Swing from A to Z: Analyzing Swing Components, Part 4, Inheritance

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 and displays inheritance information about a component.

Published: April 16, 2001
By Richard G. Baldwin

Java Programming, Lecture Notes # 1066

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 fourth 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:

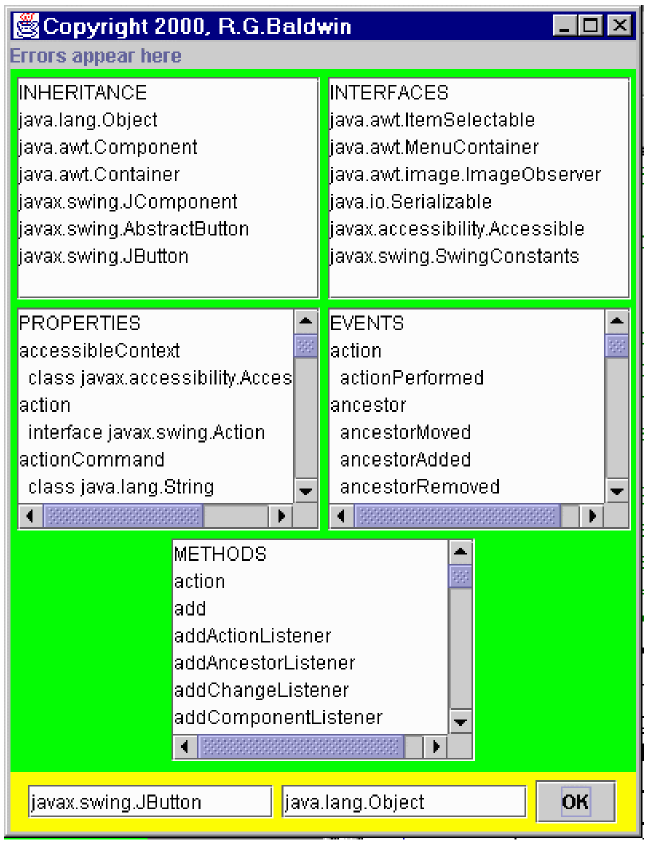
- Inheritance family tree of the component
- Interfaces implemented by the component
- Properties of the component
- Events multicast by the component
- Public methods exposed by the component

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.


```
Copyright 2000, R.G.Baldwin
Errors appear here

INHERITANCE                          INTERFACES
java.lang.Object                     java.awt.ItemSelectable
java.awt.Component                   java.awt.MenuContainer
java.awt.Container                   java.awt.image.ImageObserver
javax.swing.JComponent               java.io.Serializable
javax.swing.AbstractButton           javax.accessibility.Accessible
javax.swing.JButton                  javax.swing.SwingConstants

PROPERTIES                           EVENTS
accessibleContext                    action
 class javax.accessibility.Acces      actionPerformed
action                               ancestor
 interface javax.swing.Action         ancestorMoved
actionCommand                         ancestorAdded
 class java.lang.String               ancestorRemoved

METHODS
action
add
addActionListener
addAncestorListener
addChangeListener
addComponentListener

javax.swing.JButton    java.lang.Object    OK
```

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.

In the previous lessons, I have walked you through the code for this program, up to, and including the constructor. This included the instantiation of **Class** and **BeanInfo** objects from which information about the target class can be obtained.

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

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 inheritance family tree 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 3, Construction discussed the constructor for this program. It explained the purpose of most of the code in the constructor, including an anonymous inner class that provides action event handling capability for the OK button in the lower right-hand corner of Figure 1.

The actionPerformed() method

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

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

Listing 1
```

The doInheritance() method

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

The purpose of the **doInheritance()** method is to obtain and display the inheritance family tree beginning with the target class, and progressing up the inheritance hierarchy to the class named **Object**.

Methodology

The approach used is to get and save the superclass of the target class. Then use that superclass as a new target class and repeat the process until the class named **Object** is encountered.

The objective is to display the family tree beginning with the class named **Object**, and ending with the original target class. All of this information is saved in the process described above. The information that has been saved is then displayed in the output scroll pane in the reverse order from which it was saved.

The method signature

Listing 2 shows the beginning of the method named **doInheritance()**. As you can see, the method throws a **ClassNotFoundException**. Therefore, the method call in the **actionPerformed()** method shown earlier is enclosed in a *try* block.

```
  void doInheritance()
           throws
ClassNotFoundException{
```

Listing 2

Temporary storage

Listing 3 shows the declaration of a new object of the class **Vector**.

```
    Vector inherVector = new Vector();
```

Listing 3

In case you are unfamiliar with the use of the **Vector** class, an object of this class is an extremely useful container for an unknown amount of data that needs to be accessed later using an ordinal index. *(You can find a detailed discussion of the Vector class among the many tutorial lessons on my web site.)*

In this case, we need a container for an unknown amount of data that needs to be accessed later using an ordinal index, so a **Vector** object is my container of choice. To begin with, there is no advance knowledge of how many different classes comprise the family tree. Further, it will later be necessary to access the class names in the reverse of the order in which they are saved in order to display them.

The target class

During operation, the user enters the name of the target class in the text field shown in the lower left of the GUI in Figure 1. The code in Listing 4 invokes the **getText()** method on the reference to that **JTextField** object to obtain the name of the target class as a **String**.

```
    String theClass = targetClass.
                             getText();
```

Listing 4

Working variables

The code in Listing 5 shows the declaration of a pair of working variables that will be used later. The first variable named **theClassObj** is used to refer to an object of the **Class** class that represents the target class.

The second variable named **theSuperClass** is used to refer to an object of the **Class** class that represents the superclass of the target class.

```
    Class theClassObj = null;
    Class theSuperClass = null;
```

Listing 5

A while loop

Listing 6 shows the beginning of a **while** loop that is used to get and save the names of all the classes in the inheritance family tree. Note that this loop continues to iterate as long as the name of the class is not **java.lang.Object**. Once the **Object** class is encountered, execution of the loop is terminated.

```
    while(!(theClass.equals(
                "java.lang.Object"))){
      inherVector.add(theClass);
```

Listing 6

Saving the class name

At the beginning of the loop, the **add()** method is used to store the name of the class in the **Vector** object referred by **inherVector**. This is shown in Listing 6 above.

Getting the superclass

As explained in an earlier lesson, if you have a **Class** object that represents a target class, you can use that object to obtain another **Class** object that represents the superclass of the target object.

The first statement in Listing 7 uses the **forName()** method of the **Class** class to get and save a reference to a **Class** object that represents the target class whose name is stored in the **String** object referred to by **theClass**.

The second statement in that fragment uses the **getSuperclass()** method to get and save a reference to a **Class** object that represents the superclass of the target class.

```
      theClassObj = Class.forName(
                             theClass);
      theSuperClass = theClassObj.
                       getSuperclass();
```

Listing 7

Interfaces

Once you have a **Class** object that represents the target class, you can use the method named **getInterfaces()** to get a reference to an array of **Class** objects, each of which represents an interface implemented by the target class.

Although identification of the interfaces implemented by the target class is not the primary purpose of this lesson, this information will be needed later in the program. Therefore, the code in Listing 8 gets and saves that interface information in a **Vector** object referred to by an instance variable named **intfcsVector**.

The use of this information to produce the display shown in the top-left scroll pane of Figure 1 will be discussed in the next lesson.

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

Listing 8

Note that a class doesn't necessarily implement any interfaces. Therefore, the code in Listing 8 first tests to see if the target class does implement any interfaces before trying to get the information and save it in the vector.

The superclass name

At this point, the reference variable named **theSuperClass** contains a reference to a **Class** object that represents the superclass of the target class. We need to extract the name of the superclass from that object. The code in Listing 9 shows how to do this using the **getName()** method of the **Class** class.

```
      theClass = theSuperClass.
                          getName();
    }//end while loop
```

Listing 9

Make the superclass into a target class

The resulting **String** is a fully qualified class name for the superclass. This string is assigned to the variable named **theClass** causing it to become the target class for the next iteration.

Control then returns to the top of the while loop where the process is repeated unless the name of the target class is **Object**. In that event, the loop is terminated and the code in Listing 10 is executed.

Saving name of Object class

The code in Listing 10 simply adds the name of the **Object** class to the storage vector so that the contents of that vector will include the names of all of the classes in the family tree, beginning with the target class and ending with the class named **Object**.

```
    inherVector.add(
                   "java.lang.Object");
```

Listing 10

Display the family tree

The **Vector** object referred to by **inherVector** now contains the names of all the classes in the family tree. However, they are in the reverse of the order in which they need to be displayed.

The inheritance family tree, consisting of the class names in the vector, needs to be displayed in the **JTextArea** object referred to by **inher**. That object is displayed in the upper-left of Figure 1.

Contents of a JTextArea object

A **JTextArea** object contains a single **String** object. Therefore, it is necessary to convert the contents of the storage vector to a **String** in the correct format, and to store that **String** in the **JTextArea** object. This involves extracting the class names from the storage vector and appending them to the **String** contents of the **JTextArea** object.

Storing the family tree in the JTextArea object

The **for** loop in Listing 11 extracts the names of the classes from the storage vector referred to by **inherVector** in reverse order. It appends those names to the contents of the **JTextArea** object referred to by **inher**.

Because each class in the family tree needs to be displayed on a separate line, the code in Listing 11 also appends newline code between each of the class names.

```
    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
```

Listing 11

Downcasting is required

Note that the contents of a **Vector** object are references to objects stored in the vector as type **Object**. Therefore, it is necessary to downcast those references to type **String** before the strings to which they refer can be appended to the contents of the **JTextArea** object.

Summary

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 about the target component:

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

In this lesson, I explained the **doInheritance()** method in detail. In the discussion, I showed you how to use a **Class** object representing a target class to obtain a **Class** object representing the superclass of the target class.

I showed you how to use iteration to obtain the complete family tree of the target class, all the way up to the class named **Object**.

I showed you how to display that information in a **JTextArea** object.

What's Next?

In the next lesson, I will explain how the method named **doInterfaces()** obtains and displays interface information about the target class, in alphabetical order, in the upper-right output pane in Figure 1.

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

Complete Program Listing

A complete listing of the program is provided in Listing 12.

```
/*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 12

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-