Swing from A to Z: Analyzing Swing Components, Part 1, Concepts

Baldwin introduces a very useful program that displays information about any Java component, including inheritance, interfaces, properties, events, and methods. You can expand the program to provide even more information if you wish to do so.

Published January 15, 2001
By Richard G. Baldwin

Java Programming, Lecture Notes # 1060

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.

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 the 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.

Introduction

Small core, large library

Java consists of a relatively small core language and a very large class library. Therefore, becoming a successful Java programmer depends as much on learning how to effectively use the class library as on learning the language.

Documentation is required

When programming in Java, unless you have a very good memory, you will need access to lots of documentation. The standard Sun documentation produced using the javadoc program is very well structured and contains a voluminous amount of information. I keep an icon linked to that documentation on my desktop so that I can view it with a simple double-click on the icon. I couldn't succeed as a Java programmer without it.

The program described in this lesson is intended to be used as a supplement to, and not a replacement for the Sun documentation.

A streamlined approach

Sometimes you need something a little more streamlined than the large Sun documentation package. In this and the next few lessons, I will show you how to write a Java program that will provide almost instantaneous information about Swing and AWT components at the click of a button. The version of the program that I will show you will provide the following information:

- 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

In addition, after studying these lessons, 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 applied to extract information about a class from the class libraries. This capability can be used to write programs that tend to make the class libraries self documenting.

JavaBean Components

Introspection is designed to be used with JavaBean Components, which, fortunately, 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.

The objective here is not to learn how to write beans *(I discuss that task in other lessons that I have written)*. Rather it is to learn how to use introspection for a somewhat different purpose.

For the purposes of this lesson, suffice it to say that in order to qualify as a JavaBean Component, a class must implement the serializable interface. Unless the class is declared **final**, it should also provide a *noarg* constructor.

Starting the program

When you first start the program, the screen shot shown in Figure 1 will appear on the screen.

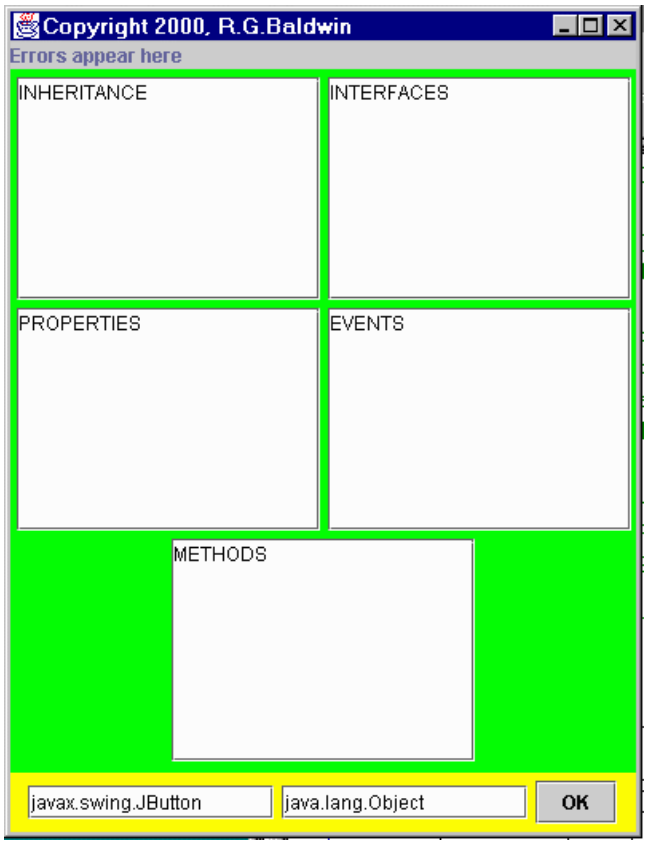

Figure 1. Screen shot upon startup.

Purposely small size

Note that I kept this GUI small so that the screen shots will fit in this narrow publication format without the need for reduction. You may want to increase the size dimensions of the **JFrame**

and the components contained in the **JFrame** to make it possible to view more information without the need to scroll.

Error message output

What you see in Figure 1 is a simple Swing GUI with six display panels. One of them is a gray output panel shown at the top. Error messages are displayed in this panel.

Component information output

Below the error panel are five rectangular white panels where the five kinds of information described in the above list are displayed. These are **JTextArea** objects in **JScrollPane** objects

The input panel

At the bottom of the GUI are two **JTextField** objects and a **JButton** object.

To use the program, you enter the target class for a component of interest in the left-hand text field.

You enter some superclass of that class in the right-hand text field. This superclass acts as a ceiling and restricts the output information for properties, events, and methods to include the target class, plus all classes up to but not including the ceiling class.

Analyze the target component

Then you click the button labeled OK in the bottom right-hand corner.

Figure 2 shows the output of the program for the target component (**JButton**) and ceiling superclass (**Object**) shown in the textfields at the bottom of Figure 1.

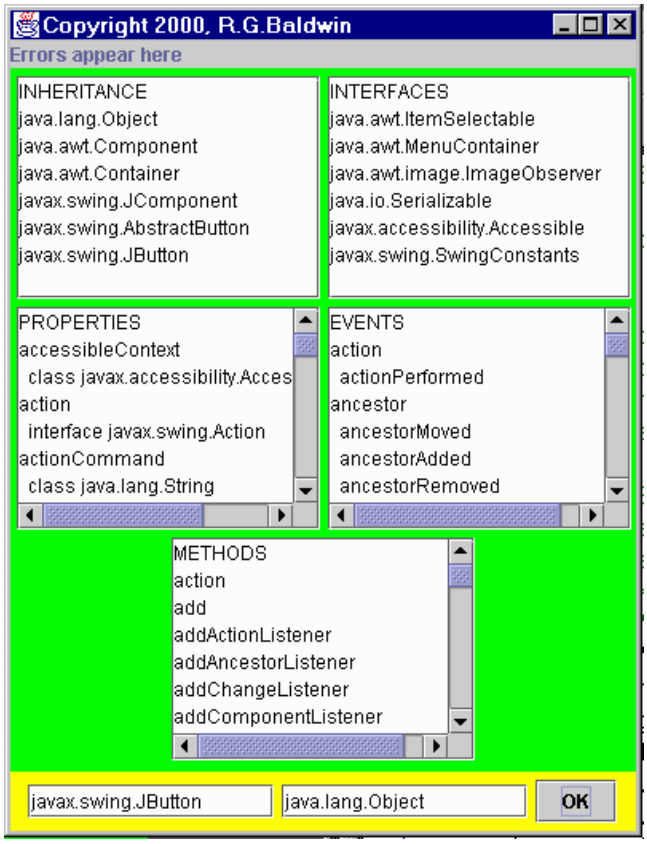

Figure 2. Screen shot after clicking the OK button.

Inheritance

The upper left-hand white panel in Figure 2 shows the inheritance family tree for the **JButton** class, in order from **Object** at the top to **JButton** at the bottom. As you can see, there are six classes in this family tree. They all fit in the allotted space with no scrolling required.

Once you have this display, you can copy class and package names from this panel and paste them into the textfields to produce more restrictive results.

Interfaces

The upper right-hand white panel in Figure 2 shows that the **JButton** class implements the six interfaces listed there. These interfaces are listed in ascending alphabetical order from top to bottom. *(The JButton class implements only the Accessible interface. The other five implementations are inherited.)*

The serializable interface

Of particular importance in this list is **java.io.serializable**. The *introspection* methodology used in this program is designed to work for any class that meets the requirements of being a JavaBean Component. As mentioned earlier, one of those requirements is implementation of the **serializable** interface.

All components are beans

All Swing components and all AWT components are JavaBean Components. In addition, many other non-visual classes in the standard class library meet the requirements for being JavaBean Components, and therefore can be analyzed using this program. The program can also be used to analyze new classes that you define, provided that you cause your new classes to implement the serializable interface.

Properties

All of the white panels in the GUI produced by this program automatically become scrolling panels when needed. As you can see, there are scroll bars on the bottom and right side of all of the panels except for the two at the top.

The **JButton** class has many properties. *(For a discussion of properties, see my lessons on JavaBean Components. You will find an index to those lessons at my web site.)*

The PROPERTIES panel at the middle left in Figure 2 shows the names and types of all the properties of **JButton**. Only a portion of the information can be seen in the screen shot. The remaining information can be viewed by scrolling the panel.

The property names are listed in ascending alphabetic order from top to bottom.

Events

The **JButton** component multicasts about fifteen different event types, such as action events, ancestor events, mouse events, key events, etc. Some of these event types, such as the action event, have only one callback method. Others, such as the ancestor event type have several callback methods.

The EVENTS panel at the middle right in Figure 2 shows the names of each event type multicast by **JButton**, along with the names of the callback methods for each of the events.

The event types are listed in ascending alphabetic order from top to bottom.

Methods

JButton exposes many public methods. The METHODS panel at the bottom of Figure 2 lists all of the public methods exposed by **JButton**. The methods are listed in ascending alphabetic order from top to bottom.

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 works.

Summary

In this 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 also provided some brief information about what makes the program work as it does.

What's Next?

Several lessons will be required to provide a complete discussion of this program. In the next lesson, I will explain the concepts behind this program in much more detail. I will also begin the discussion of the code behind the concepts.

Complete Program Listing

A complete listing of the program is provided in Listing 1.

```
/*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 1

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-