Swing from A to Z: Analyzing Swing Components, Part 3, Construction

Baldwin has previously introduced you to a very useful program that displays information about any Java component, including inheritance, interfaces, properties, events, and methods. In this lesson, Baldwin explains the constructor for the GUI using JFrame, JPanel, JTextArea, JScrollPane, JTextField, JButton, and JLabel components.

Published: April 9, 2001
By Richard G. Baldwin

Java Programming, Lecture Notes # 1064

Preface

This series of lessons entitled *Swing from A to Z*, discusses the capabilities and features of Swing in quite a lot of detail. This series is intended for those persons who need to understand Swing at a detailed level.

This is the third lesson in a miniseries discussing the use of introspection for analyzing **Swing** components. The first lesson in this miniseries was entitled Swing from A to Z: Analyzing Swing Components, Part 1, Concepts. *(You will find links to all of the lessons in the miniseries at the following web site.)*

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different figures and listings while you are reading about them.

Recommended supplementary reading

In an earlier lesson entitled *Alignment Properties and BoxLayout, Part 1*, I recommended a list of Swing tutorials for you to study prior to embarking on a study of this series of lessons.

The lessons identified on that list will introduce you to the use of Swing while avoiding much of the detail included in this series.

Where are the lessons located?

You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes my lessons are difficult to locate there. You will find a consolidated index at *Baldwin's Java Programming Tutorials*.

The index on my site provides links to the lessons at Gamelan.com.

Preview

Lots of documentation is required

You need access to lots of documentation when programming in Java. The lessons in this miniseries discuss a very useful Java program that serves as a supplement to the Sun documentation.

A streamlined approach

In this miniseries, I will show you how to write a Java program that provides information about Swing and AWT components at the click of a button. The program displays:

- Inheritance family tree of the component
- Interfaces implemented by the component
- Properties of the component
- Events multicast by the component
- Public methods exposed by the component

Introspection

I provided quite a lot of background information on *introspection* in the previous lesson entitled Swing from A to Z: Analyzing Swing Components, Part 2, GUI Setup. Introspection can be used to extract information about any class that qualifies as a *JavaBeans Component*. This includes all of the **Swing** and **AWT** components. It also includes many of the other classes in the standard library as well.

Program output

Figure 1 is a screen shot showing the program output after you start the program and click the **OK** button.

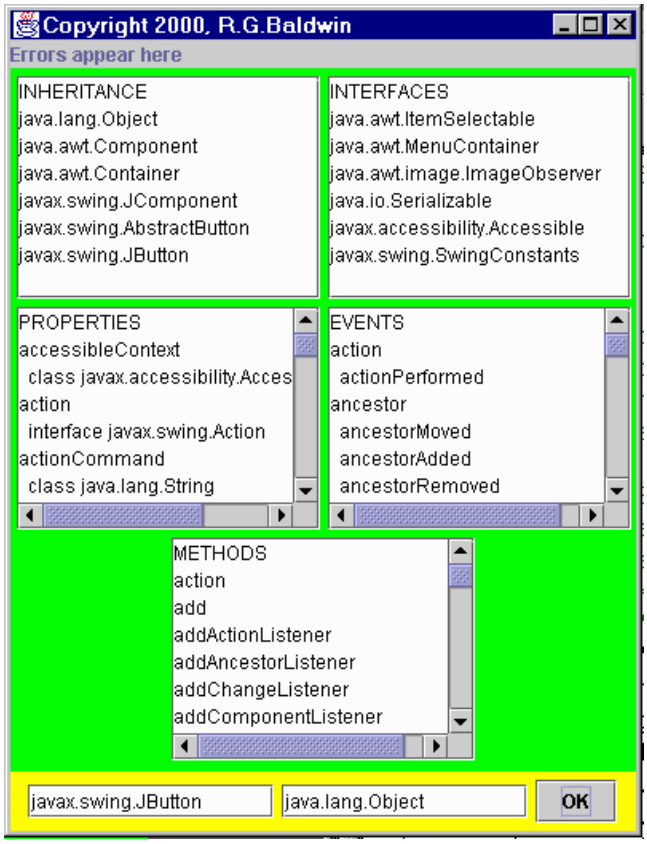

Figure 1. Screen shot showing program output.

The previous lesson explained the *target* class and the *ceiling* class, which are specified in the two text fields at the bottom of the GUI.

The program displays five kinds of information for the target class in scrollable panels.

Error messages are displayed in the gray panel at the top of the GUI.

Analysis of a JButton component

The screen shot of Figure 1 displays information about a **JButton** component, using all of the superclasses except the **Object** class.

In the previous lesson, I walked you through the code for this program, up to, but not including the constructor.

Along the way, I discussed how to combine and use objects of the **JTextArea** and **JScrollPane** classes. These are the white rectangular output panes in the GUI of Figure 1.

I will discuss the constructor in this lesson. Subsequent lessons will explain other aspects of the program.

Introduction

The Class class and the Introspector class

The previous lesson provided quite a lot of information regarding the use of the **Class** class and the **Introspector** class, and how they are used to achieve the objectives of this program.

The forName() and getBeanInfo() methods

This included a discussion of the **forName()** method of the **Class** class, and the **getBeanInfo()** method of the **Introspector** class, both of which are critical to the success of the program.

Superclasses and interfaces

That lesson explained that information about the superclass of a target class and the interfaces that it implements can be obtained through use of a **Class** object that represents the target class.

Properties, events, and methods

It also explained that an object of the interface type **BeanInfo** can be used to obtain other important information about the *properties*, *events*, and *methods* of a target class.

Sample Program

A complete listing of this program, named **Introspect03** is provided near the end of the lesson. It is provided here so that you can copy, compile, and begin using it even before you have an opportunity to learn exactly how it works.

Interesting Code Fragments

I will break this program down and discuss it in fragments.

The previous lesson discussed a large number of code fragments containing instance variables and the **main()** method. It explained the purpose of most of the instance variables as well as the behavior of the **main()** method.

This lesson will discuss the constructor for the controlling class, which is also the constructor for the GUI.

The controlling class

Just to refresh your memory, Listing 1 shows the declaration for the controlling class. The controlling class extends **JFrame**. Therefore, an object of the controlling class is a **JFrame** object, and is the GUI for the program.

```
public class Introspect03
                          extends
JFrame{

Listing 1
```

The constructor

Listing 2 shows the beginning of the constructor for the GUI.

```
  public Introspect03() {//constructor
    setDefaultCloseOperation(
                  JFrame.EXIT_ON_CLOSE);

Listing 2
```

The setDefaultCloseOperation() method

The boldface statement in Listing 2 shows a capability that was added in JDK 1.3. The method named **setDefaultCloseOperation** makes it possible to specify the operation that will be performed by default when the user initiates a "close" on the **JFrame** *(clicks the X-button in the upper right, selects Close from the menu produced by the control box in the upper left, or presses Alt+F4).*

As used in this program, the code in Listing 2 causes the program to terminate when the user initiates a "close" on the **JFrame**.

Parameters to setDefaultCloseOperation() method

If you use this method, you must specify one of the following choices:

- **DO_NOTHING_ON_CLOSE** (defined in the interface named WindowConstants, implemented by JFrame): Don't do anything; require the program to handle the operation in the windowClosing method of a registered WindowListener object.
- **HIDE_ON_CLOSE** (defined in the interface named WindowConstants): Automatically hide the frame after invoking any registered WindowListener objects.
- **DISPOSE_ON_CLOSE** (defined in the interface named WindowConstants): Automatically hide and dispose the frame after invoking any registered WindowListener objects.
- **EXIT_ON_CLOSE** (defined in JFrame): Exit the application using the System exit method. Use this only in applications.

defaultCloseOperation is a property

Note that this method is a property *setter* method. The value of the **defaultCloseOperation** property is set to **HIDE_ON_CLOSE** by default.

Using JDK 1.2

Since this capability was added in JDK 1.3, if you attempt to compile this program with an earlier version of the JDK and **Swing**, you will need to remove this statement. In addition, you will need to handle the termination operation in the **windowClosing** method of a registered **WindowListener** object.

Setting background colors

The code in Listing 3 sets the background color of the **JPanel** in the *Center* position of the **JFrame** to green, and sets the background color of the **JPanel** in the *South* position to yellow.

```
    outputPanel.setBackground(
                            Color.green);
    inputPanel.setBackground(
                           Color.yellow);

Listing 3
```

Placing the output panes

The code in Listing 4 places the five output **JScrollPane** objects in the **JPanel** object referred to by **outputPanel** in the *Center* of the **JFrame**.

```
    outputPanel.add(inherPane);
    outputPanel.add(intfcsPane);
    outputPanel.add(propsPane);
    outputPanel.add(eventsPane);
    outputPanel.add(methodsPane);

Listing 4
```

The default layout manager for a **JPanel** is **FlowLayout** with center alignment. The combination of the size of the panel and the sizes of the panes results in the layout shown in Figure 1. *(If you modify the sizes, you will see a different layout.)*

Could be larger

I purposely forced this GUI to be rather small to make it fit in this publication format. You may find it useful to increase the size of the **JFrame** and the size of each of the **JTextField** objects in order to produce larger viewable areas.

Set some default classes

The code in Listing 5 sets default values for the target class and ceiling class that appear when the GUI first appears on the screen. You may want to remove them entirely, or set them to

different default values.

```
    targetClass.setText(
                "javax.swing.JButton");
    ceilingClass.setText(
                    "java.lang.Object");

Listing 5
```

Placing the input components

The code in Listing 6 places the two input **JTextField** objects and the **JButton** object in the **JPanel** object in the *South* position in the GUI. Again, they are placed according to the default **FlowLayout** of the **JPanel** object.

```
    inputPanel.add(targetClass);
    inputPanel.add(ceilingClass);
    inputPanel.add(okButton);

Listing 6
```

Placing panels in the GUI

The code in Listing 7 places the three **JPanel** objects in the *North*, *Center*, and *South* positions of the **JFrame** object, according to the default layout manager, which is **BorderLayout**.

```
    getContentPane().add(
                        errors,"North");
    getContentPane().add(
                  outputPanel,"Center");
    getContentPane().add(
                    inputPanel,"South");

Listing 7
```

Setting GUI properties

The code in Listing 8 sets the values for several properties of the **JFrame** object.

```
    setResizable(false);
    setSize(400,520);
    setTitle(
        "Copyright 2000,
R.G.Baldwin");
    setVisible(true);
```

```
Listing 8
```

You might want to note in particular that passing **false** to the **setResizable()** method prevents the user from being able to change the size of the GUI.

An anonymous action listener

Note that we are still discussing code in the constructor for the GUI.

Listing 9 shows the beginning of an anonymous inner class that registers an anonymous action listener object on the **JButton** shown at the bottom of the GUI in Figure 1.

(If you are not familiar with the cryptic syntax of anonymous inner classes, you can find information on that topic in earlier lessons on my web site.)

```
    okButton.addActionListener(
      new ActionListener(){

Listing 9
```

Let me paraphrase the cryptic code in Listing 9 as follows:

This code instantiates an object from an unnamed class, which implements the **ActionListener** interface, and registers that object as a listener object on the **JButton** referred to by the reference variable named **okButton**.

The actionPerformed() method

Because this object implements the **ActionListener** interface, it must provide a concrete definition of the callback method named **actionPerformed()**. The code in Listing 10 begins the definition of that method.

```
        public void actionPerformed(
                          ActionEvent e){
          errors.setText(
                 "Errors appear here");
          inher.setText(
                      "INHERITANCE\n");
          intfcs.setText(
                       "INTERFACES\n");
          props.setText(
                       "PROPERTIES\n");
          events.setText(
                           "EVENTS\n");
          methods.setText("METHODS\n");
```

```
Listing 10
```

Initializing the output

The **actionPerformed()** method is invoked whenever the user clicks the **OK** button on the GUI.

As shown in Listing 10, the first thing that the method does is to reinitialize the text in the gray error panel and the five output panes. This is in preparation for getting and providing information in those panes.

Getting a Class object

As I mentioned in an earlier section, a **Class** object can be used to learn about the superclass of the class represented by the object, and the interfaces implemented by the class that the object represents.

Also, a **Class** object is required as the seed for a **BeanInfo** object from which other information about the target class can be obtained.

The code in Listing 11 begins by getting a **Class** object to represent the target class whose name has been entered into the input text field by the user.

```
          try{
            Class targetClassObject =
              Class.forName(
                targetClass.getText());

Listing 11
```

How do you get a Class object?

There are three ways to get a **Class** object that represents another class.

The getClass() method

One approach is to invoke the class method named **getClass()**, which is defined in the class named **Object**, passing a reference to an object as a parameter. This method returns a reference to a **Class** object that represents the class from which the object parameter was instantiated.

This is not the approach that I used in this program.

Using type.class

A **Class** object can represent primitive types as well as class types. The following syntax will get and save a reference to a **Class** object that represents the primitive **int** type:

```
Class targetClassObject = int.class;
```

This approach can also be used with class types as well. This is not the approach that I used in this program.

The forName() method

The third approach is to invoke the class method named **forName()** of the class named **Class**, passing the name of the target class as a **String** parameter. This method returns the **Class** object associated with the class or interface with the given string name. This is the approach that I used in this program.

The code in Listing 11 uses the **forName** method to get the **Class** object representing the target class and stores a reference to that object in the reference variable named **targetClassObject**.

Using the Class object

Once the **Class** object is available, it can be used to obtain information about the inheritance hierarchy and the interfaces implemented by the target class and its superclasses. That is accomplished in the two method calls shown in Listing 12.

```
            doInheritance();
            doInterfaces();

Listing 12
```

I will discuss these two methods in detail in a subsequent lesson.

Getting the BeanInfo object

The code in Listing 13 invokes the **getBeanInfo()** method of the **Introspector** class to get a **BeanInfo** object. It stores the reference to that object in the reference variable named **beanInfo**.

```
beanInfo = Introspector.getBeanInfo(
     targetClassObject,
     Class.forName(
               ceilingClass.getText()));

Listing 13
```

Two parameters are passed to the **getBeanInfo()** method in Listing 13. *(I colored the first one blue and the second one red to make them easier to separate visually.)*

The target class parameter

The first parameter is a reference to a **Class** object representing the target class. This object was instantiated earlier based on the **String** name of the target class extracted from the input text field at the lower left of the GUI in Figure 1.

The ceiling class parameter

The second parameter is a reference to a **Class** object representing the ceiling class. That **Class** object is instantiated by the red code in Listing 13. That code gets the **String** name from the input text field at the lower right in the GUI, and passes that name to the **forName()** method to get a **Class** object representing the ceiling class.

Using the BeanInfo object

The bean info object is used by the three methods invoked in Listing 14 to get information on properties, events, and methods, and to display that information in the output panes in the green area of the GUI in Figure 1.

```
            doProperties();
            doEvents();
            doMethods();

Listing 14
```

I will discuss those methods in detail in subsequent lessons.

Displaying errors

You may have noticed that much of the code discussed above is contained in a try-catch block.

The code in Listing 15 catches any exceptions thrown in this block of code and displays a minimal amount of information about the exception in the gray panel at the top of the GUI in figure 1.

```
          }catch(Exception ex){
                    errors.setText(
                       ex.toString());}

Listing 15
```

Tidying up

Finally, the code in Listing 16 contains the curly braces, parentheses, and semicolon required to signal the end of the **actionPerformed()** method, the **ActionListener** implementation, and the **addActionListener()** method. All of this is a part of the cryptic syntax involving the anonymous inner listener class.

In addition, Listing 16 contains the curly brace that signals the end of the constructor.

```
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener

  }//end constructor
```

Listing 16

Summary

In the lessons in this miniseries, I have introduced you to a very useful program that can be used to quickly obtain information about the following aspects of any Java class that qualifies as a JavaBeans component:

- Inheritance family tree of the component
- Interfaces implemented by the component
- Properties of the component
- Events multicast by the component
- Public methods exposed by the component

I have provided screen shots to show you how the program works in practice, and I have provided a complete listing of the program so that you can begin using it now.

I have provided quite a lot of background information on Java introspection, and have explained how introspection is used to achieve the objectives of this program.

I have walked you through the early portions of the code, explaining the purpose of a large number of instance variables and the behavior of the **main()** method.

Along the way, I discussed how to combine and use objects of the **JTextArea** and **JScrollPane** classes.

In this lesson, I walked you through the constructor and introduced you to the new JDK 1.3 capability provided by the method named **setDefaultCloseOperation()**.

I discussed the actual code that produced the layout and structure of the GUI shown in Figure 1. I also explained the use of an anonymous inner class that is used to instantiate an action listener object and register it on the OK button shown in Figure 1.

I explained how this listener object invokes the **forName()** and **getBeanInfo()** methods that produce the **Class** and **BeanInfo** objects required to achieve the objectives of this program.

I showed how the **actionPerformed()** method of the action listener object invokes the following methods to obtain and display the sought-after information about the target component:

- doInheritance()
- doInterfaces()
- doProperties()
- doEvents()
- doMethods()

What's Next?

In the next lesson, I will explain how the method named **doInheritance()** obtains and displays inheritance information about the target class in the upper left output pane in Figure 1.

In subsequent lessons, I will provide similar explanations for the other four methods in the above list.

Complete Program Listing

A complete listing of the program is provided in Listing 17.

```
/*File Introspect03.java
Copyright 2000, R.G.Baldwin

Produces a GUI that displays
inheritance, interfaces, properties,
events, and methods about components,
or about any class that is a bean.

Requires JDK 1.3 or later.  Otherwise,
must service the windowClosing event
to terminate the program.
Tested using JDK 1.3 under WinNT.
**************************************/
import java.io.*;
import java.beans.*;
import java.lang.reflect.*;
import java.util.*;
import java.awt.Color;
import java.awt.event.*;
import javax.swing.*;

public class Introspect03
                          extends JFrame{
  private JLabel errors =
      new JLabel("Errors appear here");
  private JPanel outputPanel =
                            new JPanel();
```

```
  private JPanel inputPanel =
                          new JPanel();
  private JTextField targetClass =
                    new JTextField(14);
  private JTextField ceilingClass =
                    new JTextField(14);
  private JButton okButton =
                     new JButton("OK");

  private JTextArea inher = new
       JTextArea("INHERITANCE\n",8,17);
  private JScrollPane inherPane =
                new JScrollPane(inher);
  private JTextArea intfcs = new
        JTextArea("INTERFACES\n",8,17);
  private JScrollPane intfcsPane =
               new JScrollPane(intfcs);
  private JTextArea props = new
        JTextArea("PROPERTIES\n",8,17);
  private JScrollPane propsPane =
                new JScrollPane(props);
  private JTextArea events =
        new JTextArea("EVENTS\n",8,17);
  private JScrollPane eventsPane =
               new JScrollPane(events);
  private JTextArea methods =
       new JTextArea("METHODS\n",8,17);
  private JScrollPane methodsPane =
              new JScrollPane(methods);

  private BeanInfo beanInfo;
  private Vector intfcsVector =
                          new Vector();

  public static void main(
                         String args[]){
    new Introspect03();
  }//end main

  public Introspect03() {//constructor
    //This require JDK 1.3 or later.
    // Otherwise service windowClosing
    // event to terminate the program.
    setDefaultCloseOperation(
                JFrame.EXIT_ON_CLOSE);

    outputPanel.setBackground(
                           Color.green);
    inputPanel.setBackground(
                          Color.yellow);

    outputPanel.add(inherPane);
    outputPanel.add(intfcsPane);
    outputPanel.add(propsPane);
    outputPanel.add(eventsPane);
    outputPanel.add(methodsPane);
```

```
    //Set some default values
    targetClass.setText(
                "javax.swing.JButton");
    ceilingClass.setText(
                  "java.lang.Object");

    inputPanel.add(targetClass);
    inputPanel.add(ceilingClass);
    inputPanel.add(okButton);

    getContentPane().add(
                      errors,"North");
    getContentPane().add(
                 outputPanel,"Center");
    getContentPane().add(
                  inputPanel,"South");
    setResizable(false);
    setSize(400,520);
    setTitle(
        "Copyright 2000, R.G.Baldwin");
    setVisible(true);

    //Anonymous inner class to provide
    // event handler for okButton
    okButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(
                        ActionEvent e){
          errors.setText(
                 "Errors appear here");
          inher.setText(
                     "INHERITANCE\n");
          intfcs.setText(
                     "INTERFACES\n");
          props.setText(
                     "PROPERTIES\n");
          events.setText(
                        "EVENTS\n");
          methods.setText("METHODS\n");
          try{
            Class targetClassObject =
              Class.forName(
                targetClass.getText());
            doInheritance();
            doInterfaces();
            beanInfo = Introspector.
                   getBeanInfo(
                     targetClassObject,
                       Class.forName(
                         ceilingClass.
                           getText()));
            doProperties();
            doEvents();
            doMethods();
          }catch(Exception ex){
```

```
                    errors.setText(
                       ex.toString());}
      }//end actionPerformed
     }//end ActionListener
   );//end addActionListener

}//end constructor


void doInheritance()
       throws ClassNotFoundException{
  //Get and display inheritance
  // hierarchy
  Vector inherVector = new Vector();
  String theClass = targetClass.
                            getText();
  Class theClassObj = null;
  Class theSuperClass = null;
  while(!(theClass.equals(
               "java.lang.Object"))){
    inherVector.add(theClass);
    theClassObj = Class.forName(
                            theClass);
    theSuperClass = theClassObj.
                     getSuperclass();

    //Get and save interfaces to be
    // used later
    if(theClassObj.getInterfaces()
                               != null){
      intfcsVector.add(theClassObj.
                     getInterfaces());
    }//end if



    theClass = theSuperClass.
              getName();
  }//end while loop
  inherVector.add(
                 "java.lang.Object");

  //Display vector contents in
  // reverse order
  for(int i = 0;
        i < inherVector.size();i++){
    inher.append(
     ((String)inherVector.elementAt(
       inherVector.size() - (i+1))));
    inher.append("\n");
  }//end for loop
}//end doInheritance


void doInterfaces(){
  Vector interfaceNameVector =
```

```
                          new Vector();
    //Interface information was stored
    // in intfcsVector earlier.
    for(int i = 0;

          i < intfcsVector.size();i++){
      Class[] interfaceSet =
                 (Class[])intfcsVector.
                          elementAt(i);
      for(int j = 0;
          j < interfaceSet.length;j++){
        interfaceNameVector.add(
            interfaceSet[j].getName());

      }//end for loop on j
    }//end for loop on i

    Object[] interfaceNameArray =
         interfaceNameVector.toArray();
    Arrays.sort(interfaceNameArray);

    if(interfaceNameArray.length > 0){
      intfcs.append(
                 interfaceNameArray[0].
                           toString());
      intfcs.append("\n");
    }//end if

    for(int i = 1;
         i < interfaceNameArray.length;
                                    i++){
      //Eliminate dup interface names
      if(!(interfaceNameArray[i].
          equals(
            interfaceNameArray[i-1]))){
        intfcs.append(
                 interfaceNameArray[i].
                           toString());
        intfcs.append("\n");
      }//end if
    }//end for loop
  }//end doInterfaces


  void doProperties(){
    Vector propVector = new Vector();
    PropertyDescriptor[] propDescrip =
            beanInfo.
              getPropertyDescriptors();
    for (int i = 0;
         i < propDescrip.length; i++) {
      PropClass propObj =
                       new PropClass();
      propObj.setName(propDescrip[i].
                             getName());
      propObj.setType("" +
```

```
                  propDescrip[i].
                    getPropertyType());
    propVector.add(propObj);
   }//end for-loop

   Object[] propArray = propVector.
                              toArray();
   Arrays.sort(
           propArray,new PropClass());
   for(int i = 0;
            i < propArray.length;i++){
     props.append(propArray[i].
                           toString());
     props.append("\n");
   }//end for loop
 }//end doProperties


 void doEvents(){
   Vector eventVector = new Vector();
   EventSetDescriptor[] evSetDescrip =
           beanInfo.
             getEventSetDescriptors();
   for (int i = 0;
        i < evSetDescrip.length; i++){
     EventClass eventObj =
                    new EventClass();
     eventObj.setName(evSetDescrip[i].
                            getName());
     MethodDescriptor[] methDescrip =
       evSetDescrip[i].
       getListenerMethodDescriptors();
     for (int j = 0;
        j < methDescrip.length; j++) {
       eventObj.setListenerMethod(
            methDescrip[j].getName());
     }//end for-loop
     eventVector.add(eventObj);
   }//end for-loop

   Object[] eventArray = eventVector.
                              toArray();
   Arrays.sort(
         eventArray,new EventClass());
   for(int i = 0;
           i < eventArray.length;i++){
     events.append(eventArray[i].
                           toString());
     events.append("\n");
   }//end for loop
 }//end doEvents


 void doMethods(){
   Vector methVector = new Vector();
   MethodDescriptor[] methDescrip =
```

```
       beanInfo.getMethodDescriptors();
    for (int i = 0;
         i < methDescrip.length; i++) {
        methVector.add(
             methDescrip[i].getName());
    }//end for-loop

    Object[] methodArray =
                  methVector.toArray();
    Arrays.sort(methodArray);

    if(methodArray.length > 0){
      methods.append(
            methodArray[0].toString());
      methods.append("\n");
    }//end if

    for(int i = 1;
           i < methodArray.length;i++){
      //Eliminate dup method names
      if(!(methodArray[i].equals(
                   methodArray[i-1]))){
        methods.append(
            methodArray[i].toString());
        methods.append("\n");
      }//end if
    }//end for loop
  }//end doMethods
//=================================//

//This inner class is used to
// encapsulate name and type
// information about properties.  It
// also serves as a class from which a
// Comparator object can be
// instantiated to assist in sorting
// by name.
class PropClass implements Comparator{
  private String name;
  private String type;

  public void setName(String name){
    this.name = name;
  }//end setName

  public String getName(){
    return name;
  }//end getName

  public void setType(String type){
    this.type = type;
  }//end setType

  public String toString(){
    return(name + "\n  " + type);
  }//end toString
```

```
  public int compare(
                 Object o1, Object o2){
    return ((PropClass)o1).getName().
           toUpperCase().compareTo(
             ((PropClass)o2).getName().
                        toUpperCase());
  }//end compare

  public boolean equals(Object obj){
    return this.getName().equals(
           ((PropClass)obj).getName());
  }//end equals
}//end class PropClass
//===================================//

//This inner class is used to
// encapsulate name and handler
// information about events.  It also
// serves as a class from which a
// Comparator object can be
// instantiated to assist in sorting
// by name.
class EventClass implements Comparator{
  private String name;
  private Vector lstnrMethods =
                          new Vector();

  public void setName(String name){
    this.name = name;
  }//end setName

  public String getName(){
    return name;
  }//end getName

  public void setListenerMethod(
                   String lstnrMethod){
    lstnrMethods.add(lstnrMethod);
  }//end setType

  public String toString(){
    String theString = name;

    for(int i = 0;
          i < lstnrMethods.size();i++){
      theString = theString + "\n  " +
             lstnrMethods.elementAt(i);
    }//end for loop

    return theString;
  }//end toString

  public int compare(
                 Object o1, Object o2){
    return ((EventClass)o1).getName().
```

```
          toUpperCase().compareTo(
            ((EventClass)o2).getName().
                        toUpperCase());
  }//end compare

  public boolean equals(Object obj){
    return this.getName().equals(
          ((EventClass)obj).getName());
  }//end equals
}//end EventClass inner class

}//end controlling class Introspect03
```

Listing 17

About the author

Richard Baldwin *is a college professor and private consultant whose primary focus is a combination of Java and XML. In addition to the many platform-independent benefits of Java applications, he believes that a combination of Java and XML will become the primary driving force in the delivery of structured information on the Web.*

Richard has participated in numerous consulting projects involving Java, XML, or a combination of the two. He frequently provides onsite Java and/or XML training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Java Programming Tutorials, which has gained a worldwide following among experienced and aspiring Java programmers. He has also published articles on Java Programming in Java Pro magazine.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

baldwin.richard@iname.com

-end-