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

JavaBeans, Properties of Beans, Constrained Properties

Java Programming, Lecture Notes # 512, Revised 02/19/98.

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.

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 19, 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:

> An object with constrained properties allows other objects to veto a constrained property value change. Constrained property listeners can veto a change by throwing a PropertyVetoException.

In this lesson, we will develop and test a **bean** class, named **Beans06**, that illustrates the use of *constrained* 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 **VetoableChangeListener** objects whenever the value of one of the properties becomes subject to change.

The **bean** will also be *bound*. As such, the test program will also confirm that the bean properly broadcasts an event to a list of registered **PropertyChangeListener** objects whenever the value of one of the properties actually 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:  Beans06
Class of bean: class Beans06

==== Properties: ====
Name: preferredSize
 Type:       class java.awt.Dimension
 Get method: public synchronized java.awt.Dimension
                                  Beans06.getPreferredSize()
 Set method: null
Name: theColor
 Type:       class java.awt.Color
 Get method: public synchronized java.awt.Color
                                       Beans06.getTheColor()
 Set method: public synchronized void
                         Beans06.setTheColor(java.awt.Color)

==== Events: ====
Event Name: vetoableChange
 Add Method:    public synchronized void
                       Beans06.addVetoableChangeListener(
                          java.beans.VetoableChangeListener)
 Remove Method: public synchronized void
                       Beans06.removeVetoableChangeListener(
                          java.beans.VetoableChangeListener)
 Event Type: vetoableChange

Event Name: propertyChange
 Add Method:    public synchronized void
                       Beans06.addPropertyChangeListener(
                          java.beans.PropertyChangeListener)
 Remove Method: public synchronized void
                       Beans06.removePropertyChangeListener(
                          java.beans.PropertyChangeListener)
 Event Type: propertyChange
```

```
==== Methods: ====
makeRed
removePropertyChangeListener
getTheColor
setTheColor
removeVetoableChangeListener
getPreferredSize
addPropertyChangeListener
makeBlue
addVetoableChangeListener
```

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 **VetoableChangeListener** objects whenever one of the properties is subject to change. (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 value of the other **theColor**.)

Properties

As discussed in earlier lessons, Java **beans** supports four kinds of properties:

- Simple
- Indexed
- Bound
- Constrained

Previous lessons have discussed *Simple*, *Indexed*, and *Bound* properties. This lesson will concentrate on *Constrained* properties.

The **bean** class developed for this lesson has the following property which is both *Bound* and *Constrained* properties:

```
Name: theColor
 Type:       class java.awt.Color
 Get method: public synchronized java.awt.Color
                                        Beans06.getTheColor()
 Set method: public synchronized void
                          Beans06.setTheColor(java.awt.Color)
```

A single property was included in this **bean** for simplicity. A subsequent sample program will illustrate the use of a **bean** with multiple properties which are both *Bound* and *Constrained*.

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

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

Objects that request to be added to the second list must be of a class tha timplements the **VetoableChangeListener** interface and defines the **vetoableChange()** method that is declared in that interface.

Whenever the value of the property becomes subject to change, a **vetoableChange()** event is broadcast to all of the registered **VetoableChangeListener** objects. Those objects can either accept the change or veto it. An object that vetos the change does so by raising an exception.

The event is broadcast by invoking the **vetoableChange()** 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.

Note that the **PropertyChangeEvent** object passed as a parameter to the **vetoableChange()** method in the listener objects is the same type of object passed to the **propertyChange()** method for *Bound* properties. The object 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 **VetoableChangeListener** objects takes place before the change in the property value 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;

Since the property in this sample program is both *Constrained* and *bound*, if the change is not vetoed (and if it really does represent a change in the value), a **propertyChange()** event is broadcast to all of the **PropertyChangeListener** objects registered on that list.

Sample Bean Program

This program was designed to be compiled and executed under JDK 1.1.3 or later.

This program illustrates the use of beans with a property which is both *bound* and *constrained*.

The bean has one property named **theColor**. Two separate instance variables named **oldColor** and **newColor** are used to maintain the value of the property.

Applying introspection to the bean yields the results shown earlier in this lesson.

The most significant new addition to this bean class is the ability for a listener to **veto** a proposed change in the value of a property. When that happens, the proposed new property value is discarded and the actual property value is not changed.

The following methods:

- setTheColor()
- makeRed()
- makeBlue()

all provide the opportunity for a **VetoableChangeListener** object to **veto** a proposed new color value for the property named **theColor**.

Each of these methods receives a proposed new color value as a parameter when it is invoked. The method saves the current value of the property in the instance variable named **oldColor**. Then it makes a call to the method named **notifyVetoableChange()** inside a **try** block.

The **notifyVetoableChange()** method broadcasts a **vetoableChange()** event to all of the **VetoableChangeListener** objects that are registered to receive such an event. The broadcast is accomplished making use of the **VetoableChangeSupport** class which not only handles the firing of the event, but does some additional necessary processing as well.

Any listener object that wants to veto the change throws a **PropertyVetoException** which finds its way back to method listed above that invoked **notifyVetoableChange()** in the first place.

When the exception is thrown, it is caught in a **catch** block. The code in the catch block restores the property value to its original value and displays the exception. In other words, the proposed new value is replaced by the value of the property that existed before the proposed new value was received.

Whether it is vetoed or not, the proposed new value (or the replacement for the proposed new value) becomes the current value and is used to set the background color of the bean.

The proposed new value is also compared with the value of the property that existed before the proposed new value was received. If they are different (meaning that an actual property change

has occurred) the **notifyPropertyChange()** method is invoked to broadcast a **propertyChange()** event to all **PropertyChangeListener** objects registered to receive such an event. If the property value didn't actually change, the **propertyChange()** event is not broadcast.

An important aspect of the behavior of this bean is based on the use of the **fireVetoableChange()** method of the **VetoableChangeSupport** class to actually broadcast the event. A description of this method follows. Pay particular attention to the behavior of the method in the circumstance where someone wants to veto the change.

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


  Report a vetoable property update to any registered
  listeners. If anyone vetos the change, then fire a new
  event reverting everyone to the old value and then
  rethrow the PropertyVetoException.

  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.
  Throws: PropertyVetoException
      if the recipient wishes the property change to be
      rolled back.
```

.

Interesting Code Fragments from the Bean Program

This **bean** class contains a number of interesting code fragments. The following statements are used to store the current property value and a proposed new property value. These are straightforward and the only thing that makes them interesting is the way that they are used later in dealing with the possibility of a veto.

```
protected Color oldColor;
protected Color newColor;
```

The following reference variables are used to access the list-maintenance and event-firing capabilities of the **PropertyChangeSupport** and **VetoableChangeSupport** classes. An object of each of these classes is instantiated in the constructor.

```

PropertyChangeSupport changeSupportObj;
VetoableChangeSupport vetoSupportObj;
```

This bean is a visible square that is initialized to **yellow** and can then be changed to **red** or **blue** by invoking methods of the class named **makeRed()** and **makeBlue()**. The color can be set to any color by invoking the **setTheColor()** method and passing a color in as a parameter. The following code fragment from the constructor is used to perform the initialization.

```
    newColor = Color.yellow;
    setBackground(newColor);
```

As mentioned earlier, objects of the classes **PropertyChangeSupport** and **VetoableChangeSupport** are instantiated in the constructor. These classes can either be extended or instantiated in order to take advantage of the capabilities that they offer. In this case, since this **bean** class already extends another class, it is necessary to instantiate the support classes as separate objects.

The following code fragment in the constructor performs the required instantiation. The constructor for these support classes requires an object reference as a parameter. That object reference is later used to identify the source of events fired by the support objects. In this case, we pass the **this** reference in as a parameter to specify the **bean** as the source of the events.

```
    changeSupportObj = new PropertyChangeSupport(this);
    vetoSupportObj = new VetoableChangeSupport(this);
```

In this bean, there are three different methods that can modify the values of the **Color** property:

- makeRed()
- makeBlue()
- setTheColor()

Each of these methods must deal with the possibility that a **VetoableChangeListener** object will veto the proposed change that results from invoking the method. A veto means that the proposed change must not be implemented.

When a listener object vetos a change, the event-firing mechanism in the support class automatically fires a second event specifying the old value as the new value. This has the effect of notifying all of the listener objects that the property value has been rolled back to its previous value. However, you must provide the code in the **bean** to actually implement the rollback.

The following method is invoked by all three of the methods listed above to implement the rollback. A veto occurs when one of the listener objects raises a **PropertyVetoException**. At the point where the following method is invoked, the current value of the property has been stored in the instance variable named **oldColor** and the proposed new value has been stored in the instance variable named **newColor**.

The code in the following method monitors for a veto by enclosing the call to the **notifyVetoableChange()** method inside a **try** block. If a listener object vetos the change, a

PropertyVeotException will be raised and the code in the **catch** block will be executed. Otherwise, the code in the **catch** block will be skipped.

In the case of a veto, the current value is recovered from **oldColor** and stored in **newColor**, thereby replacing the proposed new value with the unchanged current value. From that point forward, the current value is used in place of the proposed new value because the proposed new value has been replaced by the current value..

Recall that the property is also *bound* to support a list of **PropertyChangeListener** objects who have registered to be notified whenever the property value actually changes. These listener objects should not be notified of a proposed change that isn't implemented because of a veto, because in that case no change actually took place.

An **if** statement is used to determine if the property value has actually changed, and if so, the **PropertyChangeListener** objects are notified. In the case of an actual change in the property value, the background color of the bean is also changed to reflect the new value. If there was no actual change, the background color is not changed.

This strategy is implemented by the code in the following method.

```
void processTheColors(){
  try{//test to see if anyone vetos the new color
    notifyVetoableChange("theColor");
  }catch(PropertyVetoException exception){
    //Someone vetoed the new color.  Don't use newColor.
    newColor = oldColor;// Restore oldColor instead
    //Display the veto exception
    System.out.println(exception);
  }//end catch

  if(!newColor.equals(oldColor)){//if color changed
    this.setBackground(newColor);//display new color
    //notify property listeners of property change
    notifyPropertyChange("theColor");
  }//end if
}//end process the colors
```

As a result of the use of design patterms, the following "set" and "get" methods, in conjunction with the instance variables named **oldColor** and **newColor**, constitute a property named **theColor**. Note the call to the above method named **processTheColors()** inside the **setTheColor()** method. This call deals with the possibility of a veto of the proposed new **Color** value.

```
public synchronized void setTheColor(Color inColor){
  oldColor = newColor;//save current color
  newColor = inColor;//proposed new color
```

```
    processTheColors();//go process the proposed new color

  }//end setTheColor()

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

Because they are **public**, the following two methods are exposed to the builder tool as accessible methods. These two methods attempt to change the value of the **Color** property and are subject to the possibility of a veto. Note the calls to the **processTheColors()** method (discussed earlier) which handles that possibility.

```
  public synchronized void makeBlue(){
    oldColor = newColor;//save current color
    newColor = Color.blue;//establish proposed new color

    processTheColors();//go process the proposed new color

  }//end makeBlue()
  //-----------------------------------------------------//

  public synchronized void makeRed(){
    oldColor = newColor;//save current color
    newColor = Color.red;//establish proposed new color

    processTheColors();//go process the proposed new color

  }//end makeRed()
```

The following two methods are used to maintain a list of **PropertyChangeListener** objects. These are listener objects that have been registered to be notified whenever there is a change in a *bound* property.

Note that unlike a sample program in a previous lesson where we "rolled our own" list-maintenance capability, these methods simply make use of the corresponding list-maintenance methods in the previously instantiated object of type **PropertyChangeSupport** that is referenced by the reference variable named **changeSupportObj**. This results in a significant reduction in programming effort on our part.

```
  //Add a property change listener object to the list.
  public synchronized void addPropertyChangeListener(
                           PropertyChangeListener listener){
    //Pass the task on to the support class method.
    changeSupportObj.addPropertyChangeListener(listener);
```

```
  }//end addPropertyChangeListener
  //-----------------------------------------------------//

  //Remove a property change listener from the list.
  public synchronized void removePropertyChangeListener(
                          PropertyChangeListener listener){
    //Pass the task on to the support class method.
    changeSupportObj.removePropertyChangeListener(listener);
  }//end removePropertyChangeListener()
```

The following two methods are used to maintain a list of **VetoableChangeListener** objects. These are listener objects that have been registered to be notified whenever there is a ***proposed*** change in a *constrained* property. These listener objects have the right to veto the proposed change and prevent it from being implemented.

Similar to the previous discussion, these methods make use of the corresponding list-maintenance methods in the previously instantiated object of type **VetoableChangeSupport** that is referenced by the reference variable named **vetoSupportObj**. Not having to create our own list-maintenance code results in a significant reduction in programming effort on our part.

```
  //Add a vetoable change listener object to the list.
  public synchronized void addVetoableChangeListener(
                          VetoableChangeListener listener){
    //Pass the task on to the support class method.
    vetoSupportObj.addVetoableChangeListener(listener);
  }//end addVetoableChangeListener
  //-----------------------------------------------------//

  //Remove a vetoable change listener from the list.
  public synchronized void removeVetoableChangeListener(
                          VetoableChangeListener listener){
    //Pass the task on to the support class method.
    vetoSupportObj.removeVetoableChangeListener(listener);
  }//end removeVetoableChangeListener()
```

The following method is used to notify **PropertyChangeListener** objects of changes in the properties. The incoming parameter is the name of the property that has changed. That property name is encapsulated in the object that is passed when the event is fired, and can be used by the listener object to differentiate between different *bound* properties.

Note that this method makes use of the **firePropertyChange()** method of an object of the **PropertyChangeSupport** class to actually fire the event. This eliminates the requirement for us to write our own code to fire the events to all the objects on the list of registered objects.

```
  protected void notifyPropertyChange(
                                  String changedProperty){
    //Pass the task on to the support class method.
```

```
    changeSupportObj.firePropertyChange(
                          changedProperty,oldColor,newColor);
  }//end notifyPropertyChange()
```

The following method is used to notify **VetoableChangeListener** objects of ***proposed*** changes in the property values. The incoming parameter is the name of the property that is proposed to be changed. This property name is encapsulated in the object that is passed to the listener object when the event is fired.

This method uses the **fireVetoableChange()** method of the **VetoableChangeSupport** class to actually fire the event. As discussed earlier, the **fireVetoableChange()** method actually performs some data processing and does more than simply fire the event. In particular, if the proposed change is vetoed by a listener object, another round of events is fired to "roll back" the value to the value of the property before the proposed change.

```
  protected void notifyVetoableChange(
                           String vetoableProperty)
                               throws PropertyVetoException{
    //Pass the task on to the support class method.
    vetoSupportObj.fireVetoableChange(
                         vetoableProperty,oldColor,newColor);

  }//end notifyVetoableChange()
```

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 Beans06.java Copyright 1997, R.G.Baldwin

This program was designed to be compiled and executed
under JDK 1.1.3 or later.

This program illustrates the use of beans with a property
which is both bound and constrained.

The bean has one property named theColor.  Two separate
instance variables named oldColor and newColor are used
to maintain the value of the property.

Applying introspection to the bean yields the following:

Name of bean:  Beans06
Class of bean: class Beans06

==== Properties: ====
Name: preferredSize
```

```
 Type:       class java.awt.Dimension
 Get method: public synchronized java.awt.Dimension
                                    Beans06.getPreferredSize()
 Set method: null
Name: theColor
 Type:       class java.awt.Color
 Get method: public synchronized java.awt.Color
                                          Beans06.getTheColor()
 Set method: public synchronized void
                          Beans06.setTheColor(java.awt.Color)

==== Events: ====
Event Name: vetoableChange
 Add Method:    public synchronized void
                       Beans06.addVetoableChangeListener(
                          java.beans.VetoableChangeListener)
 Remove Method: public synchronized void
                       Beans06.removeVetoableChangeListener(
                          java.beans.VetoableChangeListener)
 Event Type: vetoableChange

Event Name: propertyChange
 Add Method:    public synchronized void
                       Beans06.addPropertyChangeListener(
                          java.beans.PropertyChangeListener)
 Remove Method: public synchronized void
                       Beans06.removePropertyChangeListener(
                          java.beans.PropertyChangeListener)
 Event Type: propertyChange

==== Methods: ====
makeRed
removePropertyChangeListener
getTheColor
setTheColor
removeVetoableChangeListener
getPreferredSize
addPropertyChangeListener
makeBlue
addVetoableChangeListener

The most significant new addition to this bean class is
the ability for a listener to veto a proposed change in the
value of a property.  When that happens, the proposed new
property value is discarded and the actual property value
is not changed.

  The following methods:
    setTheColor()
    makeRed()
    makeBlue()

  all provide the opportunity for a VetoableChangeListener
  object to veto a proposed new color value for the
  property named theColor.
```

Each of these methods receives a proposed new color value as a parameter when it is invoked. The method saves the current value of the property in the instance variable named oldColor. Then it makes a call to the method named notifyVetoableChange() inside a try block.

The notifyVetoableChange broadcasts a vetoableChange() event to all of the VetoableChangeListener objects that are registered to receive such an event. Any listener object that wants to veto the change throws a PropertyVetoException which finds its way back to method listed above that invoked notifyVetoableChange() in the first place.

When the exception is thrown, it is caught in a catch block. The code in the catch block restores the property value to its original value and displays the exception. In other words, the proposed new value is replaced by the value of the property before the proposed new value was received.

This proposed new value then becomes the current value and is used to set the background color of the bean. The proposed new value is also compared with the value of the property before the proposed new value was received. If they are different, meaning that a property change has occurred, the notifyPropertyChange() method is invoked to broadcast a propertyChange() event to all PropertyChangeListener objects registered to receive such an event.

An important aspect of the behavior of this bean is based on the use of the fireVetoableChange() method of the VetoableChangeSupport class to actually broadcast the event. A description of this method follows. Pay particular attention to the behavior of the method in the event that someone wants to veto the change.

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

Report a vetoable property update to any registered listeners. If anyone vetos the change, then fire a new event reverting everyone to the old value and then rethrow the PropertyVetoException.

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.

```
    Throws: PropertyVetoException
        if the recipient wishes the property change to be
        rolled back.

//=====================================================//
*/

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 Beans06 extends Canvas
                                    implements Serializable{

  //The following instance variables are used to store the
  // current property value and a proposed new property
  // value.
  protected Color oldColor;
  protected Color newColor;

  //The following reference variables are used to access
  // the list maintenance and event firing capabilities
  // of the PropertyChangeSupport and VetoableChangeSupport
  // classes. An object of each of these classes is
  // instantiated in the constructor.
  PropertyChangeSupport changeSupportObj;
  VetoableChangeSupport vetoSupportObj;
  //-----------------------------------------------------//

  public Beans06(){//constructor
    //This bean is a visible square that is initialized to
    // yellow and can then be changed to red or blue by
    // invoking methods of the class named makeRed() and
    // makeBlue().
    //The color can be set to any color by invoking the
    // setTheColor() method and passing a color in as a
    // parameter.

    //Initialize the color of the square.
    newColor = Color.yellow;
    setBackground(newColor);

    //Instantiate objects of the support classes to handle
    // list maintenance and event firing tasks.  The
    // constructor for these support classes requires this
    // object as the source of the events.
    changeSupportObj = new PropertyChangeSupport(this);
    vetoSupportObj = new VetoableChangeSupport(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()
  //-------------------------------------------------//

  //This common method is invoked by all three property-
  // changing methods to process the proposed new color.
  void processTheColors(){
      try{//test to see if anyone vetos the new color
      notifyVetoableChange("theColor");
    }catch(PropertyVetoException exception){
      //Someone vetoed the new color.  Don't use newColor.
      newColor = oldColor;// Restore oldColor instead
      //Display the veto exception
      System.out.println(exception);
    }//end catch

    if(!newColor.equals(oldColor)){//if color changed
      this.setBackground(newColor);//display new color
      //notify property listeners of property change
      notifyPropertyChange("theColor");
    }//end if
  }//end process the colors
  //-------------------------------------------------//

  //The following "set" and "get" methods in conjunction
  // with the instance variable named oldColor constitute a
  // property named theColor.
  public synchronized void setTheColor(Color inColor){
    oldColor = newColor;//save current color
    newColor = inColor;//proposed new color

    processTheColors();//go process the proposed new color

  }//end setTheColor()

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

  //The following two methods are exposed to the builder
  // tool as accessible methods.
  public synchronized void makeBlue(){
    oldColor = newColor;//save current color
    newColor = Color.blue;//establish proposed new color

    processTheColors();//go process the proposed new color

  }//end makeBlue()

  public synchronized void makeRed(){
    oldColor = newColor;//save current color
    newColor = Color.red;//establish proposed new color

    processTheColors();//go process the proposed new color
```

```
}//end makeRed()
//-------------------------------------------------------//

//The following two methods are used to maintain a list
// of PropertyChangeListener objects who request to be
// added to the list or who request to be removed from
// the list.

//Add a property change listener object to the list.
public synchronized void addPropertyChangeListener(
                        PropertyChangeListener listener){
  //Pass the task on to the support class method.
  changeSupportObj.addPropertyChangeListener(listener);
}//end addPropertyChangeListener
//-------------------------------------------------------//

//Remove a property change listener from the list.
public synchronized void removePropertyChangeListener(
                        PropertyChangeListener listener){
  //Pass the task on to the support class method.
  changeSupportObj.removePropertyChangeListener(listener);
}//end removePropertyChangeListener()
//-------------------------------------------------------//

//The following two methods are used to maintain a list
// of listener objects who request to be registered
// as VetoableChangeListener objects, or who request to
// be removed from the list.

//Add a vetoable change listener object to the list.
public synchronized void addVetoableChangeListener(
                        VetoableChangeListener listener){
  //Pass the task on to the support class method.
  vetoSupportObj.addVetoableChangeListener(listener);
}//end addVetoableChangeListener
//-------------------------------------------------------//

//Remove a vetoable change listener from the list.
public synchronized void removeVetoableChangeListener(
                        VetoableChangeListener listener){
  //Pass the task on to the support class method.
  vetoSupportObj.removeVetoableChangeListener(listener);
}//end removeVetoableChangeListener()
//-------------------------------------------------------//

//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
// firePropertyChange() method of an object of the
// PropertyChangeSupport class to actually fire the
// event.
protected void notifyPropertyChange(
                                  String changedProperty){
  //Pass the task on to the support class method.
```

```
    changeSupportObj.firePropertyChange(
                          changedProperty,oldColor,newColor);
  }//end notifyPropertyChange()
  //-----------------------------------------------------//

  //The following method is used to notify
  // VetoableChangeListener objects of proposed changes in
  // the property values.  The incoming parameter is the
  // name of the property that is proposed to be changed.
  // This method uses the fireVetoableChange() method of
  // the VetoableChangeSupport class to actually fire the
  // event.  As discussed earlier in this file, the
  // fireVetoableChange method actually performs some data
  // processing and does more than simply fire the event.
  protected void notifyVetoableChange(
                           String vetoableProperty)
                               throws PropertyVetoException{
    //Pass the task on to the support class method.
    vetoSupportObj.fireVetoableChange(
                         vetoableProperty,oldColor,newColor);

  }//end notifyVetoableChange()

}//end class Beans06.java
//=====================================================//
```

Sample Test Program

This program was designed to be compiled and executed under JDK 1.1.3 or later. The purpose of the program is to test the constrained property aspects of the bean class named **Beans06**.

This program has been *simplified* in an attempt to make it understandable. A more realistic and complex program is provided in the *Review* section of this lesson.

You will need to refer to the comments in the source code for the **Beans06** class to fully understand how this test program works.

The visual manifestation of the **Beans06** bean is a colored square. The **bean** is placed in a **Frame** object by this test program. The square is initially **yellow**.

The bean has one property named **theColor** which controls the color of the square.

Two *exposed* methods of the bean, **makeRed()** and **makeBlue()**, can be invoked to change the color to **red** or **blue**.

Invoking the **makeRed()** or **makeBlue()** methods changes the value of the property named **theColor** which in turn changes the color of the square.

You can also change the value of the property named **theColor** by invoking the **setTheColor()** method. In this case you can pass in any color as a parameter.

The property named **theColor** is a *bound constrained* property. The bean supports a multicast list of **PropertyChangeListener** objects and also supports a multicast list of **VetoableChangeListener** objects.

PropertyChangeListener objects are simply notified whenever a change in a property value occurs.

VetoableChangeListener objects are notified of a ***proposed***change in the property value and have the opportunity to **veto** the change by raising a **PropertyVetoException**.

The program begins with the **yellow** square bean and three **Buttons** in a **Frame** on the screen. The buttons are labeled *Red*, *Green*, and *Blue*.

The Red and Blue buttons invoke the **makeRed()** and **makeBlue()** methods discussed above.

The Green button invokes the **setTheColor()** method causing the color green to be passed in as a parameter. Therefore, clicking this button will attempt to change the value of the property named **theColor** to green.

A listener class is defined which implements both the **PropertyChangeListener** interface and the **VetoableChangeListener** interface. As a result, a listener object of this class can register to be notified of ***proposed*** property changes with **veto** authority and can also register to be notified of actual changes.

One such listener object is instantiated and registered to listen for both **propertyChange()** and **vetoableChange()** events.

This object is designed to veto any proposal to change the value of the property to **green**.

Therefore, if you click the **Red** button, the square will change to red and the following will appear on the screen.

Note the use of the r, g, and b in the square brackets to indicate the contribution of each of these three primary colors to the final color. The maximum contribution of a color is indicated by a value of 255 while a value of 0 indicates no contribution of that primary color.

Note also that both the **VetoableChangeListener** object and the **PropertyChangeListener** object produce output on the screen. Later we will see that when a proposed change is vetoed, there is no output from the **PropertyChangeListener** object, and there are two separate outputs from the **VetoableChangeListener** object.

```
Veto Listener,
New property value: java.awt.Color[r=255,g=0,b=0]
Change Listener,
New property value: java.awt.Color[r=255,g=0,b=0]
```

If you click the **Blue** button, the square will change to blue and the following will appear on the screen.

```
Veto Listener,
New property value: java.awt.Color[r=0,g=0,b=255]
Change Listener,
New property value: java.awt.Color[r=0,g=0,b=255]
```

If you click the Green button, the color of the square will not change. The following will appear on the screen indicating that the proposed new color was vetoed and another event was multicast which rolled the property value back to its value before the proposed change (which in this case was blue).

Note that there was no output from the **PropertyChangeListener** object in this case, because the proposed change to green was vetoed and there was no actual change in the value of the property.

Note also that there are two outputs from the **VetoableChangeListener** object. The first output indicates the proposed new property value. The second indicates that a second round of events was fired to roll the property value back to its original value.

The last line in the output was produced by code in the **bean** proper and was a display of the contents of the exception object that was instantiated and passed by the listener object that raised the exception to veto the proposed change.

```
Veto Listener,
New property value: java.awt.Color[r=0,g=255,b=0]
Veto Listener,
New property value: java.awt.Color[r=0,g=0,b=255]
java.beans.PropertyVetoException: No green allowed
```

After the veto, if you click on the **Red** button, the color of the square will change to red in the normal manner and the following will appear on the screen:

```
Veto Listener,
New property value: java.awt.Color[r=255,g=0,b=0]
Change Listener,
New property value: java.awt.Color[r=255,g=0,b=0]
```

In all of the above examples, line breaks were manually inserted in the text to make it fit better in this format.

Although this sample program was constructed for simplicity having only one property in the bean and only one listener object, the structure of the **Beans06** bean class will support any number of listener objects in either category.

Obviously the number of properties could also be expanded to a very large number.

These expansions would require more processing on the part of the listener objects. With the structure being used, every object registered to be notified of proposed changes would be notified of every proposed change on every property and it may be necessary for the listener objects to differentiate between the properties in order to decide what to do.

Likewise, every object registered to be notified of actual changes would be notified of every actual change on every property.

Each notification event contains the name of the property to which the actual or proposed change applies. The objects could use that information to make decisions on the basis of property names and proposed changes in property values.

Note that the design pattern specifications for Java beans provide for designing more selective notification schemes, but they are not being used in this example.

Interesting Code Fragments from the Test Program

This test program places the **Beans06** object and several **Button** objects on a **Frame** object. The buttons are used to test the various aspects of the **bean**.

We're going to skip all the standard stuff that creates the **Frame** object, adds buttons to the **Frame**, instantiates listener objects for the buttons, registers the listener objects for **actionPerformed()** events on the buttons, etc.

However, we will highlight the following code fragment that instantiates a **Beans06** object and adds it to the **Frame** object.

```
    Beans06 myBean = new Beans06();
    add(myBean);//Add it to the Frame
```

The following code fragment will instantiate and register an object to listen for proposed and actual changes in the bean's property. This listener object has the ability to veto proposed changes.

This dual capability for a single listener object comes about because, as we will see later, the class of this object named **MyPropertyListenerClass** implements both the **VetoableChangeListener** interface and the **PropertyChangeListenerClass**. It also defines both the **vetoableChange()** and the **propertyChange()** methods declared in those two interfaces.

```
    MyPropertyListenerClass myListenerObject =
                            new MyPropertyListenerClass();
    myBean.addPropertyChangeListener(myListenerObject);
    myBean.addVetoableChangeListener(myListenerObject);
```

An object of the following class is instantiated and registered to listen for **actionPerformed()** events on the button labeled **"setTheColor"**.

When the **setTheColor** button is pressed, the object invokes the **setTheColor()** method on the **bean** passing in a color parameter of **green**.

Insofar as the **VetoableChangeListener** objects are concerned, this represents a proposal to change the **Color** property to green. As mentioned previously, this change will be vetoed, but that is beyond the scope of the code in this class. As far as the methods in this class are concerned, this is a direct order to set the property value of the property named **theColor** to a value representing green.

```
class SetTheColorListener implements ActionListener{
  Beans06 myBean;//save a reference to the bean here

  SetTheColorListener(Beans06 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
```

The following two classes are used to instantiate objects which are registered to listen to two of the buttons on the test panel.

When the buttons with the names corresponding to the names of the methods are pressed, these objects invoke methods of the **bean** under test.

An object of the first class invokes the **makeRed()** method and an object of the second class invokes the **makeBlue()** method.

As mentioned earlier, these methods attempt to change the value of the property named **theColor**. In the large sense, any of the **VetoableChangeListener** objects have the right to veto the proposed change. However, as this test program is structured, the change to red or blue is not vetoed and the change will be implemented causing the color of the rectangle on the **Frame** object to change colors.

```
class RedActionListener implements ActionListener{
```

```
  Beans06 myBean;//save a reference to the bean here

  RedActionListener(Beans06 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{
  Beans06 myBean;//save a reference to the bean here

  BlueActionListener(Beans06 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 listener objects that may be linked to *bound* and/or *constrained* properties of the **bean**.

When one of these objects is registered on the **bean** as a **VetoableChangeListener**, it will receive notifications of all the proposed changes to all the *constrained* properties of the **bean**. Once notified, the listener object has the right to veto the proposed change by raising a **PropertyVetoException**.

When one of these objects is registered on the **bean** as a **PropertyChangeListener**, it will receive notifications of all actual changes to all of the *bound* properties of the **bean**.

When notified of a proposed or actual change, the object displays the actual or proposed new property value.

When notified of a proposed change, the object has the ability to veto the change by raising a **PropertyVetoException**. The design of this class is such that any proposed change to the color **green** will be vetoed.

Note that this class implements both the **PropertyChangeListener** and the **VetoableChangeListener** interfaces. This makes it possible for a single object of this class to be notified both of proposed changes and actual changes to the properties of the **bean**.

Note that the constructor for the **PropertyVetoException** thrown by the **vetoableChange()** method requires two arguments. The first is a message of type **String**. The second is the **PropertyChangeEvent** object that is passed to the method and which is being vetoed.

In other words, the event describing the property change that is being vetoed is encapsulated (along with a message) and sent to the **catch** block that will eventually process the exception.

Note also that the argument that is passed to the **vetoableChange()** method is of type **PropertyChangeEvent** which is the same type that is passed to the **propertyChange()** method. In particular, the incoming event type is the same for both event handlers, and there is no such thing as a ***VetoableChangeEvent*** tailored to the **vetoableChange()** method.

```
class MyPropertyListenerClass
  implements PropertyChangeListener,VetoableChangeListener{

  public void propertyChange(PropertyChangeEvent event){
    //Extract and display the new value
    System.out.println(
                    "Change Listener, New property value: "
                                    + event.getNewValue());
  }//end propertyChange()
  //-------------------------------------------------------//

  public void vetoableChange(PropertyChangeEvent event)
                                throws PropertyVetoException{
    //Extract and display proposed new value
    System.out.println(
                      "Veto Listener, New property value: "
                                    + event.getNewValue());
    //Throw an exception on proposed value of green. This
    // will veto the change.
    if(event.getNewValue().equals(Color.green))
      throw new PropertyVetoException(
                                  "No green allowed",event);
  }//end vetoableChange()
}//end MyPropertyListenerClass 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* and *constrained* property behavior of the **bean** class.

```
/*File Beans06Test.java Copyright 1997, R.G.Baldwin

This program was designed to be compiled and executed
under JDK 1.1.3 or later.

This program is designed to test the constrained property
aspects of the bean class named Beans06.

The program has been simplified as much as possible in an
attempt to make it understandable.
```

```
You will need to refer to the comments in the source code
for the Beans06 bean class to fully understand how this
test program works.

The visual manifestation of the Beans06 bean is a colored
square.  The bean is placed in a Frame object by this
test program.  The square is initially yellow.

The bean has one property named theColor which controls
the color of the square.

Two exposed methods of the bean, makeRed() and makeBlue(),
can be invoked to change the color to red or blue.

Invoking the makeRed() or makeBlue() methods changes the
value of the property named theColor which in turn changes
the color of the square.

You can also change the value of the property named
theColor by invoking the setTheColor() method.  In this
case you can pass in any color as a parameter.

The property named theColor is a bound constrained
property.  The bean supports a multicast list of
PropertyChangeListener objects and also supports a
multicast list of VetoableChangeListener objects.

PropertyChangeListener objects are simply notified
whenever a change in a property value occurs.

VetoableChangeListener objects are notified of a proposed
change in the property value and have the opportunity to
veto the change by raising a PropertyVetoException.

The program begins with the yellow square bean and three
buttons in a frame on the screen. The buttons are labeled
Red, Green, and Blue.

The Red and Blue buttons invoke the makeRed() and
makeBlue() methods discussed above.

The Green button invokes the setTheColor() method causing
the color green to be passed in as a parameter.
Therefore, clicking this button will attempt to change the
value of the property named theColor to green.

A listener class is defined which implements both the
PropertyChangeListener interface and the
VetoableChangeListener interface.  As a result, a listener
object of this class can register to be notified of
proposed property changes with veto authority and can also
register to be notified of actual changes.

One such listener object is instantiated and registered
to listen for both propertyChange() and vetoableChange()
```

events.

This object is designed to veto any proposal to change the value of the property to green.

Therefore, if you click the Red button, the square will turn to red and the following will appear on the screen:

```
Veto Listener,
New property value: java.awt.Color[r=255,g=0,b=0]
Change Listener,
New property value: java.awt.Color[r=255,g=0,b=0]
```

If you click the Blue button, the square will change to blue and the following will appear on the screen.

```
Veto Listener,
New property value: java.awt.Color[r=0,g=0,b=255]
Change Listener,
New property value: java.awt.Color[r=0,g=0,b=255]
```

If you click the Green button, the color of the square will not change. The following will appear on the screen indicating that the proposed new color was vetoed and another event was multicast which rolled the property value back to its value before the proposed change.

```
Veto Listener,
New property value: java.awt.Color[r=0,g=255,b=0]
Veto Listener,
New property value: java.awt.Color[r=0,g=0,b=255]
java.beans.PropertyVetoException: No green allowed
```

If you then click on the Red button, the color of the square will change to red and the following will appear on the screen:

```
Veto Listener,
New property value: java.awt.Color[r=255,g=0,b=0]
Change Listener,
New property value: java.awt.Color[r=255,g=0,b=0]
```

In all of the above examples, line breaks were manually inserted in the text to make it fit better in this format.

Although this sample program was constructed for simplicity having only one property in the bean and one listener object, the structure of the Beans06 bean class will support any number of listener objects in either category.

Obviously the number of properties could also be expanded to a very large number. This would require more processing on the part of the objects. With the structure being used, every object registered to be notified of proposed changes would be notified of every

```
proposed change on every property.  Likewise, every object
registered to be notified of actual changes would be
notified of every actual change on every property.

Each notification event contains the name of the property
to which the actual or proposed change applies.  The
objects would have to use that information to make
decisions on the basis of property names and proposed
changes in property values.

============================================================*/

import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.util.*;
//============================================================//

public class Beans06Test extends Frame{
  public static void main(String[] args){
    new Beans06Test();
  }//end main
  //------------------------------------------------------//

  public Beans06Test(){//constructor
    setTitle("Copyright 1997, R.G.Baldwin");
    setLayout(new FlowLayout());
    //instantiate a Bean object
    Beans06 myBean = new Beans06();
    add(myBean);//Add it to the Frame

    //Instantiate several test buttons
    Button buttonToSetTheColor = new Button("Green");
    Button buttonToInvokeRedMethod = new Button("Red");
    Button buttonToInvokeBlueMethod = new Button("Blue");

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

    //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));
    buttonToInvokeRedMethod.addActionListener(
                            new RedActionListener(myBean));
    buttonToInvokeBlueMethod.addActionListener(
                           new BlueActionListener(myBean));


    //Instantiate and register an object to listen for
```

```
    // proposed and actual changes in the bean's property.
    // This listener object has the ability to veto
    // proposed changes.
    MyPropertyListenerClass myListenerObject =
                            new MyPropertyListenerClass();
    myBean.addPropertyChangeListener(myListenerObject);
    myBean.addVetoableChangeListener(myListenerObject);

    //Terminate program when Frame is closed
    this.addWindowListener(new Terminate());
  }//end constructor
}//end class Beans06Test
//=======================================================//

//An object of this class is instantiated and registered
// to listen for actionPerformed() events on the button
// labeled "setTheColor".

// When the setTheColor button is pressed, the object
// invokes the setTheColor() method on the bean passing in
// a color parameter of green.

class SetTheColorListener implements ActionListener{
  Beans06 myBean;//save a reference to the bean here

  SetTheColorListener(Beans06 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
//-------------------------------------------------------//

//The following two classes are used to instantiate objects
// which are registered to listen to two of the buttons on
// the test panel.  When the corresponding  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{
  Beans06 myBean;//save a reference to the bean here

  RedActionListener(Beans06 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{
```

```
  Beans06 myBean;//save a reference to the bean here

  BlueActionListener(Beans06 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 proposed changes and actual changes in the
// property values in the bean object.

//When notified of a proposed or actual change, the object
// displays the actual or proposed new property value.

//When notified of a proposed change, the object has the
// ability to veto the change by raising a
// PropertyVetoException.  The design of this class is
// such that any proposed change to the color green will
// vetoed.

class MyPropertyListenerClass
  implements PropertyChangeListener,VetoableChangeListener{

  public void propertyChange(PropertyChangeEvent event){
    //Extract and display the new value
    System.out.println(
                    "Change Listener, New property value: "
                                    + event.getNewValue());
  }//end propertyChange()

  public void vetoableChange(PropertyChangeEvent event)
                                throws PropertyVetoException{
    //Extract and display proposed new value
    System.out.println(
                      "Veto Listener, New property value: "
                                    + event.getNewValue());
    //Throw an exception on proposed value of green. This
    // will veto the change.
    if(event.getNewValue().equals(Color.green))
      throw new PropertyVetoException(
                                  "No green allowed",event);
  }//end vetoableChange()
}//end MyPropertyListenerClass 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 - Without viewing the following solution, upgrade the programs named **Beans06.java** and **Beans06Test.java** to demonstrate the use of multiple *constrained* and *bound* properties in the **bean** and the use of multiple listener objects which implement the **VetoableChangeListener** and **PropertyChangeListener** interfaces in the test program.

Cause your **bean** to appear on the screen as a colored rectangle containing a date and time. Provide a **Color** property to control the background color of the rectangle. Provide a **Date** property to control the date and time that is displayed in the rectangle. Make both of the properties *bound* and *constrained*.

Design one of your **VetoableChangeListener** objects to veto proposed property changes which would otherwise cause the rectangle to be green. Design the other **VetoableChangeListener** object to veto proposed property changes that would cause the rectangle to be orange. Demonstrate that other colors are allowed.

Provide buttons on your test panel which will attempt to cause the color of the rectangle to be red, green, blue, or orange.

Also provide a button on your test panel that will set the current date and time in the **Date** property.

Provide an output on the standard output device whenever the colors green or orange are vetoed, identifying the reason for the veto and the object that raised the veto.

Provide an output on the standard output device whenever the value of a property actually changes, identifying the new value of the property, and the identification of the listener object that recognized the change.

A - See the following bean program and test program.

Bean program follows:

```
/*File Beans07.java Copyright 1997, R.G.Baldwin
This program was designed to be compiled and executed
under JDK 1.1.3 or later.

This program illustrates the use of beans with multiple
properties which are both bound and constrained.

This bean appears on the screen as a colored rectangle
```

```
containing a date and time.  The color as well as the date
and time are based on the current values of corresponding
properties.

The bean has a Color property named theColor and a Date
property named theDate. Both of these properties are bound
and constrained.

The bean has a property named preferredSize that is
neither bound nor constrained

This bean supports propertyChange and vetoableChange
notification lists for the property values.  Other
objects can register to be notified of a proposed change
in property values and can veto the change.  A proposed
change that is vetoed does not take place.  Other
objects can also register to be notified of actual
changes in property values.


A description of the bean as determined by the program
named Introspect01 follows:
-------
Name of bean:  Beans07
Class of bean: class Beans07

==== Properties: ====
Name: preferredSize
 Type:       class java.awt.Dimension
 Get method: public synchronized java.awt.Dimension
                                Beans07.getPreferredSize()
 Set method: null
Name: theDate
 Type:       class java.util.Date
 Get method: null
 Set method: public synchronized void
                       Beans07.setTheDate(java.util.Date)
Name: theColor
 Type:       class java.awt.Color
 Get method: null
 Set method: public synchronized void
                      Beans07.setTheColor(java.awt.Color)

==== Events: ====
Event Name: vetoableChange
 Add Method:    public synchronized void
                   Beans07.addVetoableChangeListener(
                       java.beans.VetoableChangeListener)
 Remove Method: public synchronized void
                   Beans07.removeVetoableChangeListener(
                       java.beans.VetoableChangeListener)
 Event Type: vetoableChange

Event Name: propertyChange
 Add Method:    public synchronized void
                   Beans07.addPropertyChangeListener(
```

```
                        java.beans.PropertyChangeListener)
 Remove Method: public synchronized void
                   Beans07.removePropertyChangeListener(
                        java.beans.PropertyChangeListener)
 Event Type: propertyChange

==== Methods: ====
setTheDate
removePropertyChangeListener
setTheColor
removeVetoableChangeListener
getPreferredSize
addPropertyChangeListener
addVetoableChangeListener
-------

  The following methods:
    setTheColor()
    setTheDate()

  both provide the opportunity for a
  VetoableChangeListener object to veto a proposed new
  value for the property.

  Each of these methods receives a proposed property value
  as a parameter when it is invoked. The method saves the
  current value of the property in an instance variable
  named old_____.  Then it makes a call to the method
  named notifyVetoableChange() inside a try block.

  notifyVetoableChange() broadcasts a vetoableChange()
  event to all of the VetoableChangeListener objects that
  are registered to receive such an event.  Any listener
  object that wants to veto the change throws a
  PropertyVetoException which finds its way back to the
  method listed above that invoked notifyVetoableChange()
  in the first place.

  When the exception is thrown, it is caught in a catch
  block.  The code in the catch block restores the
  property value to its original value. In other words,
  theproposed new value is discarded and replaced by the
  value of the property before the proposed new value
  was received.

  This proposed new value then becomes the current value
  and is used to set the background color of the bean
  or to set the new date and time.

  The proposed new value is also compared with the value
  of the property before the proposed new value was
  received.  If they are different, meaning that a
  property change has occurred, the notifyPropertyChange()
  method is invoked to broadcase a propertyChange() event
  to all PropertyChangeListener objects registered to
  receive such an event.
```

```
  An important aspect of the behavior of this bean is
  based on the use of the fireVetoableChange() method of
  the VetoableChangeSupport class to actually broadcast
  the event. A description of this method follows.

  Note in particular the behavior of this method when
  someone vetos the change.

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


    Report a vetoable property update to any registered
    listeners. If anyone vetos the change, then fire a
    new event reverting everyone to the old value and
    then rethrow the PropertyVetoException.

    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.
    Throws: PropertyVetoException
        if the recipient wishes the property change to be
        rolled back.


Additional comments describing the bean are scattered
throughout the code.

This bean was tested using the test program named
Beans07Test using JDK 1.1.3 under Win95.
//=======================================================//
*/

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 Beans07 extends Label
                                  implements Serializable{

  //The following instance variables are used to store the
  // current property value and proposed new property
  // value for both the Color and Date properties.
  protected Color oldColor;
  protected Color newColor;
```

```
  protected Date oldDate;
  protected Date newDate;

  //The following reference variables are used to access
  // the list maintenance and event firing capabilities
  // of the PropertyChangeSupport and VetoableChangeSupport
  // classs. An object of each of these classes is
  // instantiated in the constructor.
  PropertyChangeSupport changeSupportObj;
  VetoableChangeSupport vetoSupportObj;
  //-----------------------------------------------------//

  public Beans07(){//constructor
    //Initialize the property values and the display
    newColor = Color.yellow;
    setBackground(newColor);

    newDate = new Date();
    setText(newDate.toString());

    //Instantiate objects of the support classes to handle
    // list maintenance and event firing tasks.  The
    // constructor for the support classes requires this
    // object as a paremeter.  The parameter is used as the
    // source of the events when they are fired.
    changeSupportObj = new PropertyChangeSupport(this);
    vetoSupportObj = new VetoableChangeSupport(this);
  }//end constructor
  //-----------------------------------------------------//

  //This method defines the preferred display size of the
  // bean object.
  public synchronized Dimension getPreferredSize(){
    return new Dimension(200,50);
  }//end getPreferredSize()
  //-----------------------------------------------------//

  //The following "set" method in conjunction with the
  // instance variables named oldColor and newColor
  // constitute a write-only property named theColor.
  public synchronized void setTheColor(Color inColor){
    oldColor = newColor;//save current color
    newColor = inColor;//proposed new color

    try{//test to see if anyone vetos the new color
      notifyVetoableChange("theColor");
    }catch(PropertyVetoException exception){
      //Someone vetoed the new color.  Don't use newColor.
      newColor = oldColor;// Restore oldColor instead
    }//end catch

    if(!newColor.equals(oldColor)){//if color changed
      this.setBackground(newColor);//display new color
      //notify property listeners of property change
      notifyPropertyChange("theColor");
    }//end if
```

```
  }//end setTheColor()

  //-------------------------------------------------------//
  //The following "set" method in conjunction with the
  // instance variables named oldDate and newDate
  // constitute a write-only property named theDate.
  public synchronized void setTheDate(Date inDate){
    oldDate = newDate;//save current date
    newDate = inDate;//proposed new date

    try{//test to see if anyone vetos the new date
      notifyVetoableChange("theDate");
    }catch(PropertyVetoException exception){
      //Someone vetoed the new date.  Don't use newDate.
      newDate = oldDate;// Restore oldDate instead
      //Display the veto exception
      System.out.println(exception.getMessage());
    }//end catch

    if(!newDate.equals(oldDate)){//if date changed
      this.setText(newDate.toString());//display new date
      //notify property listeners of property change
      notifyPropertyChange("theDate");
    }//end if
  }//end setTheColor()
  //-------------------------------------------------------//

  //The following two methods are used to maintain a list
  // of listener objects who request to be registered as
  // PropertyChangeListener objects, or who request to be
  // removed from the list.

  //Add a property change listener object to the list.
  public synchronized void addPropertyChangeListener(
                          PropertyChangeListener listener){
    //Pass the task to the support class.
    changeSupportObj.addPropertyChangeListener(listener);
  }//end addPropertyChangeListener
  //-------------------------------------------------------//

  //Remove a property change listener from the list.
  public synchronized void removePropertyChangeListener(
                          PropertyChangeListener listener){
    //Pass the task to the support class.
    changeSupportObj.
                    removePropertyChangeListener(listener);
  }//end removePropertyChangeListener()
  //-------------------------------------------------------//

  //The following two methods are used to maintain a list
  // of listener objects who request to be registered
  // as VetoableChangeListener objects, or who request to
  // be removed from the list.

  //Add a vetoable change listener object to the list.
```

```
public synchronized void addVetoableChangeListener(
                        VetoableChangeListener listener){
  //Pass the task to the support class.
  vetoSupportObj.addVetoableChangeListener(listener);
}//end addVetoableChangeListener
//-------------------------------------------------------//

//Remove a vetoable change listener from the list.
public synchronized void removeVetoableChangeListener(
                        VetoableChangeListener listener){
  //Pass the task to the support class.
  vetoSupportObj.removeVetoableChangeListener(listener);
}//end removeVetoableChangeListener()
//-------------------------------------------------------//

//The following method is used to notify
// PropertyChangeListener objects of changes in the
// properties.  The incoming parameter is the name of the
// property that has changed.
protected void notifyPropertyChange(
                                  String changedProperty){
  if(changedProperty.compareTo("theColor") == 0)
    //TheColor property has changed, pass color info
    changeSupportObj.firePropertyChange(
                      changedProperty,oldColor,newColor);
    else //TheDate property has changed, pass date info
    changeSupportObj.firePropertyChange(
                      changedProperty,oldDate,newDate);

}//end notifyPropertyChange()
//-------------------------------------------------------//

//The following method is used to notify
// VetoableChangeListener objects of proposed changes in
// the property values.  The incoming parameter is the
// name of the property that is proposed to be changed.

// This method uses the fireVetoableChange() method of
// the VetoableChangeSupport class to actually fire the
// event.  As discussed earlier in this file, the
// fireVetoableChange method actually performs some data
// processing and does more than simply fire the event.
// In the event of a veto, it fires a second event with
// the value of the property that existed prior to the
// proposed change.
protected void notifyVetoableChange(
                        String vetoableProperty)
                            throws PropertyVetoException{
  if(vetoableProperty.compareTo("theColor") == 0)
    //theColor property is proposed to be changed
    vetoSupportObj.fireVetoableChange(
                      vetoableProperty,oldColor,newColor);
  else //theDate property is proposed to be changed
    vetoSupportObj.fireVetoableChange(
                      vetoableProperty,oldDate,newDate);
}//end notifyVetoableChange()
```

```
}//end class Beans07.java
//=========================================================//
```

Test program follows:

```
/*File Beans07Test.java Copyright 1997, R.G.Baldwin
This program was designed to be compiled and executed
under JDK 1.1.3 or later.

This program is designed to test the constrained property
aspects of the bean class named Beans07 for multiple
properties and multiple listener objects.

You will need to refer to the comments in the source code
for the Beans07 bean class to fully understand how this
test program works.

The visual manifestation of the Beans07 bean is a colored
rectangle with a date and time displayed in the rectangle.

The bean is placed in a Frame object by this test program.

The rectangle is initially yellow.

The bean has two bound and constrained properties named
theColor and theDate which control the color of the
rectangle and the date and time displayed in the rectangle.

You can change the color of the rectangle by invoking the
setTheColor() method on the bean and passing in a Color as
a parameter.

You can change the date and time displayed in the
rectangle by invoking the setTheDate() method on the bean
and passing a Date object as a parameter.

The bean supports a multicast list of
PropertyChangeListener objects and also supports a
multicast list of VetoableChangeListener objects for both
of the bound and constrained properties.

PropertyChangeListener objects are simply notified
whenever a change in a property value occurs.

VetoableChangeListener objects are notified of a proposed
change in the property value and have the opportunity to
veto the change by raising a PropertyVetoException.

This program begins with a yellow rectangular bean
containing a date and time along with five buttons in a
frame on the screen. The buttons are labeled Red, Green,
Blue, Orange, and Date.
```

Clicking one of the buttons with a color label causes the setTheColor() method to be invoked on the bean with the indicated color being passed in as a parameter.

Clicking the date button causes the setTheDate() method to be invoked on the bean, passing in a Date object containing the current date and time.

A listener class is defined which implements both the PropertyChangeListener interface and the VetoableChangeListener interface. As a result, a listener object of this class can register to be notified of proposed property changes with veto authority and can also register to be notified of actual changes.

The constructor for this listener class also allows a String object and a Color value to be passed in as a parameter.

The String object is used as an identifier when information about the listener object is displayed.

The Color value is used to establish a color that will be vetoed by the listener object.

Two such listener objects are instantiated and registered to listen for both propertyChange() and vetoableChange() events.

One object is named Joe and will veto attempts to change the Color property to green.

The other object is named Tom and will veto attempts to change the Color property to orange.

If you click the Red button, the rectangle will change to red and the following will appear on the screen:

```
Joe Change Listener
  New property value: java.awt.Color[r=255,g=0,b=0]
Tom Change Listener
  New property value: java.awt.Color[r=255,g=0,b=0]
```

If you click the Green button, the color of the rectangle will not change. The following will appear on the screen indicating that the proposed new color was vetoed.

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

If you click the Blue button, the rectangle will change to blue and the following will appear on the screen.

```
Joe Change Listener
```

```
  New property value: java.awt.Color[r=0,g=0,b=255]
Tom Change Listener
  New property value: java.awt.Color[r=0,g=0,b=255]


If you click the Orange button, the color of the rectangle
will not change.  The following will appear on the screen
indicating that the proposed new color was vetoed.

Tom vetos java.awt.Color[r=255,g=200,b=0]


If you then click on the Red button, the color of the
rectangle will change to red and the following will appear
on the screen:

Joe Change Listener
  New property value: java.awt.Color[r=255,g=0,b=0]
Tom Change Listener
  New property value: java.awt.Color[r=255,g=0,b=0]


If you click on the Date button, the new date and time will
appear in the colored rectangle and the following will
appear on the screen:

Joe Change Listener
  New property value: Sun Oct 19 15:14:07 CDT 1997
Tom Change Listener
  New property value: Sun Oct 19 15:14:07 CDT 1997



============================================================*/

import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.util.*;
//===========================================================//
public class Beans07Test extends Frame{
  public static void main(String[] args){
    new Beans07Test();
  }//end main
  //-----------------------------------------------------------//

  public Beans07Test(){//constructor
    setTitle("Copyright 1997, R.G.Baldwin");
    setLayout(new FlowLayout());
    //instantiate a Bean object
    Beans07 myBean = new Beans07();
    add(myBean);//Add it to the Frame

    //Instantiate several test buttons
    Button buttonToSetToGreen = new Button("Green");
    Button buttonToSetToRed = new Button("Red");
```

```
    Button buttonToSetToBlue = new Button("Blue");
    Button buttonToSetToOrange = new Button("Orange");
    Button buttonToSetTheDate = new Button("Date");

    //Add the test buttons to the frame
    add(buttonToSetToRed);
    add(buttonToSetToGreen);
    add(buttonToSetToBlue);
    add(buttonToSetToOrange);
    add(buttonToSetTheDate);

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

    //Register action listener objects for all the test
    // buttons
    buttonToSetToGreen.addActionListener(
              new SetTheColorListener(myBean,Color.green));
    buttonToSetToRed.addActionListener(
              new SetTheColorListener(myBean,Color.red));
    buttonToSetToBlue.addActionListener(
              new SetTheColorListener(myBean,Color.blue));
    buttonToSetToOrange.addActionListener(
             new SetTheColorListener(myBean,Color.orange));
    buttonToSetTheDate.addActionListener(
              new SetTheDateListener(myBean));

    //Instantiate and register  objects to listen for
    // proposed and actual changes in the bean's property
    // values. These listener objects havethe ability to
    // veto proposed changes.
    //This object is named Joe and vetos the green color
    MyPropertyListenerClass joeListenerObject =
            new MyPropertyListenerClass("Joe",Color.green);
    myBean.addPropertyChangeListener(joeListenerObject);
    myBean.addVetoableChangeListener(joeListenerObject);

    //This object is named Tom and vetos the orange color
    MyPropertyListenerClass tomListenerObject =
          new MyPropertyListenerClass("Tom",Color.orange);
    myBean.addPropertyChangeListener(tomListenerObject);
    myBean.addVetoableChangeListener(tomListenerObject);

    //Terminate program when Frame is closed
    this.addWindowListener(new Terminate());
  }//end constructor
}//end class Beans07Test
//=======================================================//
//An object of this class will invoke the setTheColor()
// method on the bean passing a specified color as a
// parameter.  The specified color is passed as a parameter
// to the constructor of this class.
class SetTheColorListener implements ActionListener{
  Beans07 myBean;//save a reference to the bean here
  Color colorToSet;//save the new color here
```

```
  //constructor
  SetTheColorListener(Beans07 inBean,Color inColor){
    myBean = inBean;//save a reference to the bean
    colorToSet = inColor;//save the new color
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.setTheColor(colorToSet);
  }//end actionPerformed()
}//end class SetTheColorListener
//=======================================================//

//An object of this class will invoke the setTheDate()
// method on the bean passing a Date object as a parameter.
// The date object is constructed to contain the current
// date and time.
class SetTheDateListener implements ActionListener{
  Beans07 myBean;//save a reference to the bean here

  //constructor
  SetTheDateListener(Beans07 inBean){
    myBean = inBean;//save a reference to the bean
  }//end constructor

  public void actionPerformed(ActionEvent e){
    myBean.setTheDate(new Date());
  }//end actionPerformed()
}//end class SetTheDateListener
//=======================================================//

//The following class is used to instantiate objects that
// will be bound to the bean in such a way as to be
// notified of proposed changes and actual changes in the
// property values in the bean object.

//When notified of a proposed or actual change, the object
// displays the actual or proposed new property value.

//When notified of a proposed change, the object has the
// ability to veto the change by raising a
// PropertyVetoException.

//The constructor for this class accepts a String object
// and a Color parameter as incoming parameters.  The
// String is used to identify the object when information
// is displayed.  The Color parameter specifies a color
// that will be vetoed by the object if an attempt is
// made to change the Color property of the bean to that
// Color value.

//Note that this class implements PropertyChangeListener
// and VetoableChangeListener

class MyPropertyListenerClass
  implements PropertyChangeListener,VetoableChangeListener{
```

```
  String objID; //store the object ID here
  Color vetoColor; //store the color to be vetoed here

  //constructor
  MyPropertyListenerClass(String idIn,Color vetoColorIn){
    objID = idIn;//save the object ID
    vetoColor = vetoColorIn;//save the veto color
  }//end constructor

  public void propertyChange(PropertyChangeEvent event){
    //Extract and display the new value
    System.out.println(
         objID + " Change Listener\n  New property value: "
                                    + event.getNewValue());
  }//end propertyChange()

  public void vetoableChange(PropertyChangeEvent event)
                              throws PropertyVetoException{
    if(event.getNewValue() == vetoColor){//test for veto
      System.out.println(
                  objID + " vetos " + event.getNewValue());
      //Throw an exception on proposed change. This will
      // veto the change.
      throw new PropertyVetoException("VETO",event);
    }//end if
  }//end vetoableChange()
}//end MyPropertyListenerClass 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-