Richard G Baldwin (512) 223-4758, baldwin@austin.cc.tx.us,
http://www2.austin.cc.tx.us/baldwin/

Event Handling in JDK 1.1, Container Events and More on Inner Classes

Java Programming, Lecture Notes # 95, Revised 02/24/99.

Preface

Students in Prof. Baldwin's **Intermediate Java Programming** classes at ACC are responsible for knowing and understanding all of the material in this lesson.

On 2/24/99, the sample programs in this lesson were confirmed to operate properly under JDK 1.2.

Introduction

This lesson was originally written on December 10, 1997, using the JDK 1.1.3 download package. The purpose of this lesson is to illustrate the use of container events, and to further illustrate the use of inner classes, both named and anonymous.

Overview

If you instantiate an object of type **ContainerListener** and register that object on a **Container** object, the listener object will be notified whenever an object is added to or removed from the container. Information regarding the object that was added is passed into the **ContainerListener** methods in the form of an object of type **ContainerEvent**.

First Sample Program

This program illustrates container events. It also further illustrates the use of inner classes, both named and anonymous.

The program also illustrates a programming style involving the use of the controlling class as a listener class that implements multiple listener interfaces. In this situation, the listener methods are defined as methods of the controlling class rather than being defined as methods of separately compiled listener classes.

Finally, it illustrates the fact that the code in one event handler can register event listeners for other components.

In this program, a **ContainerListener** object monitors for the addition of buttons to a **Frame** object. Each time a new button is added to the **Frame**, an **ActionEvent** listener is registered on that new **Button** object. All of the **ActionEvent** listener objects that are registered on the buttons are of the same **ActionEvent** class.

To make the whole thing interesting, this program creates the keyboard and provides the skeleton code for a four-function calculator, including the event handler needed to respond to calculator keystrokes (pressing the buttons on the calculator with the mouse) and identify the button that was pressed on the calculator keyboard.

The program does not provide the code that would be required to process the keystrokes to do the calculations, or to display the results of the calculations since that is not the primary purpose of the program.

However, the program does provide a place to display results in the form of a yellow **Label** object and it uses that display area to display the text on the button each time a calculator button is pressed..

When you run the program and click on the buttons in the **Frame** object, the labels on the buttons are displayed in the yellow **Label** object.

This program was tested using JDK 1.1.3 under Win95.

Interesting Code Fragments

Because of its unusual structure, this program contains a number of interesting code fragments. We will begin with the first few lines of the definition of the controlling class. Note in particular that the controlling class implements two listener interfaces. Having done this, the listener methods declared in these interfaces can be (must be) defined in the controlling class, and an object of the controlling class (**this**) is the listener object which gets registered on itself (**this**).

We also declare an instance variable named **displayLabel**. Later we will use this variable to refer to a yellow **Label** object that has been added to the **Frame** object to serve as a display window.

```
class Event32 extends Frame
  implements ContainerListener,ActionListener{

    Label displayLabel;//reference to a Label for display
```

We will skip the **main** method (that does nothing but instantiate an instance of the controlling class) and move on to a method that will add a series of buttons to the **Frame** object. The number of buttons added and the labels on the buttons are determined from the length and the substring values of an incoming **String** object.

```
  void addButtons(String labels){
    for(int cnt = 0; cnt < labels.length();cnt++){
      this.add(new Button(labels.substring(cnt,cnt+1)));
    }//end for loop
  }//end addButtons()
```

That brings us to the constructor where we set the layout manger to **GridLayout** with four columns and an unspecified number of rows. (Setting the number of rows to zero causes it to be unspecified.) We also set a vertical and horizontal gap size of three pixels between the components that are to be placed in the container.

```
  this.setLayout(new GridLayout(0,4,3,3));
```

The next interesting code fragment is a statement in the constructor to register a listener object to be notified of container events. This listener object will receive an event notification each time a component is added to the container.

The most interesting thing about this code fragment is the dual use of the **this** reference. The object of the controlling class is the *source* of events. In this case, it is a **Frame** object.

The *listener* object that is being registered to receive notification of events is also the object of the controlling class. This is possible because the controlling class implements the **ContainerListener** interface and contains definitions of the two methods declared in that interface.

Thus, this statement registers the object of the controlling class as a listener on itself as a source (hence **this** as both a source and a listener).

```
    this.addContainerListener(this);
```

Next, we invoke the **addButtons()** method, defined above, to add a series of buttons to this **Frame** object with the labels being determined by the individual characters in the **String** object passed as a parameter.

```
    this.addButtons("789+456-123=0.X/C");
```

We also add a display label and make it yellow. This is a quick and easy way to create a place to display information but the placement of the display area doesn't look much like a calculator. A better way would be to construct two **Panel** objects, one for the display and the other for the keyboard, and to place one above the other in the layout.

```
    displayLabel = new Label("00000");
    this.add(displayLabel);
    displayLabel.setBackground(Color.yellow);
```

Following this, we register an anonymous **WindowListener** object using an anonymous **WindowListener** class. This listener object will cause the program to terminate when the user clicks the *close* button on the **Frame** object. Note that this class automatically **extends** the **WindowAdapter** class.

If you don't understand this syntax, you should go back and review the lesson on inner classes.

```
     this.addWindowListener(
      new WindowAdapter(){//anonymous class definition
        public void windowClosing(WindowEvent e){
          System.exit(0);//terminate the program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener
  }//end constructor
```

That completes the interesting code in the constructor.

Following the constructor, we define the **componentAdded** method that is declared in the **ContainerListener** class (which is implemented by the controlling class of this program). This method is invoked each time a new component is added to this **Frame** object.

Each time a component is added, this method confirms that the new component is a button, and if so, it registers an **ActionListener** object to process action events on that individual button. All of the **ActionListener** objects are of the same **ActionListener** class. The **ActionListener** class is this extended **Frame** class which implements the **ActionListener** interface and defines the **actionPerformed** method of that interface.

Again note that **this** object is the action listener object being registered because the controlling class implements the **ActionListener** interface, and the **actionPerformed()** method is defined in the controlling class.

The **getChild()** method of the **ContainerEvent** object returns a reference to the component that was added to the container creating the event in the first place. The reference is returned as type

Component, cast to a **Button**, and assigned to the reference variable named **button**. The object named **button** then becomes the source on which the **ActionListener** object is registered.

We also define an empty **componentRemoved()** method to satisfy the compiler (all methods declared in the **ContainerListener** interface must be defined).

```
  public void componentAdded(ContainerEvent e){
    if(e.getID() == ContainerEvent.COMPONENT_ADDED){
      if(e.getChild() instanceof Button){
        Button button = (Button)e.getChild();
        //Register an ActionListener object on the button
        button.addActionListener(this);
      }//end if statement
    }//end if statement
  }//end componentAdded()


  public void componentRemoved(ContainerEvent e){}
```

Finally, we see the **actionPerformed** method that is invoked whenever the user clicks one of the buttons that are added to the **Frame** object.

All this method does is display the button's label. However, if you wanted to expend the programming effort, you could implement the calculator logic at this point.

```
  public void actionPerformed(ActionEvent e){
    displayLabel.setText(e.getActionCommand());
  }//end actionPerformed
```

A complete listing of the first sample program is presented in the next section.

First Sample Program Listing

Many of the interesting code fragments are highlighted in **boldface** in the following program listing to assist you in viewing them in context.

```
/*File Event32.java
Copyright 1997, R.G.Baldwin

This program illustrates container events.

It also illustrates the use of inner classes.

It also illustrates the use of the controlling class as a
listener class that implements multiple listener
interfaces.  In this situation, the listener methods are
defined as methods of the controlling class rather than
```

```
being defined as methods of separately compiled listener
classes.

It also illustrates the fact that the code in one event
handler can register event listeners for other components.

In particular, a ContainerListener object monitors for the
addition of buttons to a Frame object.  Each time a new
button is added to the Frame, an ActionEvent listener is
registered on that new Button object.  All of the
ActionEvent listener objects that are registered on the
buttons are of the same ActionEvent class.

To make the whole thing interesting, this program creates
the keyboard and provides the skeleton code for a
four-function calculator, including the event handler
needed to respond to keystrokes and identify the key.
However, it does not provide the code that would be
required to process the keystrokes or display the results
of the calculations since that is not the primary purpose
of the program. It does provide a place to display results
in the form of a yellow Label object.

When you run the program and click on the buttons in the
Frame object, the labels on the buttons are displayed in
the yellow display area.

This program was tested using JDK 1.1.3 under Win95.

**********************************************************/
import java.awt.*;
import java.awt.event.*;

//Note that the controlling class implements both the
// ContainerListener interface and the ActionListener
// interface.
class Event32 extends Frame
  implements ContainerListener,ActionListener{

    Label displayLabel;//reference to a Label for display

  //=======================================================//

  public static void main(String[] args){
    new Event32();//instantiate this object
   }//end main
  //=======================================================//

  //This method adds a series of Button objects to this
  // Frame object.  The number of buttons added and the
  // labels on the buttons are determines from the length
  // and the substring values of an incoming String object.
  void addButtons(String labels){
    for(int cnt = 0; cnt < labels.length();cnt++){
      this.add(new Button(labels.substring(cnt,cnt+1)));
    }//end for loop
```

```
  }//end addButtons()

  //=======================================================//

  public Event32(){//constructor

    //Set the layout manager to GridLayout with 4 col and
    // an unspecified number of rows (0) with a three-pixel
    // horizontal and vertical gap.
    this.setLayout(new GridLayout(0,4,3,3));

    this.setSize(280,280);
    this.setTitle("Copyright 1997 R.G.Baldwin");

    //Add a container listener which will receive an event
    // notification each time a component is added to the
    // container. The container is this Frame object.  Also
    // the listener class that implements the
    // ContainerListener interface is this extended Frame
    // class (hence this as an argument).
    this.addContainerListener(this);

    //Add a series of buttons to this Frame object with
    // the labels determined by the individual characters
    // in the String object passed as a parameter.
    this.addButtons("789+456-123=0.X/C");

    //Add a display label and make it yellow.
    displayLabel = new Label("00000");
    this.add(displayLabel);
    displayLabel.setBackground(Color.yellow);


    this.setVisible(true);

    //-------------------------------------------------//
    //Anonymous inner-class listener to terminate program
    this.addWindowListener(
      new WindowAdapter(){//anonymous class definition
        public void windowClosing(WindowEvent e){
          System.exit(0);//terminate the program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener
  }//end constructor

  //=======================================================//
  //This is the overridden componentAdded method of the
  // ContainerListener class.  It is invoked each time a
  // new component is added to this Frame object.  Note
  // that each time a component is added, it confirms that
  // the component is a button, and if so, it registers an
  // ActionListener object to process Action events on that
  // individual button.  All of the ActionListener objects
  // are of the same class.  The ActionListener class is
  // this extended Frame class which implements the
```

```
  // ActionListener interface and overrides the
  // actionPerformed method of that interface.
  public void componentAdded(ContainerEvent e){
    if(e.getID() == ContainerEvent.COMPONENT_ADDED){
      if(e.getChild() instanceof Button){
        Button button = (Button)e.getChild();
        //Register an ActionListener object on the button
        button.addActionListener(this);
      }//end if statement
    }//end if statement
  }//end componentAdded()

  //This empty method is required to satisfy the compiler
  // because this extended Frame class implements the
  // ContainerListener interface.
  public void componentRemoved(ContainerEvent e){}

  //====================================================//
  //This is the actionPerformed method that is invoked
  // whenever the user clicks one of the buttons that are
  // added to the Frame object.  All it does is display
  // the button's label.
  // However, if you wanted to expend the programming
  // effort, you could implement the calculator at this
  // point.
  public void actionPerformed(ActionEvent e){
    displayLabel.setText(e.getActionCommand());
  }//end actionPerformed
  //====================================================//
}//end class Event32
//======================================================//
```

Second Sample Program Discussion

This program replicates the functionality of the previous program named Event32.java. However, it does so using only anonymous inner classes. This program is provided so that you can compare the syntax using only anonymous inner classes with the comparable version named Event32 that uses named inner classes for the listener classes.

This program was tested using JDK 1.1.3 under Win95.

Second Sample Program Listing

The anonymous inner classes in this program are highlighted using either **boldface** or *Italics*. Note that this program illustrates nested anonymous inner classes. In particular, the anonymous inner class for the **ActionListener** (*Italics*) is nested inside the anonymous inner class for the **ContainerListener** (highlighted in **boldface**).

```
/*File Event33.java
Copyright 1997, R.G.Baldwin
```

```
This program replicates the functionality of the program
named Event32.java.  However, it does so using only
anonymous inner classes.  This program is provided so
that you can compare the syntax using only anonymous
inner classes with a comparable version named Event32 that
uses named inner classes for the listener classes.

This program was tested using JDK 1.1.3 under Win95.

**********************************************************/
import java.awt.*;
import java.awt.event.*;

//Note that this controlling class does not implement any
// listener interfaces.  They are effectively implemented
// in the inner classes.
class Event33 extends Frame{

  Label displayLabel;//reference to a Label for display

  //=======================================================//
  public static void main(String[] args){
    new Event33();//instantiate this object
   }//end main
  //=======================================================//

  //This method adds a series of Button objects to this
  // Frame object.  The number of buttons added and the
  // labels on the buttons are determines from the length
  // and the substring values of an incoming String object.
  void addButtons(String labels){
    for(int cnt = 0; cnt < labels.length();cnt++){
      this.add(new Button(labels.substring(cnt,cnt+1)));
    }//end for loop
  }//end addButtons()
  //=======================================================//

  public Event33(){//constructor

    this.setLayout(new GridLayout(0,4,3,3));
    this.setSize(280,280);
    this.setTitle("Copyright 1997 R.G.Baldwin");


    //The following anonymous class definition registers an
    // anonymous ContainerListener object on this Frame
    // object.  The ContainerListener object in turn
    // registers an anonymous Action Listener object on
    // each of the buttons that are added to the Frame
    // object.
    this.addContainerListener(//add anonymous Listener
      new ContainerListener(){
        public void componentAdded(ContainerEvent e){
          if(e.getID() == ContainerEvent.COMPONENT_ADDED){
            if(e.getChild() instanceof Button){
```

```
              Button button = (Button)e.getChild();
              //The following code registers an anonymous
              // ActionListener object on each button
              // object that is added to the Frame object.
              button.addActionListener(
                new ActionListener(){
                  public void actionPerformed(ActionEvent e){
                    displayLabel.setText(e.getActionCommand());
                  }//end actionPerformed
                }//end new actionListener
              );//end add anonymous ActionListener
            }//end if statement
          }//end if statement
        }//end componentAdded()

        //The following empty method is required to satisfy
        // the requirement to define all of the methods
        // that are declared in the ContainerListener
        // interface.
        public void componentRemoved(ContainerEvent e){}
      }//end new ContainerListener
    );//end add anonymous ContainerListener

    //-----------------------------------------------//
    //Add a series of buttons to this Frame object with
    // the labels determined by the individual characters
    // in the String object passed as a parameter.
    this.addButtons("789+456-123=0.X/C");

    //Add a display label and make it yellow.
    displayLabel = new Label("00000");
    this.add(displayLabel);
    displayLabel.setBackground(Color.yellow);
    this.setVisible(true);

    //-----------------------------------------------//
    //Anonymous inner-class listener to terminate program
    this.addWindowListener(
      new WindowAdapter(){//anonymous class definition
        public void windowClosing(WindowEvent e){
          System.exit(0);//terminate the program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener
  }//end constructor
  //=================================================//
}//end class Event33
//=================================================//
```

-end-