Swing from A to Z: Analyzing Swing Components, Part 5, Interfaces

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 code that gets, sorts, and displays information about the interfaces implemented by a component.

Published: April 23, 2001
By Richard G. Baldwin

Java Programming, Lecture Notes # 1068

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 fifth 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

Streamlined Documentation

The lessons in this miniseries discuss a very useful Java program that serves as a supplement to the Sun documentation.

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

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.

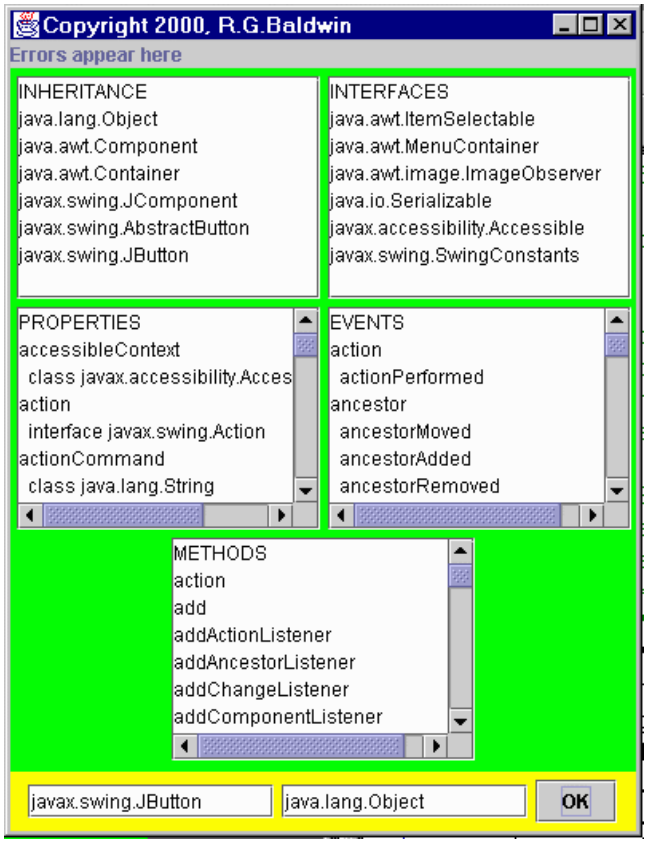

Figure 1. Screen shot showing program output.

The various parts of this GUI have been explained in the earlier lessons in this miniseries.

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.

The program code

In the previous lessons, I have walked you through the code for this program, up to, and including the code that gets and displays the inheritance information in the upper-left pane.

In this lesson, I will discuss the code that gets and displays the sorted interface information shown in the top right-hand pane of Figure 1.

Along the way, I will show you how to use the sorting capability of the Java class libraries.

Subsequent lessons will explain other aspects of the program.

Introduction

The Class class and the Introspector class

Previous lessons explained the use of the **Class** class and the **Introspector** class to achieve the objectives of this program.

This included a discussion of the **forName()** and **getBeanInfo()** methods.

Superclasses and interfaces

It was 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.

This lesson will use that fact to develop the interface information for the target class.

Properties, events, and methods

Previous lessons 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, entitled Swing from A to Z: Analyzing Swing Components, Part 4, Inheritance discussed the method named **doInheritance()**. It explained how the program uses the **getSuperclass** method of the **Class** class in a while loop to get and display the family tree for the target class. That method is invoked in the **actionPerformed()** event handler for the OK button in Figure 1.

The actionPerformed() method

The code in Listing 1 is an abbreviated version of the **actionPerformed()** method. Most of the code has been deleted because it is not germane to this lesson.

```
        public void actionPerformed(
                          ActionEvent
e){
          //...
            doInheritance();
```

```
            doInterfaces();

Listing 1
```

The doInterfaces() method

The method named **doInterfaces()** invoked by the **actionPerformed()** method is the primary topic of this lesson.

The purpose of the **doInterfaces()** method is to get, sort, and display information about the interfaces implemented by the target class and its superclasses.

Inheriting an interface implementation

Recall that the implementation of an interface is an inherited trait. If the superclass of a target class implements an interface, then, among other things, a reference to an object of the target class can be treated as the interface type.

Methodology

In addition to getting and displaying inheritance information, the method named **doInheritance()** *(discussed in the previous lesson)* gets and saves information about the interfaces implemented by each class in the family tree of the target class. The information saved by the **doInheritance()** method provides the starting point for the code in the **doInterfaces()** method.

The interface information

The interface information is stored in a **Vector** object referred to by an instance variable named **intfcsVector**. Each element in the vector is a reference to an array object containing references to **Class** objects. Each **Class** object referred to by the elements in the array object represents one of the interfaces implemented by one of the classes in the family tree.

Getting interface names into a sorted array

The code in the **doInterfaces()** method extracts the references to each of those class objects, gets the name of the interface represented by each **Class** object as a **String**, and stores references to those strings in a new **Vector** object.

Then it converts the contents of that **Vector** into an array of type **Object** where each element in the array contains a reference to a **String** containing the name of an interface.

Following this, a simple version of the class method named **sort()** of the **Arrays** class is used to sort the contents of the array.

Eliminate duplicates and display

After the contents of the array are sorted, a **for** loop is used to eliminate any duplicate interface names in the array, and the strings referred to by the elements in the array are appended to the contents of the **JTextArea** displayed in the upper-right pane of Figure 1.

Saving the interface information

Listing 2 shows the code in the **doInheritance()** method that gets and saves the interface information in the **Vector** referred to by the instance variable named **intfcsVector**. As mentioned above, this information provides the starting point for the method named **doInterfaces()**.

(The remaining code in the doInheritance() method was deleted from this display because it is not germane to this lesson.)

```
  void doInheritance()
          throws ClassNotFoundException{
    //...
      if(theClassObj.getInterfaces()
                                != null){
        intfcsVector.add(theClassObj.
                         getInterfaces());
      }//end if
    //...

Listing 2
```

The doInterfaces() method

Listing 3 shows the beginning of the method named **doInterfaces()**. As mentioned above, the behavior of this method is the primary topic for this lesson.

```
  void doInterfaces(){
    Vector interfaceNameVector =
                             new
Vector();

Listing 3
```

The code in Listing 3 above declares a local variable named **interfacesNameVector** that will be used to store the names of all the interfaces implemented by the target class and its superclasses.

Getting interface names

The first task of this method is to get the names of each interface implemented by each class in the family tree as a simple list of strings. This is accomplished by the nested **for** loops in Listing

4.

```
    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

Listing 4
```

The data format

The code in Listing 4 is pretty straightforward once you understand how the data is stored in the **Vector** referred to by **intfcsVector**.

Each element in that vector is a reference to an array containing references to objects of the class **Class**. Each of those **Class** objects represents an interface implemented by one of the classes in the family tree.

The outer for loop

During each iteration, the outer **for** loop

- Extracts an element containing a reference to one of those arrays
- Saves the reference in the local variable named **interfaceSet**

The inner for loop

During each iteration, the inner **for** loop

- Extracts an element from the array referred to by **interfaceSet**
- Gets the name of the interface represented by the **Class** object referred to by that element *(as a String)*
- Stores a reference to that **String** in the **Vector** referred to by **interfaceNameVector**

Sorting the interface data

The next task is to sort the strings containing the names of the interfaces into alphabetical order. This is accomplished by the code in Listing 5.

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

Listing 5
```

Convert from Vector to array

The code in Listing 5 above begins by invoking the **toArray()** method of the **Vector** class. This method returns an array of type **Object** containing all of the elements in the **Vector** in the correct order.

Sort the array

The **Arrays** class provides about eighteen overloaded versions of class methods named **sort()** that can be used to sort the contents of an array object.

The code in Listing 5 above uses one of the simpler versions of the **sort()** method to sort the contents of the array into ascending order. When the **sort()** method returns, the array object referred to by **intrfaceNameArray** contains the names of all of the interfaces implemented by all of the classes in the family tree sorted in ascending order.

Eliminate duplicate names and display

It is possible that two or more of the classes in the family tree may implement the same interface, resulting in duplicate interface names in the strings referred to by the elements in the array referred to by **interFaceNameArray**.

The code in Listing 6 appends the names of each interface to the contents of the **JTextArea** referred to by **intfcs**, eliminating any duplicate names in the process.

```
    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

Listing 6
```

The resulting contents of the **JTextArea** are displayed in the upper-right pane in Figure 1.

Summary

In this and the previous lessons, I have introduced you to a program that displays 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 background information on Java introspection.

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

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

In this lesson, I explained the **doInterfaces()** method in detail. In that discussion, I showed you how to use a simple form of the **sort()** method of the **Arrays** class to sort the interface names into ascending alphabetic order.

I showed you how to eliminate duplicate interface names and how to display the sorted information in a **JTextArea** object. That information appears in the upper-right pane of Figure 1.

What's Next?

In the next lesson, I will explain how the method named **doProperties()** obtains and displays information about the properties of the target class, in alphabetical order, in the left-center output pane in Figure 1.

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

Complete Program Listing

A complete listing of the program is provided in Listing 7.

```
/*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 7
```

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-