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

JavaBeans, Properties of Beans, Bound Properties

Java Programming, Lecture Notes # 510, Revised 02/19/98.

- Preface
- Introduction
- Introspection on the Bean Class
- Properties
- Sample Bean Program
  - Interesting Code Fragments from the Bean Program
  - Program Listing for the Bean Program
- Sample Test Program
  - Interesting Code Fragments from the Test Program
  - Program Listing for the Test Program
- Using the PropertyChangeSupport Class
- Review

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.

The material in this lesson is extremely important. However, there is simply too much material to be covered in detail during lecture periods. Therefore, students in Prof. Baldwin's **Advanced Java Programming** classes at ACC will be responsible for studying this material on their own, and bringing any questions regarding the material to class for discussion.

This lesson was originally written on October 18, 1997 using the software and documentation in the JDK 1.1.3 download package along with the April 97 release of the BDK 1.0 download package.

Introduction

According to the document from JavaSoft entitled Using the Beans Development Kit 1.0 April 1997 A Tutorial:

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.

In this lesson, we will develop and test a **bean** class, named **Beans03**, that illustrates the use of *bound* properties.

This **bean** class is examined with the **Introspector** and is tested with a test program designed specifically to exercise the ability of the bean to broadcast an event to a list of registered **listener** objects whenever the value of one of the properties changes.

Introspection on the Bean Class

One way to describe a **bean** is to describe it from the viewpoint of a Visual Builder Tool (VBT) using **introspection**. In an earlier lesson we developed a Java application named **Introspect01** that can be used to apply **introspection** to a bean class and to record the results in a temporary disk file. The following box shows the results of **introspection** on the sample bean class that was developed for this lesson.

```
Name of bean:  Beans03
Class of bean: class Beans03

==== Properties: ====
Name: preferredSize
 Type:        class java.awt.Dimension
 Get method: public synchronized java.awt.Dimension
                                 Beans03.getPreferredSize()
 Set method: null
Name: theDate
 Type:        class java.util.Date
 Get method: null
 Set method: public synchronized void
                         Beans03.setTheDate(java.util.Date)
Name: theColor
 Type:        class java.awt.Color
 Get method: public synchronized java.awt.Color
                                      Beans03.getTheColor()
 Set method: public synchronized void
                        Beans03.setTheColor(java.awt.Color)

==== Events: ====
Event Name: propertyChange
 Add Method:    public synchronized void
             Beans03.addPropertyChangeListener(
                         java.beans.PropertyChangeListener)
 Remove Method: public synchronized void
             Beans03.removePropertyChangeListener(
                         java.beans.PropertyChangeListener)
 Event Type: propertyChange

==== Methods: ====
makeRed
setTheDate
removePropertyChangeListener
getTheColor
setTheColor
```

```
getPreferredSize
addPropertyChangeListener
makeBlue
```

The primary functional difference between this **bean** class and **bean** classes developed in previous lessons is shown in the highlighted section entitled **Events**. This section shows the *add* and *remove* methods which provide the ability of the **bean** to notify a list of listener objects whenever a change occurs in one of the property values. (Note that this bean does not notify listeners of changes in the value of the property named **preferredSize** but it does notify listeners of changes in the values of the other two properties.)

Properties

There are four kinds of properties:

- Simple
- Indexed
- Bound
- Constrained

A previous lesson discussed *Simple* and *Indexed* properties. This lesson will concentrate on *Bound* properties. *Constrained* properties will be discussed in a future lesson.

The **bean** class developed for this lesson has the following *Bound* properties:

```
Name: theDate
 Type:        class java.util.Date
 Get method: null
 Set method: public synchronized void
                        Beans03.setTheDate(java.util.Date)
Name: theColor
 Type:        class java.awt.Color
 Get method: public synchronized java.awt.Color
                                   Beans03.getTheColor()
 Set method: public synchronized void
                        Beans03.setTheColor(java.awt.Color)
```

The bean maintains a list of objects that request to be notified whenever there is a change in the value of either of the *Bound* properties. This is often referred to as registering **listener** objects to receive an event.

Objects that request to be added to the list must be of a class that implements the **PropertyChangeListener** interface and defines the **propertyChange()** method that is declared in that interface.

Whenever a value is assigned to the instance variables used to maintain these properties (regardless of whether or not it is a different value) an event is broadcast to all of the listener objects registered on the list.

The event is broadcast by invoking the **propertyChange()** method on all of the objects on the list. When this method is invoked on a **listener** object, an object of type **PropertyChangeEvent** is passed as a parameter.

The **PropertyChangeEvent** object passed as a parameter to the **propertyChange()** method in the listener objects contains the following information:

- Object source, //the bean object in this case
- String propertyName, //the name of the changed property
- Object oldValue, //the old value of the changed property
- Object newValue //the new value of the changed property

As you can see, three of the parameters passed to the **propertyChange()** method are of type **Object**, and one is of type **String**. The parameters of type **Object** sometimes need to be downcast to the correct type to be used in the receiving method.

Notification of the **listener** objects takes place <u>after</u> the change in the property value has occurred. This bean class does not save the *old* value when it assigns a new value. As a result, it passes **null** as the *old* value of the changed property because the *old* value is no longer available when the listener objects are notified that a change has occurred.

The following methods are available to extract information from the object passed as a parameter. These methods are defined by the **PropertyChangeEvent** class or its superclass, **EventObject**:

- public Object getSource();
- public Object getNewValue();
- public Object getOldValue();
- public String getPropertyName;
- public void setPropagationId();
- public Object getPropagationId;

The test program (named **Beans03Test**) used in this lesson to partially test the **bean** class uses the first four of these methods. Apparently the PropagationID is reserved for future use.

Sample Bean Program

In this lesson, we will deal with two different programs. One program is a program that creates a **bean** class named **Beans03.java**.. The other program is a program used to partially test the **bean** named **Beans03Test.java**. This section deals with the program used to create the **bean**.

This program was designed to be compiled and executed under JDK 1.1.3. It was tested using JDK 1.1.3 and the Apr97 version of the BDK 1.0 under Win95.

The purpose of this **bean** class is to illustrate *bound* properties.

This bean contains two bound properties: a **Color** property named **theColor** and a **Date** property named **theDate**. The **Date** property is a *write-only* property because no *get* method is provided for this property. (The bean class also contains a read-only property named **preferredSize** which resulted from providing that information for the benefit of the layout manager.)

Note that as of this writing in October of 1997, this **bean** class has not been tested in the BeanBox. It was tested using the test program named **Beans03Test**.

Any of the three methods in the following list can be invoked to assign a new value to the property named **theColor.** Whenever any of these methods are invoked, and after the new value is assigned to the property, a **PropertyChangeEvent** is broadcast to all registered **listener** objects.

```
setTheColor()
makeRed()
makeBlue()
```

Only the one method listed below is available to assign a new value to the property named **theDate.** Whenever this method is invoked, a **PropertyChangeEvent** is broadcast to all registered **listener** objects.

```
setTheDate()
```

With a little extra programming effort, it would have been possible to compare the new values being assigned to the properties with the old values and to broadcast events only when the value actually changed. This probably would have been more in keeping with the philosophy of a **PropertyChangeEvent**.

Interesting Code Fragments from the Bean Program

This **bean** class contains a number of interesting code fragments. The following fragment shows the declaration of two reference variables used to maintain the *bound* properties and also shows the instantiation of a **Vector** object that is used to maintain the list of registered **listener** objects.

```
  //The following instance variables are used to store
  // property values.
  protected Color myColor;
  protected Date myDate;
```

```
//The following Vector is used to maintain a list of
// listeners who request to be notified of changes in the
// property values.
protected Vector propChangeListeners = new Vector();
```

The following *set* and *get* methods, used in conjunction with the instance variable named **myColor** constitute a property named **theColor**. Note in particular the highlighted statement that causes a **PropertyChangeEvent** to be broadcast each time the **setTheColor()** method is invoked.

```
public synchronized void setTheColor(Color inColor){
  myColor = inColor;
  this.setBackground(myColor);
  //notify property listeners of property change
  notifyPropertyChange("theColor");
}//end setTheColor()

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

The following *set* method used in conjunction with the instance variable named **myDate** constitutes a *write-only* property named **theDate**. Again, note the highlighted statement that causes a **PropertyChangeEvent** to be broadcast each time the **setTheDate()** method is invoked.

```
public synchronized void setTheDate(Date dateIn){
  myDate = dateIn;
  //notify property listeners of property change
  notifyPropertyChange("theDate");
}//end setTheDate()
```

The following two methods are exposed to the builder tool as accessible methods for switching the value of the property named **theColor** between *blue* and *red*. Again, note the highlighted statements that cause a **PropertyChangeEvent** to be broadcast each time either of these methods is invoked.

```
public synchronized void makeBlue(){
  myColor = Color.blue;
  this.setBackground(myColor);
  //notify property listeners of property change
  notifyPropertyChange("theColor");
}//end makeBlue()

public synchronized void makeRed(){
  myColor = Color.red;
  this.setBackground(myColor);
  //notify property listeners of property change
  notifyPropertyChange("theColor");
}//end makeRed()
```

The following two methods are used to maintain a list of registered **listener** objects who request to be notified of changes to the properties, or who request to be removed from the list of registered **listener** objects. These methods are consistent with the behavior of the *Delegation*

Event Model operating in a *multicast* mode.

```
//Add a property change listener object to the list.
public synchronized void addPropertyChangeListener(
                        PropertyChangeListener listener){
  //If the listener is not already registered, add it
  // to the list.
  if(!propChangeListeners.contains(listener)){
    propChangeListeners.addElement(listener);
  }//end if
}//end addPropertyChangeListener
//-----------------------------------------------------//

//Remove a property change listener from the list.
public synchronized void removePropertyChangeListener(
                        PropertyChangeListener listener){
  //If the listener is on the list, remove it
  if(propChangeListeners.contains(listener)){
    propChangeListeners.removeElement(listener);
  }//end if
}//end removePropertyChangeListener
```

The following method is invoked by several of the methods described above, and is used to notify listener objects of changes in the properties. The incoming parameter is the name of the property that has changed.

In this case, there are only two *bound* properties. The incoming property name is used in a decision tree to determine which of the *bound* properties has changed in order to determine the values to be encapsulated in the **PropertyChangeEvent** object that is passed as a parameter when the registered **listener** objects are notified of the change.

Note that objects maintained in a **Vector** object are always of type **Object**. This leads to the need to downcast the objects in the list from **Object** to **PropertyChangeListener**.

```
protected void notifyPropertyChange(
                                 String changedProperty){
  //Instantiate the event object containing information
  // about the property that has changed.
  PropertyChangeEvent event;
  if(changedProperty.compareTo("theColor") == 0)
    //Change was in theColor property
    event = new PropertyChangeEvent(
                     this,changedProperty,null,myColor);
  else//Change was in the theDate property
    event = new PropertyChangeEvent(
                      this,changedProperty,null,myDate);

  //Make a working copy of the list that cannot be
  // modified while objects on the list are being
```

```
    // notified of the change.
    Vector tempList;
    synchronized(this){
      tempList = (Vector)propChangeListeners.clone();
    }//end synchronized block

    //Notify all listener objects on the list.  Note the
    // requirement to downcast the objects in the list from
    // Object to PropertyChangeListener.
    for(int cnt = 0; cnt < tempList.size();cnt++){
      PropertyChangeListener theListener =
           (PropertyChangeListener)tempList.elementAt(cnt);
      //Invoke the propertyChange() method on theListener
      theListener.propertyChange(event);
    }//end for loop
  }//end notifyPropertyChange
}//end class Beans03.java
```

A consolidated listing of the entire **bean** class is provided in the next section.

Program Listing for the Bean Program

This section contains a consolidated listing of the **bean** class.

```
/*File Beans03.java Copyright 1997, R.G.Baldwin
This program was designed to be compiled and executed
under JDK 1.1.3 or later.

The main purpose of this program is to illustrate the use
of "bound" properties in Beans.

This is a "bean" class that satisfies the interface
requirements for beans using design patterns.

This bean class has two properties named theColor and
theDate.

The current value of the property named theColor is stored
in the instance variable named myColor.

The current value of the property named theDate is stored
in the instance variable named myDate.  theDate is a
write-only property because it has a "set" method but does
not have a "get" method.

The program maintains a list of objects that request to be
notified whenever there is a change in the value of either
of the properties.  Whenever the value of either property
changes, all of the objects on the list are notified of the
change by invoking their propertyChange() method and
passing an object of type PropertyChangeEvent as a
parameter.

Objects that request to be added to the list must be of a
```

```
class that implements the PropertyChangeListener interface
and defines the propertyChange() method that is declared
in that interface.

The PropertyChangeEvent object passed as a parameter to the
propertyChange() method in the listener objects contains
the following information:

  Object source, //the bean object
  String propertyName, //the name of the changed property
  Object oldValue, //the old value of the changed property
  Object newValue  //the new value of the changed property

This program doesn't save the old value and therefore
passes null as the old value of the changed property
because the old value is not available when the listener
objects are notified.

The following methods are available to extract information
from the object passed as a parameter.  These methods are
defined by the PropertyChangeEvent class or its superclass
named EventObject:

  public Object getSource();
  public Object getNewValue();
  public Object getOldValue();
  public String getPropertyName;
  public void setPropagationId();
  public Object getPropagationId;

Apparently the PropagationID is reserved for future use.

The program was compiled and tested under JDK 1.1.3
and Win95.  Another program named Beans03Test.java was
used to test the bean.  It was not tested in the BeanBox.
//=========================================================//
*/

import java.awt.event.*;
import java.awt.*;
import java.io.Serializable;
import java.util.*;
import java.beans.*;
//=========================================================//
//All beans should implement the Serializable interface
public class Beans03 extends Canvas
                                   implements Serializable{

  //The following instance variables are used to store
  // property values.
  protected Color myColor;
  protected Date myDate;

  //The following Vector is used to maintain a list of
  // listeners who request to be notified of changes in the
  // property values.
  protected Vector propChangeListeners = new Vector();
  //-----------------------------------------------------//

  public Beans03(){//constructor
    //This bean is a visible square that is initialized to
    // yellow and can then be changed to green, red, and
    // blue by invoking methods of the class.
    myColor = Color.yellow;
    setBackground(myColor);
  }//end constructor
  //-----------------------------------------------------//

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

  //The following "set" and "get" methods in conjunction
  // with the instance variable named myColor constitute a
  // property named theColor.
  public synchronized void setTheColor(Color inColor){
    myColor = inColor;
    this.setBackground(myColor);
    //notify property listeners of property change
    notifyPropertyChange("theColor");
  }//end setTheColor()

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

  //The following "set" method in conjunction with the
  // instance variable named myDate constitute a write-only
  // property named theDate.
  public synchronized void setTheDate(Date dateIn){
    myDate = dateIn;
    //notify property listeners of property change
    notifyPropertyChange("theDate");
  }//end setTheDate()
  //-----------------------------------------------------//

  //The following two methods are exposed to the builder
  // tool as accessible methods.
  public synchronized void makeBlue(){
    myColor = Color.blue;
    this.setBackground(myColor);
    //notify property listeners of property change
    notifyPropertyChange("theColor");
  }//end makeBlue()

  public synchronized void makeRed(){
    myColor = Color.red;
    this.setBackground(myColor);
    //notify property listeners of property change
    notifyPropertyChange("theColor");
  }//end makeRed()
  //--------------------------------------------------------//

  //The following two methods are used to maintain a list
  // of listener objects who request to be notified of
  // changes to the properties or who request to be removed
  // from the list.

  //Add a property change listener object to the list.
  public synchronized void addPropertyChangeListener(
                        PropertyChangeListener listener){
    //If the listener is not already registered, add it
    // to the list.
    if(!propChangeListeners.contains(listener)){
      propChangeListeners.addElement(listener);
    }//end if
  }//end addPropertyChangeListener

  //Remove a property change listener from the list.
  public synchronized void removePropertyChangeListener(
                        PropertyChangeListener listener){
    //If the listener is on the list, remove it
    if(propChangeListeners.contains(listener)){
      propChangeListeners.removeElement(listener);
    }//end if
  }//end removePropertyChangeListener
  //--------------------------------------------------------//

  //The following method is used to notify listener
  // objects of changes in the properties.  The incoming
  // parameter is the name of the property that has
  // changed.
  protected void notifyPropertyChange(
                                 String changedProperty){
    //Instantiate the event object containing information
    // about the property that has changed.
    PropertyChangeEvent event;
    if(changedProperty.compareTo("theColor") == 0)
      //Change was in theColor property
      event = new PropertyChangeEvent(
                  this,changedProperty,null,myColor);
    else//Change was in the theDate property
      event = new PropertyChangeEvent(
                   this,changedProperty,null,myDate);

    //Make a working copy of the list that cannot be
    // modified while objects on the list are being
    // notified of the change.
    Vector tempList;
    synchronized(this){
      tempList = (Vector)propChangeListeners.clone();
    }//end synchronized block

    //Notify all listener objects on the list.  Note the
    // requirement to cast the objects in the list from
    // Object to PropertyChangeListener.
    for(int cnt = 0; cnt < tempList.size();cnt++){
      PropertyChangeListener theListener =
           (PropertyChangeListener)tempList.elementAt(cnt);
      //Invoke the propertyChange() method on theListener
      theListener.propertyChange(event);
    }//end for loop
  }//end notifyPropertyChange
}//end class Beans03.java
//=========================================================//
```

Sample Test Program

A special test program named **Beans03Test** was written to partially test the class named **Beans03** with special emphasis on the use of *bound* properties. The program was tested using JDK 1.1.3 under Win95.

The purpose of this program is to provide the ability to test the bean class named **Beans03** in a Frame.

A **Beans03** object is placed in the frame along with five buttons. The visual manifestation of the **bean** object is a colored square.

Two of the buttons exercise the "get" and "set" methods used to get and set the **Color** value stored in the property named **theColor**.

One button exercises the "set" method used to set the date and time in a write-only **Date** property named **theDate**.

Two of the buttons invoke the **makeRed()** and **makeBlue()** methods of the **bean** which modify the value of the property named **theColor**.

Two **listener** objects are instantiated and registered to be notified by the **bean** whenever there is a change in the value of either of the bound properties. Actually the objects are notified whenever a value is assigned to the instance variables that maintain the property values regardless of whether or not the new value is different from the old value.

For those cases where information is returned from the **bean**, it is displayed on the standard output device.

Clicking the button labeled *"Set theColor property"* produced the following output on the screen. As you can see, the two different listener objects were notified of the same change in the property named **theColor**.

In this case, the value of the **theColor** property was changed to **green** as indicated by "**g=255**". The actual color is specified by percentage contributions from red, green, and blue, where the maximum contribution of any one of the primary colors is 255 and the minimum contribution is 0. In this case, both **red** and **blue** are showing a contribution of 0.

The *Old property value* is shown as **null** because this particular **bean** class doesn't save and return the value of the property that existed before the change occurred. The event doesn't happen until after the change has occurred and the old value is no longer available at that point in time.

```
FirstListener notified of change
Property change source: Beans03[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=0,g=255,b=0]
Old property value: null

SecondListener notified of change
Property change source: Beans03[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=0,g=255,b=0]
Old property value: null
```

Clicking the button labeled *"Get theColor property"* produced the following output on the screen. Since this action didn't cause the property values to change, the **listener** objects registered to listen for changes in property values were not notified of this action.

```
java.awt.Color[r=0,g=255,b=0]
```

Clicking the button labeled *"Invoke the makeRed Method"* produced the following output on the screen. Again both listener objects were notified of the change in the property named **theColor** with the new value being **red**.

```
FirstListener notified of change
Property change source: Beans03[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=255,g=0,b=0]
Old property value: null

SecondListener notified of change
Property change source: Beans03[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=255,g=0,b=0]
Old property value: null
```

Clicking the button labeled *"Invoke the makeBlue Method"* produced the following output on the screen similar to that produced by invoking the **makeRed()** method described above, except that the **theColor** property was changed to **blue**.

```
FirstListener notified of change
Property change source: Beans03[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=0,g=0,b=255]
Old property value: null

SecondListener notified of change
Property change source: Beans03[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=0,g=0,b=255]
Old property value: null
```

Finally, clicking the button labeled *"Set theDate property"* produced the following output on the screen. In this case, both listener objects were notified and the information encapsulated in the event object identified the changed property as the property named **theDate** with a new value as shown.

```
FirstListener notified of change
Property change source: Beans03[canvas0,31,33,50x50]
Property name: theDate
New property value: Sat Oct 18 10:24:49 CDT 1997
Old property value: null

SecondListener notified of change
Property change source: Beans03[canvas0,31,33,50x50]
Property name: theDate
New property value: Sat Oct 18 10:24:49 CDT 1997
Old property value: null
```

Interesting Code Fragments from the Test Program

Much of the code in this application has been seen many times before in these lessons. In this section, we will emphasize the code that is used to test *bound* properties.

The beginning of the interesting code appears at the point where we register action listener objects for all of the buttons on the test panel. Although this code is straightforward, it will be useful to show it here to establish the basis for further discussion.

```
    buttonToSetTheColor.addActionListener(
                          new SetTheColorListener(myBean));
    buttonToGetTheColor.addActionListener(
                          new GetTheColorListener(myBean));
    buttonToInvokeRedMethod.addActionListener(
                            new RedActionListener(myBean));
    buttonToInvokeBlueMethod.addActionListener(
                               new BlueActionListener(myBean));
    buttonToSetTheDate.addActionListener(
                               new DateActionListener(myBean));
```

Next, we instantiate and register two different **PropertyChangeListener** objects to listen for changes in the **bean's** properties. These are identical objects. We instantiated and registered two of them simply to confirm proper operation of the *event multicasting* capability of the **bean**.

```
    MyPropertyChangeListener firstListener =
                                new MyPropertyChangeListener();
    //Store an identifying name in the listener object
    firstListener.setTheID("FirstListener");
    myBean.addPropertyChangeListener(firstListener);

    MyPropertyChangeListener secondListener =
                                new MyPropertyChangeListener();
    //Store an identifying name in the listener object
    secondListener.setTheID("SecondListener");
    myBean.addPropertyChangeListener(secondListener);
```

In its final form, the test program does not test the ability of the **bean** to remove listener objects from the registration list. Two statements are included in the program as comments which can be used to test this capability. When one or the other (or both) of the following two statements is activated by removing the comment indicator, and the program is recompiled and run, only the **listener** object that was not removed from the list is notified of changes in the values of properties in the **bean**.

```
//    myBean.removePropertyChangeListener(firstListener);
//    myBean.removePropertyChangeListener(secondListener);
```

An **ActionListener** object of the following class is registered to respond to **Action** events on the button labeled *"setTheDate"*. When that button is clicked on the test panel, this object invokes the **setTheDate()** method on the **bean** passing in a new **Date** object as a parameter.

```

class DateActionListener implements ActionListener{
  Beans03 myBean;//save a reference to the bean here

  DateActionListener(Beans03 inBean){//constructor
    myBean = inBean;//save a reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    //Store the current date and time in the bean property
    // named theDate.
    myBean.setTheDate(new Date());
  }//end actionPerformed()
}//end class DateActionListener
```

An **ActionListener** object of the following class is registered to respond to **Action** events on the button labeled *"setTheColor"*. When that button is clicked on the test panel, the **setTheColor()** method of the **bean** is invoked to set the **theColor** property to green.

```
class SetTheColorListener implements ActionListener{
  Beans03 myBean;//save a reference to the bean here

  SetTheColorListener(Beans03 inBean){//constructor
    myBean = inBean;//save a reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.setTheColor(Color.green);
  }//end actionPerformed()
}//end class SetTheColorListener
```

An **ActionListener** object of the following class is registered to respond to **Action** events on the button labeled *"getTheColor"*. When that button is clicked on the test panel, the **getTheColor()** method of the **bean** is invoked which returns the current value of the **theColor** property. The code in the object displays that color on the screen.

```

class GetTheColorListener implements ActionListener{
  Beans03 myBean;//save a reference to the bean here

  GetTheColorListener(Beans03 inBean){//constructor
    myBean = inBean;//save reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    //Display value of the theColor property on the
    // standard output device.
    System.out.println(myBean.getTheColor().toString());
  }//end actionPerformed()
}//end class GetTheColorListener
```

ActionListener objects of the following two classes are registered to respond to **Action** events on the buttons which invoke the **makeRed()** and **makeBlue()** methods of the **bean**. Clicking the corresponding button on the test panel causes one or the other of these objects to invoke the method on the **bean** which in turn causes the **theColor** property of the bean to be set to either red or blue.

```
class RedActionListener implements ActionListener{
  Beans03 myBean;//save a reference to the bean here

  RedActionListener(Beans03 inBean){//constructor
    myBean = inBean;//save the reference to the bean
  }//end constructor

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

class BlueActionListener implements ActionListener{
  Beans03 myBean;//save a reference to the bean here

  BlueActionListener(Beans03 inBean){//constructor
    myBean = inBean;//save the reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.makeBlue();
  }//end actionPerformed()
}//end class BlueActionListener
```

Two **ActionListener** objects of the following class are registered to listen for **propertyChange** events which are multicast by the **bean**. When notified of such changes, code in the **propertyChange()** method of this class extracts and displays information about the identification of the listener object, the source of the event (the **bean**) and the property values. Note that this class implements the **PropertyChangeListener** interface and defines the **propertyChange()** method declared in that interface.

```

class MyPropertyChangeListener
                         implements PropertyChangeListener{
  String theID; //store listener object ID here

  void setTheID(String nameIn){
    //method to save the ID of the object
    theID = nameIn;
  }//end setTheID()

  public void propertyChange(PropertyChangeEvent event){
    //Extract and display information about the event
    System.out.println(theID + " notified of change");
    System.out.println("Property change source: "
                                     + event.getSource());
    System.out.println("Property name: "
                               + event.getPropertyName());
    System.out.println("New property value: "
                                   + event.getNewValue());
    System.out.println("Old property value: "
                                  + event.getOldValue());
    System.out.println();//blank line
  }//end propertyChange()
}//end MyPropertyChangeListener class
```

A consolidated listing of the complete test program is contained in the next section.

Program Listing for the Test Program

This section contains a complete listing of the test program written to test the *bound* property behavior of the **bean** class. The output from running the test is also contained in the listing.

```
/*File Beans03Test.java Copyright 1997, R.G.Baldwin
This program was designed to be compiled and executed
under JDK 1.1.3 or later.

The purpose of this program is to provide the ability to
test the bean class named Beans03.class in a Frame.

A Beans03 object is placed in the frame along with five
buttons.

The visual manifestation of the Bean object is a colored
square.

Two of the buttons exercise the "get" and "set" methods
used to get and set the color value stored in the property
named theColor.

One button exercises the "set" method used to set the date
and time in a write-only property named theDate.

Two of the buttons invoke the makeRed() and makeBlue()
methods of the Bean which modify the value of the property
named theColor.

Two listener objects are instantiated and registered to
be notified by the bean whenever there is a change in the
value of either of the properties.  Actually the objects
are notified whenever a value is assigned to the instance
variables that maintain the property values regardless of
whether or not the new value is different from the old
value.

For those cases where information is returned from the
Bean, it is displayed on the standard output device.

The program was tested using JDK 1.1.3 under Win95.

Clicking the button labeled "Set theColor property"
produced the following output on the screen.  As you can
see, the two different listener objects were notified of
the same change in the property named theColor. In this
```

case, the value of theColor property was changed to green.

```
FirstListener notified of change
Property change source: Beans03[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=0,g=255,b=0]
Old property value: null

SecondListener notified of change
Property change source: Beans03[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=0,g=255,b=0]
Old property value: null
```

Clicking the button labeled "Get theColor property" produced the following output on the screen. Since this action didn't cause the property values to change, the listener objects were not notified of the action.

```
java.awt.Color[r=0,g=255,b=0]
```

Clicking the button labeled "Invoke the makeRed Method" produced the following output on the screen. Again both listener objects were notified of the change in the property named theColor with the new value being red.

```
FirstListener notified of change
Property change source: Beans03[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=255,g=0,b=0]
Old property value: null

SecondListener notified of change
Property change source: Beans03[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=255,g=0,b=0]
Old property value: null
```

Clicking the button labeled "Invoke the makeBlue Method" produced the following output on the screen similar to that produced by invoking the makeRed() method described above.

```
FirstListener notified of change
Property change source: Beans03[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=0,g=0,b=255]
Old property value: null

SecondListener notified of change
Property change source: Beans03[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=0,g=0,b=255]
```

```
Old property value: null


Finally, clicking the button labeled "Set theDate
property" produced the following output on the screen.
In this case, both listener objects were notified and
the information encapsulated in the event object
identified the changed property as the property named
theDate with a new value as shown.

FirstListener notified of change
Property change source: Beans03[canvas0,31,33,50x50]
Property name: theDate
New property value: Sat Oct 18 10:24:49 CDT 1997
Old property value: null

SecondListener notified of change
Property change source: Beans03[canvas0,31,33,50x50]
Property name: theDate
New property value: Sat Oct 18 10:24:49 CDT 1997
Old property value: null

*/

import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.util.*;
//=========================================================//
public class Beans03Test extends Frame{
  public static void main(String[] args){
    new Beans03Test();
  }//end main
  //-------------------------------------------------------//

  public Beans03Test(){//constructor
    setTitle("Copyright 1997, R.G.Baldwin");
    setLayout(new FlowLayout());
    //instantiate a Bean object
    Beans03 myBean = new Beans03();
    add(myBean);//Add it to the Frame

    //Instantiate several test buttons
    Button buttonToSetTheColor =
                       new Button("Set theColor property");
    Button buttonToGetTheColor =
                       new Button("Get theColor property");
    Button buttonToInvokeRedMethod =
                       new Button("Invoke makeRed Method");
    Button buttonToInvokeBlueMethod =
                      new Button("Invoke makeBlue Method");
    Button buttonToSetTheDate =
                        new Button("Set theDate property");

    //Add the test buttons to the frame
    add(buttonToSetTheColor);
    add(buttonToGetTheColor);
    add(buttonToInvokeRedMethod);
    add(buttonToInvokeBlueMethod);
    add(buttonToSetTheDate);

    //Size the frame and make it visible
    setSize(250,350);
    setVisible(true);

    //Register action listener objects for all the test
    // buttons
    buttonToSetTheColor.addActionListener(
                          new SetTheColorListener(myBean));
    buttonToGetTheColor.addActionListener(
                          new GetTheColorListener(myBean));
    buttonToInvokeRedMethod.addActionListener(
                            new RedActionListener(myBean));
    buttonToInvokeBlueMethod.addActionListener(
                           new BlueActionListener(myBean));
    buttonToSetTheDate.addActionListener(
                           new DateActionListener(myBean));

    //Instantiate and register two PropertyChangeListener
    // objects to listen for changes in the bean's
    // properties.
    MyPropertyChangeListener firstListener =
                            new MyPropertyChangeListener();
    //Store an identifying name in the listener object
    firstListener.setTheID("FirstListener");
    myBean.addPropertyChangeListener(firstListener);

    MyPropertyChangeListener secondListener =
                            new MyPropertyChangeListener();
    //Store an identifying name in the listener object
    secondListener.setTheID("SecondListener");
    myBean.addPropertyChangeListener(secondListener);

    //The following statements can be activated to confirm
    // proper operation of the removePropertyChangeListener
    // interface of the bean object.  When one or the other
    // of these statements is activated, and the program is
    // recompiled, only the other listener object is
    // notified of changes in the values of properties
    // in the bean.
//    myBean.removePropertyChangeListener(firstListener);
//    myBean.removePropertyChangeListener(secondListener);

    //terminate when Frame is closed
    this.addWindowListener(new Terminate());
  }//end constructor
}//end class Beans03Test
//=========================================================//
//The following class is used to instantiate objects to
// be registered to listen to one of the buttons on the
// test panel.  When the setTheDate button is pressed, the
// theDate property is set to the current date and time.
```

```
class DateActionListener implements ActionListener{
  Beans03 myBean;//save a reference to the bean here

  DateActionListener(Beans03 inBean){//constructor
    myBean = inBean;//save a reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    //Store the current date and time in the bean property
    // named theDate.
    myBean.setTheDate(new Date());
  }//end actionPerformed()
}//end class DateActionListener
//=========================================================//

//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 setTheColor button is pressed, the theColor
// property is set to green.

// When the getTheColor button is pressed, the current
// color is displayed on the standard output device.

class SetTheColorListener implements ActionListener{
  Beans03 myBean;//save a reference to the bean here

  SetTheColorListener(Beans03 inBean){//constructor
    myBean = inBean;//save a reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.setTheColor(Color.green);
  }//end actionPerformed()
}//end class SetTheColorListener
//---------------------------------------------------------//

class GetTheColorListener implements ActionListener{
  Beans03 myBean;//save a reference to the bean here

  GetTheColorListener(Beans03 inBean){//constructor
    myBean = inBean;//save reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    //Display value of the theColor property on the
    // standard output device.
    System.out.println(myBean.getTheColor().toString());
  }//end actionPerformed()
}//end class GetTheColorListener

//=========================================================//
//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 corresponding 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{
  Beans03 myBean;//save a reference to the bean here

  RedActionListener(Beans03 inBean){//constructor
    myBean = inBean;//save the reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.makeRed();
  }//end actionPerformed()
}//end class RedActionListener
//-------------------------------------------------------//

class BlueActionListener implements ActionListener{
  Beans03 myBean;//save a reference to the bean here

  BlueActionListener(Beans03 inBean){//constructor
    myBean = inBean;//save the reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.makeBlue();
  }//end actionPerformed()
}//end class BlueActionListener
//=======================================================//

//The following class is used to instantiate objects that
// will be bound to the bean in such a way as to be
// notified of changes in the property values in the bean
// object.  When notified of such changes, code in the
// propertyChange() method of this class extracts and
// displays information about the bean and the properties.
class MyPropertyChangeListener
                         implements PropertyChangeListener{
  String theID; //store listener object ID here

  void setTheID(String nameIn){
    //method to save the ID of the object
    theID = nameIn;
  }//end setTheID()

  public void propertyChange(PropertyChangeEvent event){
    //Extract and display information about the event
    System.out.println(theID + " notified of change");
    System.out.println("Property change source: "
                                      + event.getSource());
    System.out.println("Property name: "
                                + event.getPropertyName());
    System.out.println("New property value: "
                                    + event.getNewValue());
    System.out.println("Old property value: "
                                    + event.getOldValue());
    System.out.println();//blank line
  }//end propertyChange()
}//end MyPropertyChangeListener class
//=========================================================//

class Terminate extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    //terminate the program when the window is closed
    System.exit(0);
  }//end windowClosing
}//end class Terminate
//=========================================================//
```

Using the PropertyChangeSupport Class

Now that you know how to roll your own beans with *bound* properties, I am going to let you in on a secret that can reduce your programming effort a little.

Java provides the **java.beans.PropertyChangeSupport** class that can be used to handle the maintenance of the registration list as well as the task of firing events to the registered listener objects on that list. A description of the constructor and methods of the class is shown below.

```
Constructor
public PropertyChangeSupport(Object sourceBean)

Methods
public synchronized void addPropertyChangeListener(
                        PropertyChangeListener listener)
Adds a PropertyChangeListener to the listener list.

Parameters:
  listener - The PropertyChangeListener to be added

public synchronized void removePropertyChangeListener(
                       PropertyChangeListener listener)
Removes a PropertyChangeListener from the listener list.

Parameters:
  listener - The PropertyChangeListener to be removed

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

Reports a bound property update to any registered
listeners. No event is fired if old and new are
equal and non-null.

Parameters:
  propertyName - The name of the property that was
                 changed.
  oldValue - The old value of the property.
  newValue - The new value of the property.
```

As you can see, an object of this class can be used to maintain the registration list and to fire the events. This class can be either extended or inherited. In the case of the sample program in this section, it is not possible to extend the support class because the **bean** class already extends the **Canvas** class. Therefore, in this sample program, the support class was instantiated into a separate object that is used to handle the list-maintenance and event-firing tasks.

The requirement to instantiate the support class into a separate object resulted in a small amount of extra programming effort. In particular, it was necessary to define *add* and *remove* method shells for the **bean** class and then pass the **listener** object parameters received by those method shells to the *add* and *remove* methods in the support object. If the **bean** class extended the support class, the *add* and *remove* methods of the support class would be inherited into the **bean** class.

Complete listings of the revised **bean** class and the corresponding test program follow. Important information is included in the comments in these two programs.

```
/*File Beans05.java Copyright 1997, R.G.Baldwin
This program was designed to be compiled and executed
under JDK 1.1.3 or later.

This bean class is designed to replicate the functionality
of the bean class named Beans04 by making use of the
support class named java.beans.PropertyChangeSupport to
reduce the level of programming effort required.

This support class provides  methods for maintaining
the list of registered PropertyChange listeners and for
firing events to all of the listener objects on that list,
thus eliminating the need to code those capabilities by
hand as was done with Beans04.

The support class can either be extended or instantiated.
In this case, because this bean class was already
extending the Canvas class and multiple inheritance is
not allowed, the support class was instantiated.  This
made it necessary to do a little extra work in providing
bean interface methods to add and remove listener objects
and then to call the corresponding methods in the support
class object passing the listener references as parameters.
//=========================================================//
*/

import java.awt.event.*;
import java.awt.*;
import java.io.Serializable;
import java.util.*;
import java.beans.*;
```

```
//=====================================================//
//All beans should implement the Serializable interface
public class Beans05 extends Canvas
                                   implements Serializable{

  //The following instance variables are used to store
  // property values.
  protected Color myColor;
  protected Date myDate;

  //The following reference variable is used to access
  // the list maintenance and event firing capabilities
  // of the PropertyChangeSupport class, an object of
  // which is instantiated in the constructor.
  PropertyChangeSupport supportObj;
  //-----------------------------------------------------//

  public Beans05(){//constructor
    //This bean is a visible square that is initialized to
    // yellow and can then be changed to green, red, and
    // blue by invoking methods of the class.

    myColor = Color.yellow;
    setBackground(myColor);

    //Instantiate an object of the support class to handle
    // list maintenance and event firing tasks.  The
    // constructor requires this object as the source of
    // the events.
    supportObj = new PropertyChangeSupport(this);

  }//end constructor
  //-----------------------------------------------------//

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

  //The following "set" and "get" methods in conjunction
  // with the instance variable named myColor constitute a
  // property named theColor.
  public synchronized void setTheColor(Color inColor){
    Color oldColor = myColor;
    myColor = inColor;
    this.setBackground(myColor);
    //notify property listeners of property change
    if(!myColor.equals(oldColor))
      notifyPropertyChange("theColor");
  }//end setTheColor()

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

  //The following "set" method in conjunction with the
  // instance variable named myDate constitute a write-only
  // property named theDate.
  public synchronized void setTheDate(Date dateIn){
    Date oldDate = myDate;
    myDate = dateIn;
    //notify property listeners of property change
    if(!myDate.equals(oldDate))
      notifyPropertyChange("theDate");
  }//end setTheDate()
  //----------------------------------------------------//

  //The following two methods are exposed to the builder
  // tool as accessible methods.
  public synchronized void makeBlue(){
    Color oldColor = myColor;
    myColor = Color.blue;
    this.setBackground(myColor);
    //notify property listeners of property change
    if(!myColor.equals(oldColor))
      notifyPropertyChange("theColor");
  }//end makeBlue()

  public synchronized void makeRed(){
    Color oldColor = myColor;
    myColor = Color.red;
    this.setBackground(myColor);
    //notify property listeners of property change
    if(!myColor.equals(oldColor))
      notifyPropertyChange("theColor");
  }//end makeRed()
  //----------------------------------------------------//

  //The following two methods are used to maintain a list
  // of listener objects who request to be notified of
  // changes to the properties or who request to be removed
  // from the list.  Note that these two methods do
  // nothing more than to accept a reference to the
  // object requesting registration and pass that reference
  // to the list maintenance facility provided by an object
  // of the PropertyChangeSupport class.

  //Add a property change listener object to the list.
  public synchronized void addPropertyChangeListener(
                        PropertyChangeListener listener){
    supportObj.addPropertyChangeListener(listener);
  }//end addPropertyChangeListener

//------------------------------------------------------//

  //Remove a property change listener from the list.
  public synchronized void removePropertyChangeListener(
                        PropertyChangeListener listener){
    supportObj.removePropertyChangeListener(listener);
  }//end removePropertyChangeListener()
  //-----------------------------------------------------//

  //The following method is used to notify listener
  // objects of changes in the properties.  The incoming
  // parameter is the name of the property that has
  // changed.  Note that this method makes use of the
  // event-firing capability of an object of the
  // PropertyChangeSupport class.
  protected void notifyPropertyChange(
                                 String changedProperty){
    if(changedProperty.compareTo("theColor") == 0)
      //Change was in theColor property
      supportObj.firePropertyChange(
                           changedProperty,null,myColor);
    else//Change was in the theDate property
      supportObj.firePropertyChange(
                            changedProperty,null,myDate);
  }//end notifyPropertyChange()

}//end class Beans05.java
//=========================================================//
```

A listing of the test program follows.

```
/*File Beans05Test.java Copyright 1997, R.G.Baldwin
This program was designed to be compiled and executed
under JDK 1.1.3 or later.

The purpose of this program is to provide the ability to
test the bean class named Beans05.class in a Frame.

See the comments in the file named Beans05.java to
understand how this test program and the bean differ from
the test program named Beans04Test and its corresponding
bean.

Briefly, this pair of programs replicates the Beans04
pair in functionality.  However, Beans05 uses a support
class named java.beans.PropertyChangedSupport to reduce
the programming effort required for the bean.

See the program named Beans04Test for an operational
description of this program along with sample output
produced by the program.
*/

import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.util.*;
//=========================================================//
public class Beans05Test extends Frame{
  public static void main(String[] args){
    new Beans05Test();
  }//end main
  //-----------------------------------------------------//

  public Beans05Test(){//constructor
    setTitle("Copyright 1997, R.G.Baldwin");
    setLayout(new FlowLayout());
    //instantiate a Bean object
    Beans05 myBean = new Beans05();
    add(myBean);//Add it to the Frame

    //Instantiate several test buttons
    Button buttonToSetTheColor =
                       new Button("Set theColor property");
    Button buttonToGetTheColor =
                       new Button("Get theColor property");
    Button buttonToInvokeRedMethod =
                       new Button("Invoke makeRed Method");
    Button buttonToInvokeBlueMethod =
                      new Button("Invoke makeBlue Method");
    Button buttonToSetTheDate =
                        new Button("Set theDate property");

    //Add the test buttons to the frame
    add(buttonToSetTheColor);
    add(buttonToGetTheColor);
    add(buttonToInvokeRedMethod);
    add(buttonToInvokeBlueMethod);
    add(buttonToSetTheDate);

    //Size the frame and make it visible
    setSize(250,350);
    setVisible(true);

    //Register action listener objects for all the test
    // buttons
    buttonToSetTheColor.addActionListener(
                          new SetTheColorListener(myBean));
    buttonToGetTheColor.addActionListener(
                          new GetTheColorListener(myBean));
    buttonToInvokeRedMethod.addActionListener(
                            new RedActionListener(myBean));
    buttonToInvokeBlueMethod.addActionListener(
                           new BlueActionListener(myBean));
    buttonToSetTheDate.addActionListener(
                           new DateActionListener(myBean));

    //Instantiate and register two PropertyChangeListener
    // objects to listen for changes in the bean's
    // properties.
    MyPropertyChangeListener firstListener =
                            new MyPropertyChangeListener();
    //Store an identifying name in the listener object
    firstListener.setTheID("FirstListener");
    myBean.addPropertyChangeListener(firstListener);
    MyPropertyChangeListener secondListener =
                            new MyPropertyChangeListener();
    //Store an identifying name in the listener object
    secondListener.setTheID("SecondListener");
    myBean.addPropertyChangeListener(secondListener);

    //The following statements can be activated to confirm
    // proper operation of the removePropertyChangeListener
    // interface of the bean object.  When one or the other
    // of these statements is activated, and the program is
    // recompiled, only the other listener object is
    // notified of changes in the values of properties
    // in the bean.
//    myBean.removePropertyChangeListener(firstListener);
//    myBean.removePropertyChangeListener(secondListener);

    //terminate when Frame is closed
    this.addWindowListener(new Terminate());
  }//end constructor
}//end class Beans05Test
//=========================================================//
//The following class is used to instantiate objects to
// be registered to listen to one of the buttons on the
// test panel.  When the setTheDate button is pressed, the
// theDate property is set to the current date and time.
class DateActionListener implements ActionListener{
  Beans05 myBean;//save a reference to the bean here

  DateActionListener(Beans05 inBean){//constructor
    myBean = inBean;//save a reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    //Store the current date and time in the bean property
    // named theDate.
    myBean.setTheDate(new Date());
  }//end actionPerformed()
}//end class DateActionListener
//=========================================================//

//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 setTheColor button is pressed, the theColor
// property is set to green.

// When the getTheColor button is pressed, the current
// color is displayed on the standard output device.

class SetTheColorListener implements ActionListener{
  Beans05 myBean;//save a reference to the bean here

  SetTheColorListener(Beans05 inBean){//constructor
    myBean = inBean;//save a reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.setTheColor(Color.green);
  }//end actionPerformed()
}//end class SetTheColorListener
//----------------------------------------------------------//

class GetTheColorListener implements ActionListener{
  Beans05 myBean;//save a reference to the bean here

  GetTheColorListener(Beans05 inBean){//constructor
    myBean = inBean;//save reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    //Display value of the theColor property on the
    // standard output device.
    System.out.println(myBean.getTheColor().toString());
  }//end actionPerformed()
}//end class GetTheColorListener

//==========================================================//
//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 corresponding 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{
  Beans05 myBean;//save a reference to the bean here

  RedActionListener(Beans05 inBean){//constructor
    myBean = inBean;//save the reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.makeRed();
  }//end actionPerformed()
}//end class RedActionListener
//----------------------------------------------------------//

class BlueActionListener implements ActionListener{
  Beans05 myBean;//save a reference to the bean here

  BlueActionListener(Beans05 inBean){//constructor
    myBean = inBean;//save the reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.makeBlue();
  }//end actionPerformed()
}//end class BlueActionListener
//==========================================================//
```

```
//The following class is used to instantiate objects that
// will be bound to the bean in such a way as to be
// notified of changes in the property values in the bean
// object.  When notified of such changes, code in the
// propertyChange() method of this class extracts and
// displays information about the bean and the properties.
class MyPropertyChangeListener
                         implements PropertyChangeListener{
  String theID; //store listener object ID here

  void setTheID(String nameIn){
    //method to save the ID of the object
    theID = nameIn;
  }//end setTheID()

  public void propertyChange(PropertyChangeEvent event){
    //Extract and display information about the event
    System.out.println(theID + " notified of change");
    System.out.println("Property change source: "
                                      + event.getSource());
    System.out.println("Property name: "
                                + event.getPropertyName());
    System.out.println("New property value: "
                                    + event.getNewValue());
    System.out.println("Old property value: "
                                    + event.getOldValue());
    System.out.println();//blank line
  }//end propertyChange()
}//end MyPropertyChangeListener class
//=========================================================//

class Terminate extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    //terminate the program when the window is closed
    System.exit(0);
  }//end windowClosing
}//end class Terminate
//=========================================================//
```

.

Review

Q - The bean class named Beans03 broadcasts a **PropertyChangeEvent** whenever a new value is assigned to one of the instance variables used to maintain the *bound* property values regardless of whether the new value is different from, or the same as the old value.

Without viewing the following solution, upgrade that bean class to produce a new bean class that will broadcast a **PropertyChangeEvent** only if the new value is different from the old value.

Provide a bean test program to demonstrate proper operation of the new bean class.

A - See the following bean program and test program.

Bean program follows:

```
/*File Beans04.java Copyright 1997, R.G.Baldwin
This program was designed to be compiled and executed
under JDK 1.1.3 or later.

The bean class named Beans03 broadcasts a PropertyChange
Event whenever a new value is assigned to one of the
instance variables used to maintain the property values
regardless of whether the new value is different from, or
the same as the old value.

Upgrade that bean class to produce a new bean class that
will refrain from broadcasting a PropertyChangeEvent if
the new value is the same as the old value.

Provide a bean test program to demonstrate proper operation
of the new bean class.

The main purpose of this program is to illustrate the use
of "bound" properties in Beans.

This is a "bean" class that satisfies the interface
requirements for beans using design patterns.

This bean class has two properties named theColor and
theDate.

The current value of the property named theColor is stored
in the instance variable named myColor.

The current value of the property named theDate is stored
in the instance variable named myDate.  theDate is a
write-only property because it has a "set" method but does
not have a "get" method.

The program maintains a list of objects that request to be

notified whenever there is a change in the value of either
of the properties.  Whenever the value of either property
changes, all of the objects on the list are notified of the
change by invoking their propertyChange() method and
passing an object of type PropertyChangeEvent as a
parameter.

Objects that request to be added to the list must be of a
class that implements the PropertyChangeListener interface
and defines the propertyChange() method that is declared
in that interface.

The PropertyChangeEvent object passed as a parameter to the
propertyChange() method in the listener objects contains
```

```
the following information:

  Object source, //the bean object
  String propertyName, //the name of the changed property
  Object oldValue, //the old value of the changed property
  Object newValue  //the new value of the changed property

This program doesn't save the old value and therefore
passes null as the old value of the changed property
because the old value is not available when the listener
objects are notified.

The following methods are available to extract information
from the object passed as a parameter.  These methods are
defined by the PropertyChangeEvent class or its superclass
named EventObject:

  public Object getSource();
  public Object getNewValue();
  public Object getOldValue();
  public String getPropertyName;
  public void setPropagationId();
  public Object getPropagationId;

Apparently the PropagationID is reserved for future use.

The program was compiled and tested under JDK 1.1.3
and Win95.  Another program named Beans03Test.java was
used to test the bean.  It was not tested in the BeanBox.
//=========================================================//
*/

import java.awt.event.*;
import java.awt.*;
import java.io.Serializable;
import java.util.*;
import java.beans.*;
//=========================================================//
//All beans should implement the Serializable interface
public class Beans04 extends Canvas
                                   implements Serializable{

  //The following instance variables are used to store
  // property values.
  protected Color myColor;
  protected Date myDate;

  //The following vector is used to maintain a list of
  // listeners who request to be notified of changes in the
  // property values.
  protected Vector propChangeListeners = new Vector();
  //-----------------------------------------------------//

  public Beans04(){//constructor
    //This bean is a visible square that is initialized to
    // yellow and can then be changed to green, red, and
    // blue by invoking methods of the class.
    myColor = Color.yellow;
    setBackground(myColor);
  }//end constructor
  //-----------------------------------------------------//

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

  //The following "set" and "get" methods in conjunction
  // with the instance variable named myColor constitute a
  // property named theColor.
  public synchronized void setTheColor(Color inColor){
    Color oldColor = myColor;
    myColor = inColor;
    this.setBackground(myColor);
    //notify property listeners of property change
    if(!myColor.equals(oldColor))
      notifyPropertyChange("theColor");
  }//end setTheColor()

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

  //The following "set" method in conjunction with the
  // instance variable named myDate constitute a write-only
  // property named theDate.
  public synchronized void setTheDate(Date dateIn){
    Date oldDate = myDate;
    myDate = dateIn;
    //notify property listeners of property change
    if(!myDate.equals(oldDate))
      notifyPropertyChange("theDate");
  }//end setTheDate()
  //-----------------------------------------------------//

  //The following two methods are exposed to the builder
  // tool as accessible methods.
  public synchronized void makeBlue(){
    Color oldColor = myColor;
    myColor = Color.blue;
    this.setBackground(myColor);
    //notify property listeners of property change
    if(!myColor.equals(oldColor))
      notifyPropertyChange("theColor");
  }//end makeBlue()

  public synchronized void makeRed(){
    Color oldColor = myColor;
    myColor = Color.red;
    this.setBackground(myColor);
    //notify property listeners of property change
    if(!myColor.equals(oldColor))
      notifyPropertyChange("theColor");
  }//end makeRed()
  //-------------------------------------------------------//

  //The following two methods are used to maintain a list
  // of listener objects who request to be notified of
  // changes to the properties or who request to be removed
  // from the list.

  //Add a property change listener object to the list.
  public synchronized void addPropertyChangeListener(
                         PropertyChangeListener listener){
    //If the listener is not already registered, add it
    // to the list.
    if(!propChangeListeners.contains(listener)){
      propChangeListeners.addElement(listener);
    }//end if
  }//end addPropertyChangeListener

  //Remove a property change listener from the list.
  public synchronized void removePropertyChangeListener(
                         PropertyChangeListener listener){
    //If the listener is on the list, remove it
    if(propChangeListeners.contains(listener)){
      propChangeListeners.removeElement(listener);
    }//end if
  }//end removePropertyChangeListener
  //-------------------------------------------------------//

  //The following method is used to notify listener
  // objects of changes in the properties.  The incoming
  // parameter is the name of the property that has
  // changed.
  protected void notifyPropertyChange(
                                  String changedProperty){
    //Instantiate the event object containing information
    // about the property that has changed.
    PropertyChangeEvent event;
    if(changedProperty.compareTo("theColor") == 0)
      //Change was in theColor property
      event = new PropertyChangeEvent(
                       this,changedProperty,null,myColor);
    else//Change was in the theDate property
      event = new PropertyChangeEvent(
                        this,changedProperty,null,myDate);

    //Make a working copy of the list that cannot be
    // modified while objects on the list are being
    // notified of the change.
    Vector tempList;
    synchronized(this){
      tempList = (Vector)propChangeListeners.clone();
    }//end synchronized block
    //Notify all listener objects on the list.  Note the
    // requirement to cast the objects in the list from
    // Object to PropertyChangeListener.
    for(int cnt = 0; cnt < tempList.size();cnt++){
      PropertyChangeListener theListener =
           (PropertyChangeListener)tempList.elementAt(cnt);
      //Invoke the propertyChange() method on theListener
      theListener.propertyChange(event);
    }//end for loop
  }//end notifyPropertyChange
}//end class Beans04.java
//=========================================================//
```

Test program follows:

```
/*File Beans04Test.java Copyright 1997, R.G.Baldwin
This program was designed to be compiled and executed
under JDK 1.1.3 or later.

The purpose of this program is to provide the ability to
test the bean class named Beans04.class in a Frame.

A Beans04 object is placed in the frame along with five
buttons.

The visual manifestation of the Bean object is a colored
square.

Two of the buttons exercise the "get" and "set" methods
used to get and set the color value stored in the property
named theColor.

One button exercises the "set" method used to set the date
and time in a write-only property named theDate.

Two of the buttons invoke the makeRed() and makeBlue()
methods of the Bean which modify the value of the property
named theColor.

Two listener objects are instantiated and registered to
be notified by the bean whenever there is a change in the
value of either of the properties.

For those cases where information is returned from the
Bean, it is displayed on the standard output device.

The program was tested using JDK 1.1.3 under Win95.

Clicking the button labeled "Set theColor property"
produced the following output on the screen.  As you can
see, the two different listener objects were notified of
the same change in the property named theColor. In this
case, the value of theColor property was changed to green.
```

```
FirstListener notified of change
Property change source: Beans04[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=0,g=255,b=0]
Old property value: null

SecondListener notified of change
Property change source: Beans04[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=0,g=255,b=0]
Old property value: null

As discussed below regarding the makeRed() method,
clicking the button labeled "Set theColor property"
several times in succession produces only the single
output shown above.


Clicking the button labeled "Get theColor property"
produced the following output on the screen.  Since
this action didn't cause the property values to change,
the listener objects were not notified of the action.

java.awt.Color[r=0,g=255,b=0]


Clicking the button labeled "Invoke the makeRed Method"
produced the following output on the screen. Again both
listener objects were notified of the change in the
property named theColor with the new value being red.

FirstListener notified of change
Property change source: Beans04[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=255,g=0,b=0]
Old property value: null

SecondListener notified of change
Property change source: Beans04[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=255,g=0,b=0]
Old property value: null


Clicking the button labeled "Invoke the makeRed Method"
several times in succession produces only the single
output shown above.  The Beans04 class only broadcasts an
event when the value of the property changes to a new
value, and "changing" the property to the same value
as before does not cause it to broadcast an event.


Clicking the button labeled "Invoke the makeBlue Method"
produced the following output on the screen similar to
that produced by invoking the makeRed() method described
```

```
above.

FirstListener notified of change
Property change source: Beans04[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=0,g=0,b=255]
Old property value: null

SecondListener notified of change
Property change source: Beans04[canvas0,31,33,50x50]
Property name: theColor
New property value: java.awt.Color[r=0,g=0,b=255]
Old property value: null


As discussed above with regard to the makeRed method,
clicking the "Invoke the makeBlue Method" several times
in succession produces only the single output shown
above.


Finally, clicking the button labeled "Set theDate
property" produced the following output on the screen.
In this case, both listener objects were notified and
the information encapsulated in the event object
identified the changed property as the property named
theDate with a new value as shown.

FirstListener notified of change
Property change source: Beans04[canvas0,31,33,50x50]
Property name: theDate
New property value: Sat Oct 18 10:24:49 CDT 1997
Old property value: null

SecondListener notified of change
Property change source: Beans04[canvas0,31,33,50x50]
Property name: theDate
New property value: Sat Oct 18 10:24:49 CDT 1997
Old property value: null

Unlike the previous discussions involving the theColor
property, each time you click the "Set theDate property"
button, a new output will be produced.  This is because
the date information being stored in the theDate property
also includes the time and the time always changes
between successive clicks on the button.
*/

import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.util.*;
//=========================================================//
public class Beans04Test extends Frame{
  public static void main(String[] args){
    new Beans04Test();
  }//end main
  //----------------------------------------------------//

  public Beans04Test(){//constructor
    setTitle("Copyright 1997, R.G.Baldwin");
    setLayout(new FlowLayout());
    //instantiate a Bean object
    Beans04 myBean = new Beans04();
    add(myBean);//Add it to the Frame

    //Instantiate several test buttons
    Button buttonToSetTheColor =
                       new Button("Set theColor property");
    Button buttonToGetTheColor =
                       new Button("Get theColor property");
    Button buttonToInvokeRedMethod =
                       new Button("Invoke makeRed Method");
    Button buttonToInvokeBlueMethod =
                      new Button("Invoke makeBlue Method");
    Button buttonToSetTheDate =
                        new Button("Set theDate property");

    //Add the test buttons to the frame
    add(buttonToSetTheColor);
    add(buttonToGetTheColor);
    add(buttonToInvokeRedMethod);
    add(buttonToInvokeBlueMethod);
    add(buttonToSetTheDate);

    //Size the frame and make it visible
    setSize(250,350);
    setVisible(true);

    //Register action listener objects for all the test
    // buttons
    buttonToSetTheColor.addActionListener(
                          new SetTheColorListener(myBean));
    buttonToGetTheColor.addActionListener(
                          new GetTheColorListener(myBean));
    buttonToInvokeRedMethod.addActionListener(
                            new RedActionListener(myBean));
    buttonToInvokeBlueMethod.addActionListener(
                           new BlueActionListener(myBean));
    buttonToSetTheDate.addActionListener(
                           new DateActionListener(myBean));

    //Instantiate and register two PropertyChangeListener
    // objects to listen for changes in the bean's
    // properties.
    MyPropertyChangeListener firstListener =
                            new MyPropertyChangeListener();
    //Store an identifying name in the listener object
    firstListener.setTheID("FirstListener");
    myBean.addPropertyChangeListener(firstListener);

    MyPropertyChangeListener secondListener =
                            new MyPropertyChangeListener();
    //Store an identifying name in the listener object
    secondListener.setTheID("SecondListener");
    myBean.addPropertyChangeListener(secondListener);

    //The following statements can be activated to confirm
    // proper operation of the removePropertyChangeListener
    // interface of the bean object.  When one or the other
    // of these statements is activated, and the program is
    // recompiled, only the other listener object is
    // notified of changes in the values of properties
    // in the bean.
//    myBean.removePropertyChangeListener(firstListener);
//    myBean.removePropertyChangeListener(secondListener);

    //terminate when Frame is closed
    this.addWindowListener(new Terminate());
  }//end constructor
}//end class Beans04Test

//=========================================================//
//The following class is used to instantiate objects to
// be registered to listen to one of the buttons on the
// test panel.  When the setTheDate button is pressed, the
// theDate property is set to the current date and time.
class DateActionListener implements ActionListener{
  Beans04 myBean;//save a reference to the bean here

  DateActionListener(Beans04 inBean){//constructor
    myBean = inBean;//save a reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    //Store the current date and time in the bean property
    // named theDate.
    myBean.setTheDate(new Date());
  }//end actionPerformed()
}//end class DateActionListener
//=========================================================//

//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 setTheColor button is pressed, the theColor
// property is set to green.

// When the getTheColor button is pressed, the current
// color is displayed on the standard output device.

class SetTheColorListener implements ActionListener{
  Beans04 myBean;//save a reference to the bean here

  SetTheColorListener(Beans04 inBean){//constructor
    myBean = inBean;//save a reference to the bean
  }//end constructor
  public void actionPerformed(ActionEvent e){
    myBean.setTheColor(Color.green);
  }//end actionPerformed()
}//end class SetTheColorListener
//-------------------------------------------------------//

class GetTheColorListener implements ActionListener{
  Beans04 myBean;//save a reference to the bean here

  GetTheColorListener(Beans04 inBean){//constructor
    myBean = inBean;//save reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    //Display value of the theColor property on the
    // standard output device.
    System.out.println(myBean.getTheColor().toString());
  }//end actionPerformed()
}//end class GetTheColorListener

//=======================================================//
//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 corresponding 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{
  Beans04 myBean;//save a reference to the bean here

  RedActionListener(Beans04 inBean){//constructor
    myBean = inBean;//save the reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.makeRed();
  }//end actionPerformed()
}//end class RedActionListener
//-------------------------------------------------------//

class BlueActionListener implements ActionListener{
  Beans04 myBean;//save a reference to the bean here

  BlueActionListener(Beans04 inBean){//constructor
    myBean = inBean;//save the reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.makeBlue();
  }//end actionPerformed()
}//end class BlueActionListener
//=======================================================//

//The following class is used to instantiate objects that
```

```
// will be bound to the bean in such a way as to be
// notified of changes in the property values in the bean
// object.  When notified of such changes, code in the
// propertyChange() method of this class extracts and
// displays information about the bean and the properties.
class MyPropertyChangeListener
                         implements PropertyChangeListener{
  String theID; //store listener object ID here

  void setTheID(String nameIn){
    //method to save the ID of the object
    theID = nameIn;
  }//end setTheID()

  public void propertyChange(PropertyChangeEvent event){
    //Extract and display information about the event
    System.out.println(theID + " notified of change");
    System.out.println("Property change source: "
                                      + event.getSource());
    System.out.println("Property name: "
                                + event.getPropertyName());
    System.out.println("New property value: "
                                    + event.getNewValue());
    System.out.println("Old property value: "
                                    + event.getOldValue());
    System.out.println();//blank line
  }//end propertyChange()
}//end MyPropertyChangeListener class
//=========================================================//

class Terminate extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    //terminate the program when the window is closed
    System.exit(0);
  }//end windowClosing
}//end class Terminate
//=========================================================//
```

.

-end-