static`; `transient` and `static` variables are not saved.

What to Save

Generally, a Bean should store the state of any exposed properties. Selected internal state variables may also be saved. Beans should not, however, store pointers to external Beans.

Changes and Versioning

As you update software, you can add fields, add or remove references to classes, change a field's `private/protected/public` status without altering the persistence schema of the class. However, deleting fields from the class, changing a variable's position in the class hierarchy, changing a field to or from `transient/static`, or changing a field's data type will change the persistence schema.

If you need to make changes to a class which alter its persistence, you might define a version id field which can be checked at runtime. For example,

```
static final long serialVersionUID 348749695999L;
```

JavaBeans are distributed through JAR files. A JAR file is a ZIP format archive file that may optionally have a MANIFEST file. The MANIFEST describes the contents of the JAR file. A JAR file may contain .class files, serialized Beans (.ser), help files in HTML format, and resources (images , audio, text).

MANIFEST file

If a JAR file does not have a MANIFEST, then all classes and serialized objects in the package are treated as beans. Providing a MANIFEST file allows you to specify which classes are Beans via "Java-Bean: True" entries (see Example below).

Example

This example .mk file illustrates the compiling and packaging of three Beans and two auxiliary classes. This .mk file was used to package the example discussed in chapter 3, "Events".

```
CLASSFILES= \  
    sunw\demo\valves\WaterListener.class \  
    sunw\demo\valves\WaterSource.class \  
    sunw\demo\valves\Valve.class \  
    sunw\demo\valves\Pipe.class \  
    sunw\demo\valves\WaterEventObject.class  
  
JARFILE= ..\jars\valves.jar  
  
.SUFFIXES: .java .class  
  
all: $(JARFILE)  
  
# Create a JAR file with a suitable manifest.  
  
$(JARFILE): $(CLASSFILES) $(GIFFILES)  
jar cfm $(JARFILE) <<manifest.tmp sun\demo\valves\*.class $(GIFFILES)  
  
do not display _____  
in ToolBox  
  
Name: sunw/demo/valves/WaterListener.class  
Java-Bean: False  
  
Name: sunw/demo/valves/WaterSource.class  
Java-Bean: True  
  
Name: sunw/demo/valves/Valve.class  
Java-Bean: True  
  
Name: sunw/demo/valves/Pipe.class  
Java-Bean: True  
  
do not display _____  
in ToolBox  
  
Name: sunw/demo/valves/WaterEventObject.class  
Java-Bean: False  
<<  
  
# Rule for compiling a normal .java file  
{sunw\demo\valves}.java{sun\demo\valves}.class :  
    set CLASSPATH=..\classes;.  
    javac $<  
  
clean:  
    -del sunw\demo\valves\*.class  
    -del $(JARFILE)
```

Additional Jar and Manifest File Information

Refer to `beans\doc\jar.html` in the distribution for more information.