Richard G Baldwin (512) 223-4758, baldwin@austin.cc.tx.us,
http://www2.austin.cc.tx.us/baldwin/

JavaBeans, A Skeleton Bean Program

Java Programming, Lecture Notes # 502, Revised 02/16/98.

Preface

Students in Prof. Baldwin's **Advanced Java Programming** classes at ACC are responsible for knowing and understanding all of the material in this lesson.

Introduction

A previous lesson provided an overview of Java Beans. This lesson expands on that overview by providing a skeleton Java Bean program that contains many of the interface characteristics required of all Beans.

In addition, this lesson provides a Java application program to test the Bean. As you will see, the test program is longer and more complex than the Bean program. The test program is included for two reasons:

- To illustrate the ease with which Beans can be incorporated into ordinary Java applications.
- To help you appreciate the simplicity of testing a Bean using the JavaSoft BeanBox which we will study in a subsequent lesson.

Even though the test program is longer and more complex, there is nothing contained in that program that you haven't seen many times in previous lessons. Therefore, very little space will be devoted to discussing the test program.

Skeleton Bean Program

This program was designed to be compiled and executed under JDK 1.1.1 in compliance with the JavaBeans specification 1.0.

A Java Bean is a *reusable software component* that can be manipulated visually in a builder tool. Creating such a component requires careful adherence to a specified interface. This program produces a *skeleton* Bean that exhibits many of the required interface characteristics and other required attributes of a Bean.

Information about the Bean interface can be provided by the programmer to the builder tool either

- by adherence to *design patterns*, or
- by publishing that information in an object of a class that implements the **BeanInfo** interface.

This program relies on *design patterns*.

Some essential information for developing Beans is listed below.

- All beans should implement the **Serializable** interface so that their state can be saved and later restored.
- *Methods* that are to be *exposed* to the builder tool and to other beans must be made **public**.
- All exposed methods should be made threadsafe (possibly **synchronized)** to prevent more than one thread from calling a method at any given time.
- *Properties* are exposed through the use of public *"set"* and *"get"* methods. Properties with no "set" method are *read-only*. Properties with no "get" method are *write-only*.
- The "get" side of a **boolean** property may be exposed either through the use of a public *"is"* method or an ordinary *"get"* method.
- Events which the bean can *multicast* are exposed through public *"add"* and *"remove"* methods.

The program was tested using JDK 1.1.1 and Win95.

The following package specification was required in order to install the Bean in the JavaSoft **BeanBox**. The BeanBox is a Java application that can be used to test Beans. The package specification may change depending on where the Bean is installed in the directory structure.

```
package sunw.demo.beans01;
```

Interesting Code Fragments

The first interesting code fragment is the first line in the class definition showing that the class implements the **Serializable** interface. All Beans should implement this interface so that their state can be saved and later restored.

```
public class Beans01 extends Canvas
                                implements Serializable{
```

The next interesting code fragment is the declaration of the pair of instance variables that will later combine with methods to become *properties*. According to the Java Beans Specification from JavaSoft,

"Properties are discrete, named attributes of a Java Bean that can affect its appearance or its behaviour."

For example, a GUI button might have a property named "Label" that represents the text displayed in the button. However, the **String** object that constitutes the Label may or may not be maintained in an instance variable with the same name.

A bean's properties will usually be *persistent*. That is, their current state can be stored away as part of the *persistent* state of the bean. This is one reason that the class of the Bean object should implement the **Serializable** interface.

Properties can have arbitrary types, including primitive types such as **int** and class or interfaces types such as **java.awt.Color**.

As you can see, the instance variables that are used to maintain property values in this skeleton program are of type **Color** and **boolean**.

```
  protected Color myColor;
  protected boolean myBooleanInstanceVariable = true;
```

The constructor for the Bean is pretty straightforward except that it contains a call to a method named **sizeToFit()**. This method is interesting because it is required to cause the Bean to appear at the proper size when placed in the BeanBox, but is not required for that purpose when the Bean is placed in an ordinary Frame object.

As of this writing, I have not determined why it is required in one case and not required in the other case. The **sizeToFit()** method follows. Since the body of this method is really more germane to *graphics* than to Beans, an analysis of the body of the method will be left as an exercise for the student.

(Update Notice: April 13, 1997. The apparent bug in the February 1997 version of the BDK 1.0 that required use of the **sizeToFit()** method has been eliminated in the April 1997 method. When using the April version, this method is no longer required.)

Note that an auxiliary method, **getPreferredSize()** that specifies the display size of the Bean is also shown in this code fragment.

```
private void sizeToFit () {
    this.setSize(getPreferredSize());
    Component p = getParent();
    if (p != null) {
```

```
      p.invalidate();
      p.doLayout();
    }//end if
  }//end sizeToFit()

  public synchronized Dimension getPreferredSize(){
    return new Dimension(50,50);
  }//end getPreferredSize()
```

The next interesting code fragment is the pair of methods that, when combined with the instance variable named **myBooleanInstanceVariable**, constitute the property named **myBooleanProperty**. Note that in the special case of boolean properties, you can use either a "get" method or an "is" method (or both) to satisfy the "get" side of the *design pattern* for properties. In this case, we use an "is" method.

```
  public synchronized boolean isMyBooleanProperty(){
    return myBooleanInstanceVariable;
  }//end isMyBooleanProperty()

  public synchronized void setMyBooleanProperty(
                                          boolean data){
    myBooleanInstanceVariable = data;
  }//end setMyBooleanProperty
```

The next interesting code fragment is the pair of "set" and "get" methods which, along with the instance variable named **myColor**, constitute the property named **color**.

```
  public synchronized void setColor(Color inColor){
    myColor = inColor;
    this.setBackground(myColor);
  }//end setColor()

  public synchronized Color getColor(){
    return myColor;
  }//end getColor
```

The following two methods are exposed to the builder tool as accessible methods by virtue of the fact that they are public, and no explicit instructions as to which methods are accessible were provided. Thus, a builder tool will consider these methods as being available for being invoked as a result of events emitted by other Beans.

```
  public synchronized void makeBlue(){
    myColor = Color.blue;
    this.setBackground(myColor);
  };//end makeBlue()

  public synchronized void makeRed(){
```

```
    myColor = Color.red;
    this.setBackground(myColor);
  };//end makeRed()
```

The following pair of "add" and "remove" **Listener** methods expose the fact that this Bean has the ability to *multicast* **Action** events.

Note, however, that the body of these methods is incomplete for brevity and they are not really capable of *multicasting* such events. The builder tool identifies the capability to *multicast* events using the "add" and "remove" *design pattern* and has no way to determine if the body of the methods really supports the claim made by the *design pattern*.

```
  public synchronized void addActionListener(
                                          ActionListener e){
    //Incomplete.  Need to put some substance here
    System.out.println(
                "addActionListener not fully implemented");
  }//end addActionListener()

  public synchronized void removeActionListener(
                                          ActionListener e){
    //Incomplete.  Need to put some substance here
    System.out.println(
              "removeActionListener not fully implemented");
  }//end removeActionListener
```

These are the interesting code fragments for this skeleton Bean program. A complete listing of the program is provided in the next section.

Listing of Skeleton Bean Program

This section contains a complete listing of the skeleton Bean program with additional comments.

```
/*File Beans01.java.java Copyright 1997, R.G.Baldwin
This program was designed to be compiled and executed
under JDK 1.1.1.

A Java Bean is a reusable software component that can be
manipulated visually in a builder tool.  This requires
careful adherence to a specified interface.

This program produces a skeleton Bean that exhibits many
of the required interface characteristics of a Bean.

Information about the interface to the bean can be
provided to the builder tool either by adherence to design
patterns, or by publishing that information in an object
of a class that implements the BeanInfo interface.  This
program relies on design patterns.
```

```
Some guidelines are:

All beans should implement the Serializable interface so
that their state can be saved and later restored.

Methods that are to be exposed to the builder tool and to
other beans must be made public.

All exposed methods should be synchronized to prevent more
than one thread from calling a method at any given time.

Properties are exposed through the use of public
"set"/"get" methods. Properties with no "set" method are
read-only.  Properties with no "get" method are write-only.

The "get" side of a Boolean property may be exposed either
through the use of a public "is" methods or an ordinary
"get" method.

Events which the bean can multicast are exposed through
public "add"/"remove" methods.

The program was tested using JDK 1.1.1 and Win95.
*****************************************************/

//The following package specification is required in order
// to install the Bean in the BeanBox.  The package
// specification changes depending on where the Bean is
// installed in the directory structure.
//package sunw.demo.beans01;

import java.awt.event.*;
import java.awt.*;
import java.io.Serializable;
//=====================================================//
//All beans should implement the Serializable interface
public class Beans01 extends Canvas
                                    implements Serializable{

  //The following two instance variables are used for
  // properties
  protected Color myColor;
  protected boolean myBooleanInstanceVariable = true;

  public Beans01(){//constructor
    myColor = Color.yellow;
    setBackground(myColor);
    sizeToFit();//see discussion below
  }//end constructor

  //The following method and the above call to the method
  // in the constructor are required to cause the bean to
  // be the correct size in the BeanBox.  It is not
  // required when the Bean is tested in an ordinary Frame
  // object.  The reason for the difference hasn't been
  // determined.
```

```
  private void sizeToFit () {
    this.setSize(getPreferredSize());
    Component p = getParent();
    if (p != null) {
      p.invalidate();
      p.doLayout();
    }//end if
  }//end sizeToFit()

  //This method defines the display size of the object.
  public synchronized Dimension getPreferredSize(){
    return new Dimension(50,50);
  }//end getPreferredSize()

  //The following "set" and "is" methods in conjunction
  // with the instance variable myBooleanInstanceVariable
  // constitute a boolean property. For boolean properties,
  // either a "get" method or an "is" method will support
  // the design pattern requirement.
  public synchronized boolean isMyBooleanProperty(){
    return myBooleanInstanceVariable;
  }//end isDummyInstanceVariable()

  public synchronized void setMyBooleanProperty(
                                          boolean data){
    myBooleanInstanceVariable = data;
  }//end setMyBooleanProperty

  //The following "set" and "get" methods in conjunction
  // with the instance variable myColor constitute a
  // property of type Color.
  public synchronized void setColor(Color inColor){
    myColor = inColor;
    this.setBackground(myColor);
  }//end setColor()

  public synchronized Color getColor(){
    return myColor;
  }//end getColor

  //The following two methods are exposed to the builder
  // tool as accessible methods.
  public synchronized void makeBlue(){
    myColor = Color.blue;
    this.setBackground(myColor);
  };//end makeBlue()

  public synchronized void makeRed(){
    myColor = Color.red;
    this.setBackground(myColor);
  };//end makeRed()

  //The following two methods expose to the builder tool
  // the fact that this Bean is able to multicast Action
  // events (but the methods are incomplete for brevity).
  public synchronized void addActionListener(
```

```
                                          ActionListener e){
    //Incomplete.  Need to put some substance here
    System.out.println(
                "addActionListener not fully implemented");
  }//end addActionListener()

  public synchronized void removeActionListener(
                                          ActionListener e){
    //Incomplete.  Need to put some substance here
    System.out.println(
              "removeActionListener not fully implemented");
  }//end removeActionListener

}//end class Beans01.java
```

Listing of Test Program

The following Java application can be used to test the interface of the skeleton Bean program presented above. There is nothing in this application that hasn't been covered in depth in previous lessons, so no analysis of the program, other than that provided in the comments, will be provided.

A subsequent lesson will explain how to also test the skeleton Bean program using the JavaSoft BeanBox.

```
/*File Beans01Test.java.java Copyright 1997, R.G.Baldwin
This program was designed to be compiled and executed
under JDK 1.1.1.

The purpose is to test the bean named Beans01.java in a
Frame.

A Bean01 object is placed in the frame along with eight
buttons. The visual manifestation of the Bean object is a
colored square.

One pair of buttons exercises the "get" and "set" color
properties of the Bean.

A second pair of buttons invokes the makeRed() and
makeBlue() methods of the Bean.

A third pair of buttons invokes the addActionListener()
and removeActionListener() methods of the Bean.

A fourth pair of buttons exercises the "set" and "is"
boolean properties of the Bean.

For those cases where information is returned from the
```

```
Bean, it is displayed on the standard output device.

The program was tested using JDK 1.1.1 and Win95.
**********************************************************/

import java.awt.*;
import java.awt.event.*;
//========================================================//
public class Beans01Test extends Frame{
  public static void main(String[] args){
    new Beans01Test();
  }//end main

  public Beans01Test(){//constructor
    setTitle("Copyright 1997, R.G.Baldwin");
    setLayout(new FlowLayout());
    Beans01 myBean = new Beans01();//instantiate Bean obj
    add(myBean);//Add it to the Frame

    //Instantiate several test buttons
    Button buttonToSetColor = new Button(
                                      "set color property");
    Button buttonToGetColor = new Button(
                                      "get color property");
    Button buttonToInvokeRedMethod = new Button(
                                   "Invoke makeRed Method");
    Button buttonToInvokeBlueMethod = new Button(
                                  "Invoke makeBlue Method");
    Button buttonToAddActionListener = new Button(
                                    "Add Action Listener");
    Button buttonToRemoveActionListener = new Button(
                                  "Remove Action Listener");
    Button buttonToSetBooleanProperty = new Button(
                                    "Set boolean Property");
    Button buttonToGetBooleanProperty = new Button(
                                    "Get boolean Property");

    //Add the test buttons to the frame
    add(buttonToSetColor);
    add(buttonToGetColor);
    add(buttonToInvokeRedMethod);
    add(buttonToInvokeBlueMethod);
    add(buttonToAddActionListener);
    add(buttonToRemoveActionListener);
    add(buttonToSetBooleanProperty);
    add(buttonToGetBooleanProperty);

    setSize(250,350);
    setVisible(true);

    //Register action listener objects for all the test
    // buttons
    buttonToSetColor.addActionListener(
                              new SetColorListener(myBean));
    buttonToGetColor.addActionListener(
                              new GetColorListener(myBean));
```

```
    buttonToInvokeRedMethod.addActionListener(
                              new RedActionListener(myBean));
    buttonToInvokeBlueMethod.addActionListener(
                             new BlueActionListener(myBean));
    buttonToAddActionListener.addActionListener(
                            new AdditActionListener(myBean));
    buttonToRemoveActionListener.addActionListener(
                              new RmovitActionLstnr(myBean));
    buttonToSetBooleanProperty.addActionListener(
                               new SetBoolPropLstnr(myBean));
    buttonToGetBooleanProperty.addActionListener(
                               new GetBoolPropLstnr(myBean));

    //terminate when frame is closed
    this.addWindowListener(new Terminate());
  }//end constructor
}//end class Beans01Test.java
//=======================================================//
//The following two classes are used to instantiate
// objects to be registered to listen to two of the
// buttons on the test panel.  When the
// buttonToSetBooleanProperty is pressed, the boolean
// property is set to false. When the
// buttonToGetBooleanProperty is pressed, the current
// boolean property is displayed on the standard output
// device.

class SetBoolPropLstnr implements ActionListener{
  Beans01 myBean;

  SetBoolPropLstnr(Beans01 inBean){//constructor
    myBean = inBean;
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.setMyBooleanProperty(false);
  }//end actionPerformed()
}//end class SetBoolPropLstnr
//=======================================================//
class GetBoolPropLstnr implements ActionListener{
  Beans01 myBean;

  GetBoolPropLstnr(Beans01 inBean){//constructor
    myBean = inBean;
  }//end constructor

  public void actionPerformed(ActionEvent e){
    System.out.println(myBean.isMyBooleanProperty());
  }//end actionPerformed()
}//end class GetBoolPropLstnr

//=======================================================//

//The following two classes are used to instantiate
// objects to be registered to listen to two of the
// buttons on the test panel.  When the setColor button is
```

```
// pressed, the Color property is set to green. When the
// getColor button is pressed, the current color is
// displayed on the standard output device.
class SetColorListener implements ActionListener{
  Beans01 myBean;

  SetColorListener(Beans01 inBean){//constructor
    myBean = inBean;
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.setColor(Color.green);
  }//end actionPerformed()
}//end class SetColorListener
//=======================================================//
class GetColorListener implements ActionListener{
  Beans01 myBean;

  GetColorListener(Beans01 inBean){//constructor
    myBean = inBean;
  }//end constructor

  public void actionPerformed(ActionEvent e){
    System.out.println(myBean.getColor().toString());
  }//end actionPerformed()
}//end class GetColorListener

//=======================================================//
//The following class is used to instantiate a dummy
// ActionListener object which is passed to the
// addActionListener() and removeActionListener() methods
// of the Bean.
class MyDummyActionListener implements ActionListener{
  public void actionPerformed(ActionEvent e){
  }//end empty actionPerformed() method
}//end class MyDummyActionListener
//=======================================================//
//The following two classes are used to instantiate
// objects to be registered to listen to two of the
// buttons on the test panel.  When the buttons are
// pressed, the addActionListener() and
// removeActionListener() methods of the Bean are invoked.
class AdditActionListener implements ActionListener{
  Beans01 myBean;

  AdditActionListener(Beans01 inBean){//constructor
    myBean = inBean;
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.addActionListener(new MyDummyActionListener());
  }//end actionPerformed()
}//end class AdditActionListener
//=======================================================//
class RmovitActionLstnr implements ActionListener{
  Beans01 myBean;
```

```
  RmovitActionLstnr(Beans01 inBean){//constructor
    myBean = inBean;
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.removeActionListener(
                               new MyDummyActionListener());
  }//end actionPerformed()
}//end class AdditActionListener
//=======================================================//
//The following two classes are used to instantiate
// objects to be registered to listen to two of the
// buttons on the test panel.  When the buttons are
// pressed, these objects invoke methods of the Bean under
// test. The first class invokes the makeRed() method and
// the second class invokes the makeBlue() method.
class RedActionListener implements ActionListener{
  Beans01 myBean;

  RedActionListener(Beans01 inBean){//constructor
    myBean = inBean;
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.makeRed();
  }//end actionPerformed()
}//end class RedActionListener
//=======================================================//
class BlueActionListener implements ActionListener{
  Beans01 myBean;

  BlueActionListener(Beans01 inBean){//constructor
    myBean = inBean;
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.makeBlue();
  }//end actionPerformed()
}//end class RedActionListener
//=======================================================//
class Terminate extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    System.exit(0);//terminate the program
  }//end windowClosing
}//end class Terminate
//=======================================================//
```

-end-