Java Programming

Javabeans

Introduction

Visual Programming Languages (such as Visual Basic and Delphi) have been very popular in building GUI applications. In visual programming, you can drag and drop a visual component into a Application Builder and attach event handler to the component. Visual programming is ideal for *rapid prototyping* of GUI applications. Visual programming relies on component and event-driven technology. Components are *reusable software units* that can be assembled into an application via an application building tool (e.g., Visual Studio, JBuilder, Netbeans, Eclipse).

In Java, visual programming is supported via the "Javabean" API. The application builder tool loads the beans into a "toolbox" or "palette". You can select a bean from the toolbox, drop it into a "form", modify its appearance or properties, and define its interaction with other beans. Using the JavaBeans component technology, you can compose (or assemble) an application with just a few lines of codes.

"A Javabean is a *reusable software component* that can be manipulated visually in an application builder tool."

"A Javabean is an independent, reusable software component. Beans may be visual object, like Swing components (e.g. `JButton`, `JTextField`) that you can drag and drop using a GUI builder tool to assemble your GUI application. Beans may also be invisible object, like queues or stacks. Again, you can use these components to assemble your application using a builder tool."

Javabeans expose their features (such as properties, methods, events) to the application builder tools for visual manipulation. These feature names must adhere to a strict naming convention in order for them to be examined automatically. In other words, an application builder tool relies on these naming conventions to discover the exposed features, in a process known as introspection. For examples,

1. A property called `propertyName` of type `PropertyType` has the following convention:

   ```
   PropertyType propertyName                    // declaration
   public PropertyType getPropertyName()        // getter
   public void setPropertyName(PropertyType p)  // setter
   ```

2. For an event *source* object, which can fire an *event* called `XxxEvent` specified in an interface `XxxListener`, the following methods must be provided to register and remove *listener*:

   ```
   public void addXxxListener(XxxListener l)
   public void removeXxxListener(XxxListener l)
   ```

I assume that you are familiar with OOP concepts (such as interface, polymorphism) and GUI programming (in AWT and Swing). Otherwise, study the earlier chapters.

JavaBean Development Software

Bean Development Kit (BDK)

NOTE: BDK is no longer available for download from the Java website.

Bean Development Kit (BDK) is a tool for testing whether your Javabeans meets the JavaBean specification. Follow the instruction provided to install the BDK. Read the documentation and tutorial provided (in particular, "The Java Tutorial, specialized trial on JavaBeans"). BDK comes with a set of sample demo beans. You should try them out and closely study these demo beans before writing our own beans.

Let's try to assemble (or compose) an application using the BDK demo beans.

1. Start the "beanbox" by running "$bdk\beanbox\run.bat".
2. From the "Toolbox" window, select "Juggler" (a demo bean) and place it inside the "beanbox" (by clicking the desired location in the "beanbox" window). Observe the "Property" window of the Juggler bean.
3. Create a button by selecting "OurButton" demo bean from the "Toolbox" and place it inside the "Beanbox". In the "Proprety" window, change the "label" from "press" to "start".

4. Focus on "OurButton", choose "Edit”" from menu ⇒ "Events" ⇒ "mouse" ⇒ "mouseClicked" and place it onto the "Juggler" (i.e., "Juggler" is the target of this event). In the "EventTargetDialog", select method "startJuggling" as the event handler.
5. Create another button by selecting "OurButton" bean from "Toolbox" and place it inside the "Beanbox" again. In the "Proprety" window, change the "label" from "press" to "stop".
6. Focus on the stop button, choose "Edit" from menu ⇒ "Events" ⇒ "mouse" ⇒ "mouseClicked" and place it onto the "Juggler". In the "EventTargetDialog", select method "stopJuggling" as the event handler.
7. Click on the buttons, and observe the result.

[TODO] BDK diagram

It is easy to assemble an application from components. You can do it without writing a single line code, if these components are readily available.

NOTES:

- BDK is old (since JDK 1.1), and does not make use of many of the latest Java features. For example, it uses AWT GUI classes rather than the Swing.
- To run BDK under JDK 1.5 and above, you may have to recompile the program.

Bean Builder

Bean Builder can be downloaded from https://bean-builder.dev.java.net/.

[TODO]

NetBeans

[TODO]

Writing Your Own Javabeans

The JavaBeans APIs covers five aspects:

1. *Properties*: represent the attributes of a component.
2. *Event Handling*: allows beans to communicate with each others (JavaBean uses JDK 1.1 AWT event-delegation model).
3. *Persistence*: allows beans' internal states to be stored and later restored.
4. *Introspection*: allows Application Builder tool to analyze beans.
5. *Application Builder Tool*: for composing applications from Javabeans components.

Property:

A bean has properties, which define the attributes of the bean, and can be manipulated by an application builder tool.

Javabean Property Naming Convention: For a *property* called `propertyName` of type `PropertyType`, a *getter* and a *setter* method must be defined as follows:

```
private PropertyType PropertyName                  // declare
public PropertyType getPropertyName()              // getter
public void setPropertyName(PropertyType value) // setter
```

For properties of `boolean` type, the getter shall be:

```
private boolean PropertyName
public boolean isPropertyName()                    // getter for boolean property
public void setPropertyName(boolean value)         // setter
```

First Javabean - A LightBulb

Let's create our first bean - a light bulb, which can be switched on or off.

```
package elect;
import java.awt.*;
import java.io.Serializable;

public class LightBulb extends Canvas implements Serializable {

   public LightBulb() {      // constructor
       setSize(50,50);
       setBackground(Color.GRAY);
   }

   // Properties
   private static final Color COLOR_OFF = Color.BLACK;
   private Color color = Color.ORANGE;         // property with a default value
   public Color getColor() { return color; }  // getter
   public void setColor(Color color) { this.color = color; } // setter

   boolean on = false;                         // property with a default value
   public boolean isOn() { return on; }        // getter for boolean
   public void setOn(boolean on) { this.on = on; } // setter

   // Override the paint() method to draw the LightBulb
   public void paint(Graphics g) {
      if (on) g.setColor(color);
      else g.setColor(COLOR_OFF);
      g.fillOval(10, 10, 30, 30);
   }

   public void switchOn() {   // switch on the Light
      on = true;
      repaint();
   }

   public void switchOff() {  // switch off the Light
      on = false;
      repaint();
   }

   public void toggle() {   // If on turns off; else turns on
      on = !on;
      repaint();
   }
}
```

Dissecting "`LightBulb.java`"

- A JavaBean must implement `java.io.Serializable` interface (a tag interface without declaring any method). It is to ensure that the internal state of a bean can be stored in an external persistent storage and later restored.
- This bean has two `private` properties: `color` (of the type `Color`) and `on` (of the type `boolean`). Each of the `private` properties has its own `public` getter and setter, which follow the Javabeans property naming convention. The builder tool can discover (or introspect) these `private` properties based on the `public` getters and setters, based on the property naming convention.
- Three `public` methods are defined: `switchOn()`, `switchOff()`, and `toggle()`. These methods can be used as event handlers, which will be fired upon triggering of a certain event.

This bean, belonging to the package `elect`, has a fully-qualified class name of "`elect.LightBulb`", with a corresponding directory structure of "`elect\lightBulb`".

Suppose that the source and class files are kept in separate directories as shown below (so that classes can be distributed without the source). The source file is denoted as "`$SRC_BASEDIR\elect\lightBulb.java`", where `$SRC_BASEDIR` denotes the base directory of the source file ("`c:\javabeans\src`" in our example). The class is denoted as "`$CLASS_BASEDIR\elect\LightBulb.class`" where `$CLASS_BASEDIR` denotes the base directory of the classes.

[TODO] directory diagram

To compile the all the source files in the package and place the classes in `$CLASS_BASEDIR`:

```
> cd $SRC_BASEDIR                        // set current directory at source base directory
> javac -d $CLASS_BASEDIR elect\*.java  // compile the entire package and place in class base directory
```

Create a *manifest* (to be included into the jar file) called "`$CLASS_BASEDIR\manifest.Bulb`" as follows:

```
Manifest-Version: 1.0

Name: elect/LightBulb.class
Java-Bean: True
```

Notes:

- The last line must be terminated with a newline.
- Use forward slash `'/'` as the directory separator.

Put the bean into a jar file (because builder tool usually loads a bean from a jar file), assuming that the jar file is to be kept in directory `$JAR_BASEDIR`:

```
> cd $CLASS_BASEDIR
> jar cmfv manifest.Bulb $JAR_BASEDIR/lightbulb.jar elect\LightBulb.class
```

Notes:

- The `'c'` option is used for *creating* new jar file. The `'m'` option indicates that a *manifest* file is provided. The `'f'` option indicates that the *filename* of the output jar file is provided. The `'v'` option enables the *verbose* mode.
- You can keep you manifest and jar file in any directory. But the jar command must be issue from `$CLASS_BASEDIR` and you have to follow the package sub-directory structure to reference you classes, so that they can be retrieved later.
- The name of the jar file is usually in lowercase.

Testing the LightBulb Bean (with BDK)

1. Start the BDK beanbox (by executing "`$BDK\beanbox\run.bat`").
2. From the "`Beanbox`" window, choose "File" ⇒ "LoadJar" ⇒ Chose "`lightbulb.jar`". You shall see `LightBulb` appears at the bottom of the "ToolBox" window.
3. Select LightBulb from the "ToolBox" window, and place it into the "Beanbox". Observe that the "Property" window shows the two properties defined in this bean: `color` and `on`. Try changing these properties and observe the result. The "Property" window also shows the properties inherited from the superclasses.
4. Select "OurButton" bean (a demo bean provided by BDK) from "Toolbox" and place it inside the "Beanbox". In the "Property" window, change the "label" from "press" to "toggle".
5. Focus on "OurButton", choose "Edit" from menu ⇒ "Events" ⇒ "mouse" ⇒ "mouseClicked" and place it onto the `LightBulb`. In the "EventTargetDialog", select method `toggle()` as the event handler.
6. Click the "toggle" button and observe the result.
7. Focus on `LightBulb`, choose "Edit" from menu ⇒ "Events" ⇒ "mouse" ⇒ "mouseClicked" and place it back to `LightBulb`. In the "EventTargetDialog", select method `toggle()` as the event handler.
8. Click the `LightBulb` and observe the effect.

[TODO] Test under Bean Builder and NetBeans.

Second Javabean - a Switch

Let's write another bean, a switch, that can be used to switch on/off the light bulb.

```java
package elect;
import java.awt.*;
import java.io.Serializable;

public class Switch extends Canvas implements Serializable {

   public Switch() { setSize(80,40); }  // constructor

   private boolean closed = false;      // property
   public boolean isClosed() { return closed; }
   public void setClosed(boolean b) { closed = b; }

   public void paint(Graphics g) {
      g.drawLine(10, 20, 30, 20);
      g.drawLine(50, 20, 70, 20);
      g.fillOval(30-2, 20-2, 5, 5);
      g.fillOval(50-2, 20-2, 5, 5);
      if (closed)
         g.drawLine(30, 20, 50, 20);
      else
         g.drawLine(30, 20, 47, 10);
   }

   // Toggle the switch
   public void toggle() {
      closed = !closed;
      repaint();
   }
}
```

Dissecting "`Switch.java`"

- One `private` property, `closed` (of the type `boolean`), is defined. The `public` setter and getter are also defined, which conforms to the Javabean property naming convention.

- A `public` method called `toggle()` is provided, to be used as event handler.

Testing the Switch Bean (with BDK)

1. Start the BDK beanbox and Load the `Switch` bean and `LightBulb` bean (created earlier).
2. Select `Switch` bean from "Toolbox", and place it into the "Beanbox". Observe the "Property" window.
3. Select `LightBulb` from "Toolbox", and place it into the beanbox.
4. Focus on the `Switch` bean, choose "Edit" from the menu ⇒ "Events" ⇒ "mouse" ⇒ "mouseClicked" ⇒ place it onto the `LightBulb`. In the "EventTargetDialog", select method `toggle()` as the event handler.
5. Focus on the `Switch`, choose "Edit" from the menu ⇒ "Events" ⇒ "mouse" ⇒ "mouseClicked" ⇒ place it back to the `Switch`. In the "EventTargetDialog", select method `toggle()` as the event handler.
6. Click the `Switch` and observe the result.

Event Handling

Beans communicate with other beans by sending and receiving event notification. Javabeans use the AWT event-delegation model (in package `java.awt.event`, since JDK 1.1). The model consists of three parts: *source*, *listener* and *event*. The procedures for event handling are:

1. For a particular *event* says `XxxEvent`, a companion `XxxListener` interface is set up, which declares the various methods that could be fired under this `XxxEvent`. For example, for `java.awt.event.MouseEvent`, an interface called `java.awt.event.MouseListener` is declared with the following `abstract` methods:

   ```
   public void mouseClicked(MouseEvent evt)
   public void mouseEntered(MouseEvent evt)
   public void mouseExited(MouseEvent evt)
   public void mousePressed(MouseEvent evt)
   public void mouseReleased(MouseEvent evt)
   ```

2. The source object that fires `XxxEvent` must maintain a `private` listener list of `XxxListener`, and provide two `public` methods for registering and removing listener to and from this list.

   ```
   public void addXxxListener(XxxListener lis)
   public void removeXxxListener(XxxListener lis)
   ```

   For example, the `java.awt.Component` may fire `MouseEvent`, the following two methods are provided to register and remove listener in `Component`.

   ```
   public void addMouseListener(MouseListener lis)
   public void removeMouseListener(MouseListener lis)
   ```

3. A listener interested in `XxxEvent` must (a) implement `XxxListener` interface and provide implementation to all the abstract methods declared in the interface; (b) register with the source object via the `source.addXxxListener(XxxListener lis)` method. Notice that the listener object is upcasted to the super-type `XxxListener`.
4. When the particular event is triggered on the source object, e.g., `mouseClicked` of `MouseListener`, the source object scans its listener list and invoke the corresponding method (e.g., `mouseClicked()`) for all the listeners.

In BDK, you set the "focus" on the *source* object and choose the *event method* (e.g., `mouseClicked()` of `MouseListener`). The *source* object is then connected to the *listener* object (called *event target* in BDK) and you pick the handling method, among the available methods in the listener object. BDK automatically create an adapter class (kept in "`$BDK\beanbox\tmp\sunw\beanbox`"). For example,

```
// Automatically generated event hookup file.
package tmp.sunw.beanbox;
import elect.Bulb;
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;

public class ___Hookup_1a379733cc implements java.awt.event.MouseListener, java.io.Serializable {

    public void setTarget(elect.Bulb t) {
        target = t;
    }

    public void mouseClicked(java.awt.event.MouseEvent arg0) {
        target.toggle();
    }
```

```
    public void mouseEntered(java.awt.event.MouseEvent arg0) { }
    public void mouseExited(java.awt.event.MouseEvent arg0) { }
    public void mousePressed(java.awt.event.MouseEvent arg0) { }
    public void mouseReleased(java.awt.event.MouseEvent arg0) { }
    private elect.Bulb target;
}
```

In the LightBulb example, the methods `switchOn()`, `switchOff()` and `toggle()` can be used as the event handlers in response to firing of an event (such as `MouseEvent` of mouse-clicked).

Methods in Javabeans that can be exposed to application builder tool for use as the event handlers are ordinary Java methods except that they take no argument or a single argument of `XxxEvent` (or its super-type `java.util.EventObject`).

BeanInfo

An application builder tool can examine and exposes a bean's feature (such as properties, methods, and event) in a "properties sheet". This discovery process is called *introspection*.

You can optionally provide an additional `BeanInfo` class to a bean. The `BeanInfo` can be used to:

- Restrict the properties, methods and events available to the builder tools.
- Associate an icon with the bean.
- Specify a customizer class.
- Provide a more descriptive display name, or additional information about a bean feature.

Creating BeanInfo

The procedure for writing the companion `BeanInfo` class is:

1. Name your `BeanInfo` class by appending "`BeanInfo`" to the target bean class. E.g., if the bean is called "`LightBulb`", the companion `BeanInfo` class must be called "`LightBulbBeanInfo`".
2. `BeanInfo` is specified in the interface `java.beans.BeanInfo`. `BeanInfo` interface declares the following `abstract` methods:

```
BeanDescriptor getBeanDescriptor()
PropertyDescriptor[] getPropertyDescriptors()
EventSetDescriptor[] getEventSetDescriptors()
MethodDescriptor[] getMethodDescriptors()
Image getIcon(int iconType)
int getDefaultEventIndex()
int getDefaultPropertyIndex()
BeanInfo[] getAdditionalBeanInfo()
```

   `BeanInfo` interface declares the following constants for use in `getIcon()` method:

```
static int ICON_COLOR_16x16     // 16x16 color icon
static int ICON_COLOR_32x32     // 32x32 color icon
static int ICON_MONO_16x16      // 16x16 monochrome icon
static int ICON_MONO_32x32      // 32x32 monochrome icon
```

   Application builder tool can invoke `getPropertyDescriptors()`, `getEventSetDescriptors()` and `getMethodDescriptors()` to obtain the properties, events and methods that are exposed by the bean to the application builder tool. `BeanInfo` also maintains up to four icons. The application builder tool can retrieve these icons by invoking the `getIcon(int iconType)`.

   Instead of implementing `java.beans.BeanInfo` directly, you can sub-class from `java.beans.SimpleBeanInfo`. The `SimpleBeanInfo` provides default implementation of the `BeanInfo` interface. You only have to override the desired methods.

Exposing Properties

The following `BeanInfo` (called `LightBulbBeanInfor`) exposes two properties of the target bean "`LightBulb`": `color` and `on`.

```
package elect;
import java.beans.*;

public class LightBulbBeanInfo extends SimpleBeanInfo {

   // This beaninfo class is meant for the following bean
   private final static Class beanClass = LightBulb.class;

   public BeanDescriptor getBeanDescriptor() {
      return new BeanDescriptor(beanClass);
   }

   // Publish the "properties" available to builder tools
   public PropertyDescriptor[] getPropertyDescriptors() {
      try {
         PropertyDescriptor color =
            new PropertyDescriptor("color", beanClass);
         color.setBound(true);
         PropertyDescriptor on =
            new PropertyDescriptor("on", beanClass);
         on.setBound(true);
         PropertyDescriptor rv[] = {color, on};
         return rv;
      } catch (IntrospectionException e) {
         throw new Error(e.toString());
      }
   }
   public int getDefaultPropertyIndex() { return 1; }
}
```

You need to jar the BeanInfo together with the target bean:

```
> cd $CLASS_BASEDIR
> jar cmfv manifest.Bulb $JAR_BASEDIR/lightbulb.jar elect\LightBulb*.class
```

Try loading the jar file into BDK, and observe that only the property declared in the `BeanInfo` are exposed in the "Property" window.

Exposing Methods

The following codes in `BeanInfo` expose three methods of the bean "`LightBulb`": `switchOn()`, `swithcOff()` and `toggle()`. Notice that these methods take no argument. You can expose methods with no argument or with an argument of of `EventObject` only. These exposed methods can be used as event handling method of the listener.

```
   // Expose only the selected methods to the Builder tool
   public MethodDescriptor[] getMethodDescriptors() {

      Class args[] = {};       // argument of the method
      // Use "Class args[] = { java.util.EventObject.class };"
      // if the method takes Event as sole argument.

      try {
         MethodDescriptor toggle =
            new MethodDescriptor(
               LightBulb.class.getMethod("toggle", args));
         MethodDescriptor switchOn =
            new MethodDescriptor(
               LightBulb.class.getMethod("switchOn", args));
         MethodDescriptor switchOff =
            new MethodDescriptor(
               LightBulb.class.getMethod("swtichOff", args));
         MethodDescriptor result[] = {toggle, switchOn, switchOff};
         return result;
      } catch (Exception ex) {
         ex.printStackTrace();
      }
   }
```

Exposing Events

For exposing events, we shall work on `Switch` bean, which is a source object capable of firing MouseEvent. Create a `BeanInfo` class called `SwitchBeanInfo` for the `Swtich` bean as follows:

```
package elect;
import java.beans.*;

public class SwitchBeanInfo extends SimpleBeanInfo {

   // This beaninfo class is meant for the following bean
   private final static Class beanClass = Switch.class;

   public BeanDescriptor getBeanDescriptor() {
      return new BeanDescriptor(beanClass);
   }

   // publish the "events" availeble to builder tools
   public EventSetDescriptor[] getEventSetDescriptors() {
      try {
         // To expose MouseEvent
         String[] mouseListenerMethods =
            {"mouseClicked", "mousePressed", "mouseReleased",
             "mouseEntered", "mouseExited"};
         EventSetDescriptor mouse =
            new EventSetDescriptor(
               beanClass,
               "mouse",
               java.awt.event.MouseListener.class,
               mouseListenerMethods,
               "addMouseListener",
               "removeMouseListener");
         mouse.setDisplayName("mouse");

         EventSetDescriptor[] rv = {mouse};
         return rv;
      } catch (IntrospectionException e) {
         ex.printStackTrace();
      }
   }
}
```

Try:

- Tailor your `SwitchBeanInfo` to expose the property `closed`.
- Tailor your `SwitchBeanInfo` to expose the method `toggle()`.

Associating Icons with the Bean

The following codes in `BeanInfo` can be used to provide up to four icons (16x16 or 32x32, color or monochrome) for a bean to the application builder tool:

```
    // Specify the "icons" availeble to builder tools
    public java.awt.Image getIcon(int iconKind) {
       if (iconKind == BeanInfo.ICON_MONO_16x16 ||
           iconKind == BeanInfo.ICON_COLOR_16x16 ) {
          java.awt.Image img = loadImage("LightBulbIcon16.gif");
          return img;
       } else if (iconKind == BeanInfo.ICON_MONO_32x32 ||
          iconKind == BeanInfo.ICON_COLOR_32x32 ) {
          java.awt.Image img = loadImage("LightBulbIcon32.gif");
          return img;
       } else {
          return null;
       }
    }
```

You have to jar the icon images together with the `BeanInfo` and target bean class:

```
> cd $CLASS_BASEDIR
> jar cmfv manifest.Bulb $JAR_BASEDIR/lightbulb.jar elect\LightBulb*.class LightBulbIcon*.gif
```

Try:

- Try creating some icons and add them into the `BeanInfo` for `Switch` as well as `LightBulb`.

Specifying a Customizer & Property Editors

A bean's appearance and behavior can be customized at design time by an application builder tool. There are two ways to customize a bean:

- Using property editors: Each bean property has its own property editor which can be displayed on the property sheet.
- Using customizers: gives you complete control over bean’s customization, to be used when the property editors are not practical or application.

A property editor is a tool for customizing a particular property type. A property editor implements `java.beans.PropertyEditor` interface, which provides methods that specify how a property should be displayed in a property sheet.

```
public Object getValue();
public void setValue(Object value);
public String getAsText();
public void setAsText(String text);
public boolean isPaintable();
public void paintValue(Graphics g, Rectangle box);
public boolean supportsCustomEditor();
public Component getCustomEditor();
```

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 be used.

The property editors of type `Color` and `Font` use a separate panel, and use the property sheet to display the current property value. To display the current property value inside the property sheet, you need to override `isPaintable()` to return `true`, and override the `paintValue()` to paint the current property value in a rectangle in the property sheet. For example, in the `ColorEditor`:

```
public void paintValue(Graphics g, Rectangle box) {
   Color oldColor = g.getColor();
   g.setColor(Color.BLACK);
   g.drawRect(box.x, box.y, box.width-3, box.height-3);
   g.setColor(color);
   g.FillRect(box.x+1, box.y+1, box.width-4, box.height-4);
   g.setColor(oldColor);
}
```

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

Property editors are discovered and associate with a given property by:

- Explicit association via the associated BeanInfo object of the target bean.
- Explicitly register via `java.beans.PropertyEditorManager.registerEditor()`. This method takes two arguments: the class type, and the editor to be associated with that type.
- Search by appending "`Editor`" to the fully qualified name, e.g., "`elect.LightBulbEditor`".

The following codes can be used to specify a `Cstomizer` in `BeanInfo`.

```
// Specify the target Bean class, and,
// If the Bean has a customizer, specify it also.
private final static Class beanClass = LightBulb.class;
private final static Class customizerClass = LightBulbCustomizer.class;

public BeanDescriptor getBeanDescriptor() {
    return new BeanDescriptor(beanClass, customizerClass);
}
```

[TODO] Example.

More on Properties

Bound Property and PropertyChangeEvent

When a property of a bean changes, you may want another bean to be notified of the change and react to the change. These properties are called *bound property*.

Javabean API introduces a new event called `java.beans.PropertyChangeEvent` with an associated interface called `java.beans.PropertyChangeListener`. This interface declares the following `abstract` methods:

```
public void propertyChange(PropertyChangeEvent evt)
public String getPropertyName()
```

A source bean can fire `PropertyChangeEvent` whenever the value of a bound property is changed, e.g., via the Property Editor.

To provide support for `PropertyChangeEvent`, the source bean containing bound properties must maintain a list of property change listeners and allow registering and removing of listener to and from the list. To provide such support, either you do you own coding or make use of the `java.beans.PropertyChangeSupport` by including the following codes in your source bean:

```
// "this" object maintains the property change listener list
private PropertyChangeSupport changes = new PropertyChangeSupport(this);

// Listeners can be registered using these methodss.
public void addPropertyChangeListener(PropertyChangeListener l) {
   changes.addPropertyChangeListener(l);
}
public void removePropertyChangeListener(PropertyChangeListener l) {
   changes.removePropertyChangeListener(l);
}
```

For the sake of illustration, let's modify our `Switch` bean to set the `boolean` property `closed` as a bound property that fires `PropertyChangeEvent` if the value is changed. Let's called this new bean "`SwitchBound.java`".

```
package elect;
import java.awt.*;
import java.io.Serializable;
import java.beans.*;

public class SwitchBound extends Canvas implements Serializable {

   public SwitchBound(){ setSize(80,40); }  // constructor

   private boolean closed = false;          // property
   public boolean isClosed() { return closed; }

   // Property change support for bound property
   private PropertyChangeSupport changes = new PropertyChangeSupport(this);
   public void addPropertyChangeListener(PropertyChangeListener lis) {
         changes.addPropertyChangeListener(lis);
   }
   public void removePropertyChangeListener(PropertyChangeListener lis) {
         changes.removePropertyChangeListener(lis);
   }

   public void setClosed(boolean newStatus) {
      boolean oldStatus = closed;
      closed = newStatus;
      changes.firePropertyChange("closed", new Boolean(oldStatus),new Boolean(newStatus));
   }

   public void paint(Graphics g) {
      g.drawLine(10, 20, 30, 20);
      g.drawLine(50, 20, 70, 20);
      g.fillOval(30-2, 20-2, 5, 5);
      g.fillOval(50-2, 20-2, 5, 5);
      if (closed) g.drawLine(30, 20, 50, 20);
      else g.drawLine(30, 20, 47, 10);
   }

   // Toggle the switch
   public void toggle() {
      closed = !closed;
      repaint();
   }
}
```

Dissecting "`SwitchBound.java`"

- The `PropertyChangeSupport` codes are included.
- The setter for the bound property `closed` is modified to fire the `PropertyChangeEvent`. "`change`" refers to the `PropertyChangeSupport` instance declared earlier. All the registered listeners will receive the `PropertyChangeEvent` if this setter method is invoked.
- For JDK 1.5, you can use the *autoboxing* feature and change the statement to:

```
changes.firePropertyChange("closed", oldStatus, newStatus);  // autobox to boolean
```

Testing the Property Change

1. Start the beanbox. Load the `SwitchBound` and `LightBulb` beans. Drop an instance of `SwitchBound` and an instance of `LightBulb` into the beanbox.
2. Focus of `SwtichBound`, select "Event" ⇒ "bound property change" ⇒ "property change" and choose `LightBulb` as the target of the Event. Select `toggle()` as the handling method.
3. Focus on `SwtichBound`, change the bound property `closed` in the "Property" window and observe the result.

Try:

- Create the `BeanInfo` for `SwitchBound` to expose the "mouse" event and "property change" events.

```java
package elect;
import java.beans.*;

public class SwitchBoundBeanInfo extends SimpleBeanInfo {

   // This beaninfo class is meant for the following bean
   private final static Class beanClass = SwitchBound.class;

   public BeanDescriptor getBeanDescriptor() {
      return new BeanDescriptor(beanClass);
   }

   // publish the "events" availeble to builder tools
   public EventSetDescriptor[] getEventSetDescriptors() {
      try {
         // To expose MouseEvent
         String[] mouseListenerMethods =
               {"mouseClicked", "mousePressed", "mouseReleased",
                "mouseEntered", "mouseExited"};
         EventSetDescriptor mouse =
            new EventSetDescriptor(
               beanClass,
               "mouse",
               java.awt.event.MouseListener.class,
               mouseListenerMethods,
               "addMouseListener",
               "removeMouseListener");
         mouse.setDisplayName("mouse");

         // To expose PropertyChangeEvent
         EventSetDescriptor changed =
            new EventSetDescriptor(
               beanClass,
               "propertyChange",
               java.beans.PropertyChangeListener.class,
               "propertyChange");
         changed.setDisplayName("bound property change");

         EventSetDescriptor[] rv = {mouse, changed};
         return rv;
      } catch (IntrospectionException ex) {
         ex.printStackTrace();
      }
   }
}
```

- Create the `BeanInfo` for `SwitchBound` to expose the property `closed`.
- Create the `BeanInfo` for `SwitchBound` to expose the method `toggle()`.

Constrained Property & VetoableChangeEvent

A bean property is constrained when any change to that property can be vetoed by another object. The mechanism is similar to bound property and consists of three parts:

- A source bean contains one or more constrained properties.
- Listener object(s) that implement `java.beans.VetoableChangeListener` interface. The `VetoableChangeListener` interface declares one abstract method:

```java
public void vetoableChange(PropertyChangeEvent evt)
```

- A `PropertyChangeEvent` object containing the property name, and its old and new values.

For illustrating purpose, let's rewrite our `Switch` (called `SwitchConstrained`) with `closed` as a constrained property. Instead of implementing `VetoableChangeListener` interface directly, we use an adaptor class called `VetoableChangeSupport` (similar to `PropertyChangeSupport` for bound property).

```java
package elect;
import java.awt.*;
import java.io.Serializable;
import java.beans.*;

public class SwitchConstrained extends Canvas implements Serializable {

   public SwitchConstrained() { setSize(80,40); }  // constructor

```

```
    private boolean closed = false;      // Property
    public boolean isClosed() { return closed; }

    // Property change support for constrained property
    private VetoableChangeSupport vetos = new VetoableChangeSupport(this);
    public void addVetoableChangeListener(VetoableChangeListener lis) {
         vetos.addVetoableChangeListener(lis);
    }
    public void removeVetoableChangeListener(VetoableChangeListener lis) {
         vetos.removeVetoableChangeListener(lis);
    }

    public void setClosed(boolean newStatus) throws PropertyVetoException {
       boolean oldStatus = closed;
       // 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("closed", new Boolean(oldStatus),new Boolean(newStatus));
       // No-one vetoed, so go ahead and make the change.
       closed = newStatus;
    }

    public void paint(Graphics g) {
       g.drawLine(10, 20, 30, 20);
       g.drawLine(50, 20, 70, 20);
       g.fillOval(30-2, 20-2, 5, 5);
       g.fillOval(50-2, 20-2, 5, 5);
       if (closed) g.drawLine(30, 20, 50, 20);
       else g.drawLine(30, 20, 47, 10);
    }

    public void toggle() {   // Toggle the switch
       closed = !closed;
       repaint();
    }
}
```

Testing SwitchConstrained bean

When you drop `SwitchConstrained` into beanbox, you shall see the event interface "`vetoableChange`" and method "`vetoableChange`" in the "Event" menu.

BDK has a demo bean called `Vetor`, which can be used to test our `SwitchConstrained`. Read the source code of in "`$BDK\sunw\demo\misc\Vetor.java`". A `boolean` property called `vetoAll` is defined:

```
private boolean vetoAll = true;
The Vetor bean provides the following event handler for the VetoableChangeEvent:
public void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException {
   if (vetoAll) {
      throw new PropertyVetoException("NO!", evt);
   }
}
```

You can change the property vetoAll to veto or not to veto constrained property change.

The testing procedure is:

1. Start BDK. Load the `SwtichConstrained` bean. Drop an instance of `SwtichConstrained` bean and `Vetor` bean into the beanbox.
2. Focus on `SwtichConstrained` bean, select event "`vetoableChange`" and method "`vetoableChange`". Choose `Vetor` as the target of the event, and `vetoableChange()` method as the event handler.
3. Try changing the `closed` property of the `SwtichConstrained` bean.
4. Change the `vetoAll` property of the `Vetor` to `true`. Try changing the `closed` property of the `SwtichConstrained` bean again.

Indexed Property

Indexed property represents collections of values, which can be accessed by an index. Indexed property is kept in an array.

For example,

```
// Methods to access the entire indexed property
public PropertyType[] getPropertyName();
public void setPropertyName(PropertyType[] values);

// Methods to access individual item
public PropertyType getPropertyName(int index);
public void setPropertyName(int index, PropertyType value);
```

Implementing indexed property is straightforward. See BDK demo bean `OurListBox`.

Persistence (Serializable & XML)

A bean's state can be saved and later restore. All beans are serializable (i.e., implement the Serializable interface) via `ObjectInputStream` and `ObjectOutputStream`. All non-static and non-`transient` instance variable can be saved.

JDK 1.4 introduces XML persistence.

[TODO] more

Manifest for JavaBeans' JAR file

A manifest is a special file that contains information about the files packaged in a JAR file. When you create a JAR file, it automatically receives a default manifest file called "`META-INF\MANIFEST.MF`" which contains the following:

```
Manifest-Version: 1.0
Created-By: 1.5.0_04 (Sun Microsystems Inc.)
```

The entries in manifest take the form of "name: values" pair. The name and value are separated by a colon `':'`. The names are also called attributes.

To extract the manifest from a JAR file:

```
> jar xfv jarFileName META-INF\MANIFEST.MF
```

To create a JAR file with your own manifest:

```
> jar cmfv manifestFile jarFileName filesToJar
```

A manifest consists of a "main" section followed by a list of sections for individual JAR file entries. The sections are separated by a newline. The main section contains security and configuration information about the JAR file itself; and defines the main attributes that apply to every individual manifest entry. Main section should not contain attribute "Name".

Individual section defines various attributes for package or files contained in this JAR file. Each section starts with an attribute "Name", having value of a relative path to the file, or an absolute URL referencing data outside the archive.

An example of a manifest used for jarring JavaBeans is:

```
Manifest-Version: 1.0

Name: elect/Switch.class
Java-Bean: True

Name: elect/LightBulb.class
Java-Bean: True
```

Notes:

- The manifest used in BDK is incorrect. E.g., in the demo bean "buttons.jar", the main section has attribute "Name".
- The "Name" attribute specifies the location of the bean class. Hence, directory is used instead of package name. Forward slash `'/'` is used.
- The sections are separated by a newline. The last line must be terminated by a new line.
- Do not put trailing spaces behind the value.

Writing your Application Builder Tool

The following codes can be used to filter all the JAR files from a particular directory:

```
String jarDirName = "c:\\_javabin\\bdk1.1\\jars";
File jarDir = new File(jarDirName);
// filter only ".jar"
String[] jarFiles = jarDir.list(new FilenameFilter() {
   public boolean accept(File dir, String fileName) {
      return (fileName.endsWith(".jar"));
   }
});
```

The following codes can be use to set up a `JarURLClassLoader`, which is capable of loading a class or resource from all the JAR files.

```
int numFiles = jarFiles.length;
URL[] urls = new URL[numFiles];
for (int i = 0; i < numFiles; i++) {
   // Create an URL object from string
   URL url = new URL("file:" + jarDirName + "\\" + jarFiles[i]);
   urls[i] = url;
}
// URLClassLoader loads classes by searching a list of URLs.
URLClassLoader loader = new URLClassLoader(urls);
```

For each of the JAR file, the following codes can be used to read the manifest to look for JavaBeans packaged within the JAR file, and retrieve the bean name.

```
JarFile jarfile = new JarFile(jarDirName + "\\" + file);
Manifest mf = jarfile.getManifest();
String beanName = null;

// Check the manifest's main section
// BDK's demo bean put the Javabean on the main section
// which is incorrect.
Attributes attribs = mf.getMainAttributes();
if (attribs != null) {
   // Determine if this is a java bean.
   String isJavaBean = attribs.getValue(Attributes_Name_JAVA_BEAN);
   if (isJavaBean != null
       && isJavaBean.equalsIgnoreCase("True")) {
      beanName = attribs.getValue(Attributes_Name_NAME);
      processBean(beanName, loader);
   }
}

// Check the individual section for Javabean
Iterator iterator = mf.getEntries().keySet().iterator();
while (iterator.hasNext()) {
   beanName = (String) iterator.next();
   attribs = mf.getAttributes(beanName);
   if (attribs != null) {
      String isJavaBean = attribs.getValue(Attributes_Name_JAVA_BEAN);
      if (isJavaBean != null && isJavaBean.equalsIgnoreCase("True")) {
      processBean(beanName, loader);
      }
   }
}
```

The `processBead(beanName, loader)` method can be used to load the bean. We can then apply the introspection to list the properties, events, and methods available in the bean.

Introspection

Application builder tool can use the class `java.beans.Introspector` to *introspect* properties, events and methods supported by a target Javabean. For each of these three kinds of information, the Introspector will separately analyze the target bean's class and superclasses looking for either explicit or implicit information and use this information to build a `java.beans.BeanInfo` object to describe the target bean.

The explicit information of a target bean (says `Xxx`) is obtained through a companion class `XxxBeanInfo`.

If the target bean does not have a companion BeanInfo class, then the Introspector uses the low-level reflection mechanism (in `java.lang.reflect`) to study the public methods of the target class. It applies the Javabean Naming Convention to identify properties, event sources, or event handling methods of the target class and its superclasses.

The most important method in Introspector is `getBeanInfo()` which return a `BeanInfo` object to describe the target bean:

```
public static BeanInfo getBeanInfo(Class beanClass);
```

The `BeanInfo` interface declares the following abstract methods for retrieving the property, events and event handling methods:

```
public BeanDescriptor[] getBeanDescriptors()
public PropertyDescriptor[] getPropertyDescriptors()
public EventSetDescriptor[] getEventSetDescriptors()
public MethodDescriptor[] getMethodDescriptors()
public java.awt.Image getIcon(int iconKind)
  // iconKind ICON_COLOR_16x16, ICON_COLOR_32x32, ICON_MONO_16x16, ICON_MONO_32x32
```

Example:

```
static void processBean(String beanName, URLClassLoader loader) {
   // change "elect/LightBulb.class" to "elect.LightBulb"

   if (!beanName.endsWith(".class")) return;
   beanName = beanName.replace('/', '.');
   beanName = beanName.substring(0, beanName.length() - 6);
   try {
      Class beanClass = loader.loadClass(beanName);
      BeanInfo beanInfo = Introspector.getBeanInfo(beanClass);
      PropertyDescriptor[] properties = beanInfo.getPropertyDescriptors();
      for (PropertyDescriptor p : properties)
         System.out.println("  Property: " + p.getName());

      MethodDescriptor[] methods = beanInfo.getMethodDescriptors();
      for (MethodDescriptor m : methods)
         System.out.println("  Method: " + m.getName());

      EventSetDescriptor[] eventSets = beanInfo.getEventSetDescriptors();
      for (EventSetDescriptor e : eventSets)
         System.out.println("  Event: " + e.getName());

      // Image img = beanInfo.getIcon(BeanInfo.ICON_COLOR_16x16);
   } catch (ClassNotFoundException ex) {
      ex.printStackTrace();
   } catch (IntrospectionException ex) {
      ex.printStackTrace();
   }
}
```

Reflection

Read "The Java Tutorial", specialized trail on "Reflection".

`Class` (`java.lang.Class`)

Class Loader: `java.lang.ClassLoader`, `java.net.URLClassLoader`

Package: `java.lang.reflect`

Package: `java.beans`

- Introspector: `java.beans.Introspector`
- Feature Descriptors: `BeanDescriptor`, `EventSetDescriptor`, `MethodDescriptor`, `PropertyDescriptor`, `ParameterDescriptor`.
- Bean Utilites: `java.bean.Beans`
- Property change and vetoable change:
- BeanInfo:
- Editor:
- Customizer:
- Serialization: `XMLEncoder`, `XMLDecoder`.

[TODO] to be continued...

Bean Context

package: `java.beans.beancontext`.

"Bean context" is the standard mechanism through which Java developers can logically group a set of related JavaBeans into a "context" that the beans can become aware of and/or interact with. This context, or "containing environment", is known as the `BeanContext`.

There are two distinct types of `BeanContext`: one which supports membership only (interface `java.beans.beancontext.BeanContext`) and one which supports membership and offers services (interface `java.beans.beancontext.BeanContextServices`) to its JavaBeans nested within.

[TODO] to be continued...

REFERENCES & RESOURCES

- JavaBeans home page @ http://java.sun.com/products/javabeans
- JavaBeans API Specification
- "The Java Tutorial", specialized trail on "JavaBeans"
- "The Java Tutorial", specialized trail on "Jar Files"
- "The Java Tutorial", specialized trail on "Reflection"
- Bean Development Kit (BDK) 1.1 (no longer available)
- Bean Builder 0.6 alpha @ https://bean-builder.dev.java.net

Latest version tested: JDK 1.6
Last modified: August 4, 2008