Swing from A to Z: Analyzing Swing Components, Part 2, GUI Setup

In the previous lesson, Baldwin 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 how the GUI for this program is set up using JFrame, JPanel, JTextArea, JScrollPane, JTextField, JButton, and JLabel components.

Published February 19, 2001
By Richard G. Baldwin

Java Programming, Lecture Notes # 1062

- Preface
- Preview
- Introduction
- Sample Program
- Interesting Code Fragments
- Summary
- What's Next
- Complete Program Listing

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 second 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 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

Documentation is required

You will need access to lots of documentation when programming in Java. The standard Sun documentation contains a voluminous amount of information and is very useful. This lesson discusses a Java program that is also very useful from a documentation viewpoint. The program described in this lesson is intended to be used as a supplement to the Sun documentation.

A streamlined approach

Sometimes you need something a little more streamlined than the large Sun documentation package. In this miniseries, I will show you how to write a Java program that provides almost instantaneous 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

Customization is possible

After studying the lessons in this miniseries, you should be able to customize the program to provide more or less information, in the same or different formats.

Introspection

Java provides a capability, known as introspection, which can be used to extract information about a class that qualifies as a *JavaBean Component*. Fortunately, this includes all of the **Swing** components and all of the **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.

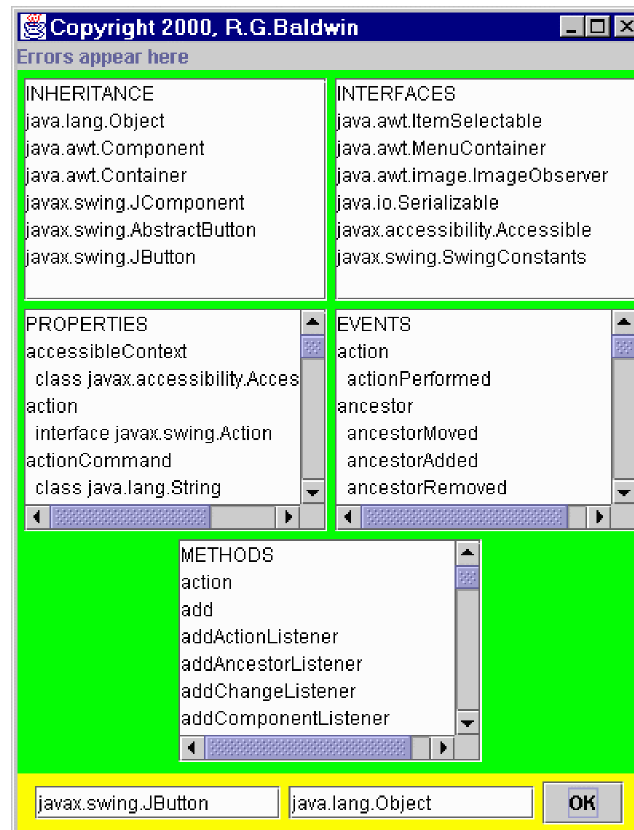

Figure 1. Screen shot showing program output.

You specify the target component to be analyzed by entering its package and class name in the text field at the lower left. The program displays five kinds of information for the class specified in that text field in scrollable panels.

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

The ceiling class

A ceiling class is entered in the text field at the bottom right. The program uses the ceiling class to determine how much of the inheritance hierarchy to consider when determining properties, events, and methods for the target class.

Analysis of a JButton component

The screen shot of Figure 1 displays information about a **JButton** component, using all of the superclasses up to, but not including the **Object** class.

In this lesson, I will provide quite a lot of background information on introspection, and will walk you through initial code up to, but not including the constructor for the program.

Along the way, I will discuss how to combine and use objects of the **JTextArea** and **JScrollPane** classes. These are the rectangular white panes showing in the green area of the GUI in Figure 1.

I will discuss the constructor in the next lesson. Subsequent lessons will explain other material in the program.

Introduction

The Class class

JDK 1.3 includes the following class:

java.lang.Class

An object of the class whose name is **Class** represents another class or interface. The object can be used to learn many things about the class that the object represents.

Getting superclass and interfaces

Among other things, the object can be used to determine the superclass of the class that the object represents, as well as the interfaces implemented by the class that the object represents. Those two capabilities will be used in developing the program discussed in this lesson.

The Introspector class

JDK 1.3 also includes the following class:

java.beans.Introspector

The **Introspector** class, intended primarily for use with JavaBean Components, provides a way for tools to learn about the *properties*, *events*, and *methods* of a target bean's class. Of course, this includes those properties, events, and methods that the target class inherits from its superclasses.

As mentioned earlier, all Swing and AWT components are beans, making them suitable for analysis by introspection.

getBeanInfo

The **Introspector** class provides three overloaded versions of a method named **getBeanInfo**. Each of these methods accepts a **Class** object representing the target bean as an input parameter. *(Two of the versions accept other incoming parameters as well.)*

This method can be used to analyze the target bean's class and superclasses looking for information about properties, events, and methods. *(This process is discussed in detail in my tutorial lessons on JavaBean Components. You will find a consolidated index to those lessons at my web site.)*

The BeanInfo object

The information discovered through introspection is used to build and return an object of the type **BeanInfo**. The **BeanInfo** object contains even more information about the target bean than is provided by the **Class** object that forms its seed.

The **BeanInfo** interface declares a variety of methods that can be used to extract specific information about the bean from that object.

Performing the analysis

The **getBeanInfo()** method is the primary method of the **Introspector** class used to analyze a bean. The program developed in this lesson will use one of the three overloaded versions of this method. Simply put, this method takes a **Class** object (representing a target class) as a parameter and returns a **BeanInfo** object containing information about the target class.

The ceiling class

The overloaded version of **getBeanInfo** used in this lesson accepts a second **Class** object as a parameter. It uses the class represented by that object as a ceiling for introspection up the inheritance hierarchy. Information is returned taking into account all the superclasses up to, but not including the ceiling class.

For example, if this second class is the direct superclass of the primary target class, only information about the primary target class is returned.

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 how it all works.

Interesting Code Fragments

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

The controlling class

Listing 1 shows the declaration for the controlling class. As you can see, the 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
```

Major GUI panels

Listing 2 begins the declaration of a long series of instance variables of the controlling class. This listing shows the major panels that will be contained in the North, Center, and South positions of the **JFrame** GUI object.

```
  private JLabel errors =
      new JLabel("Errors appear
here");

  private JPanel outputPanel =
                          new
JPanel();
  private JPanel inputPanel =
                          new
JPanel();

Listing 2
```

errors

The **JLabel** object referred to by **errors** in listing 2 will be used to display any error messages produced by the program. This output appears near the top of the GUI in Figure 1.

outputPanel

The **JPanel** object referred to by **outputPanel** will be used as a container for five **JScrollPane** objects used to display the desired information about the target component. This panel appears in the center of the GUI in Figure 1.

inputPanel

The **JPanel** object referred to by **inputPanel** will be used as a container for two **JTextField** objects and one **JButton** object, appearing at the bottom of the GUI. These three components are used for user input to the program.

The input components

Listing 3 shows the three input components that will be contained by the **JPanel** object at the bottom of the GUI.

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

Listing 3

targetClass

The **JTextField** object referred to by **targetClass** is used to collect user input. This is where the user enters the package and class name for the component of interest. *Note that the user must enter a fully-qualified package and class name unless the class file is located in the directory from which the program is being executed.*

ceilingClass

The **JTextField** object referred to by **ceilingClass** is also used to collect user input. This is where the user enters the package and class name for an analysis ceiling. The program will take into account all superclasses up to, but not including the ceiling class when reporting properties, events, and methods. In other words, properties, events, and methods inherited from and above the ceiling class will not be reflected in the program output.

okButton

As you have probably already guessed, after entering the requisite package and class names in the two text fields, the user clicks the **JButton** referred to by **okButton** to cause the program to compute and display a new set of results.

The output components

Listing 4 shows the code that instantiates the first of the five output components that appear in the green area of the GUI in Figure 1. This object appears in the upper-left corner of the green

area.

```
  private JTextArea inher = new
       JTextArea("INHERITANCE\n",8,17);
  private JScrollPane inherPane =
                 new JScrollPane(inher);

Listing 4
```

This component consists of a combination of a **JTextArea** object and a **JScrollPane** object.

JTextArea

A **JTextArea** object is a multi-line area that displays plain text. Unlike **java.awt.TextArea**, a **JTextArea** component doesn't manage scrolling. *(Thus, an object of this class is not a direct functional replacement for an AWT TextArea object.)*

If you need scrolling capability for a **JTextArea** object, *which we do in this program*, you need to place it inside of a **JScrollPane** object.

The **JTextArea** class has several constructors. The constructor used in Listing 4 causes the object to display an initial text value of **INHERITANCE** followed by a newline.

Size of JTextArea objects

The same version of the constructor was used for all five of the **JTextArea** objects in the green area of Figure 1. The parameters passed to the constructors caused the preferred size of each object to accommodate 8 rows of text and 17 columns of text.

If you examine the GUI in Figure 1, you will see that each **JTextArea** object does accommodate 8 rows. However, since the widths of the characters differ in the display font used, the specification for the number of columns did not match the number of characters displayed horizontally.

(You may also notice that when a horizontal scroll bar appears, it consumes one of the available rows.)

JScrollPane

A **JScrollPane** object provides a scrollable view of a component. *(In this case, it provides a scrollable view of a JTextArea component, but JScrollPane can be used with many different kinds of objects.)*

A **JScrollPane** object manages a viewport, optional vertical and horizontal scroll bars, and optional row and column heading viewports. *(In this case, I didn't make use of the latter option.)*

JScrollPane Constructors

JScrollPane has several constructors. I elected to use a constructor that is defined as shown in Listing 5:

```
JScrollPane(Component view)
 Creates a JScrollPane that displays
 the contents of the specified
 component, where both horizontal and
 vertical scrollbars appear whenever
 the component's contents are larger
 than the view.

Listing 5
```

In other words, I used a version of the constructor that accepts a reference to my **JTextArea** object, and manages a viewport on that object with scroll bars appearing when needed.

As you can see in Listing 4 above, I passed a reference to the **JTextField** object (**inher**) to the constructor for the new **JScrollPane** object.

Additional output panes

Listing 6 shows the code that instantiates the remaining four output panes. This code is essentially the same as that discussed above, and won't be discussed further here.

```
  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);

Listing 6
```

Remaining instance variables

Listing 7 shows the two remaining instance variables declared in the definition of the controlling class. I will defer a discussion of the purpose of these two instance variables until later when I discuss the code that uses them.

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

Listing 7
```

The main() method

Listing 8 shows the definition of the **main** method. As you can see, the sole purpose of the **main** method is to instantiate an object of the controlling class to serve as the GUI for the program.

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

Listing 8
```

Summary

In this and the previous lesson, 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 JavaBean 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 provided screen shots to show you how the program works in practice, and I provided a complete listing of the program so that you can begin using it.

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.

What's Next?

Several lessons will be required to provide a complete discussion of this program. In the next lesson, I will explain the constructor for the GUI used in this program. This will include examples of the use of the **forName()** method of the **Class** class, and the **getBeanInfo()** method of the **Introspector** class.

Subsequent lessons will contain detailed discussions on how the objects produced by these methods are used to achieve the objectives of the program.

Complete Program Listing

A complete listing of the program is provided in Listing 9.

```
/*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 9

Copyright 2000, Richard G. Baldwin. Reproduction in whole or in part in any form or medium without express written permission from Richard Baldwin is prohibited.

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-