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

Event Handling in JDK 1.1, Handling Events in Extended Components without Listener Objects

Java Programming, Lecture Notes # 102, Revised 02/25/99.

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.

JDK 1.1 was formally released on February 18, 1997 and JDK 1.1.1 was formally released on March 27, 1997. This lesson was originally written on March 29, 1997 using the software and documentation in the JDK 1.1 download package because I experienced installation problems with JDK 1.1.1 and I had not succeeded in getting it installed by March 29.

On 2/25/99, the sample program in this lesson was confirmed to work properly under JDK 1.2 and Win95.

Introduction

The purpose of this lesson is to illustrate one way that you can create your own custom components by extending the existing components. When you extend an existing component, you need to handle the events associated with the new component.

This lesson demonstrates an event-handling approach for the events on the new component that does not follow the *Source/Listener* mode of the *Delegation Event Model*. Sometimes this approach can be more compact. However, you should take a look at the cautions in the JDK 1.1 documentation regarding this approach.

Essential Ingredients for Extending Existing Components

Using the techniques illustrated in this program, the essential ingredients for creating and handling events on an extended component are:

- Define and instantiate the primary container for the **GUI**. In this lesson, the primary container extends **Frame**.
- Define the class for the new component and make it *extend* an existing component. In this program the class for the new component *extends* **TextField**.
- Create a mechanism for handling events on the new component. In this program, the mechanism is an overridden **processKeyEvent()** method. Note that **processKeyEvent()** is only one of several available methods that you can override. Other available methods include **processMouseEvent()**, **processActionEvent()**, etc.
    - If you use methods such as **processKeyEvent()** for processing events, you must *enable* events of the proper type using statements such as **enableEvents(AWTEvent.KEY_EVENT_MASK)**. A good place to put this statement is in the constructor for the object of the new type.
    - Also, if you override one of the event processing methods such as **processKeyEvent()** you must always invoke the superclass version of the same event using a statement such as **super.processKeyEvent(e)** passing it the event object as a parameter.
    - If you take this approach, it isn't necessary to instantiate and register listener objects on the new components. Rather, all of the event handling code can be placed inside the **processKeyEvent()** method.
- Add the new component to its container.

.

The following rather long explanation of the rule regarding **super.processKeyEvent(e)** along with a sample program was extracted from the JavaSoft JDK 1.1.3 documentation at the location shown below.

/java/docs/guide/awt/designspec/events.html

The most important information has been highlighted using boldface.

Note in particular the *unconditional* call to **super.processFocusEvent(e)** at the end of the method named **processFocusEvent(FocusEvent e)**. Not all books show this as an unconditional call.

```
By default, the single processEvent method will invoke the
proper
event-class processing method. The event-class processing
method by
default will invoke any listeners, which are registered.


It's important to remember that these methods perform a
```

**critical
function in the event processing for an AWT component and so if you
override them you should remember to call the superclass' method
somewhere within your own!**

Selecting for Event Types

One of the goals of the listener model is to improve performance by NOT
delivering events which components are not interested in. By default, if
a listener type is not registered on a
component, then those events will NOT be delivered and these processing
methods will therefore NOT be called. So if you are using this extension
mechanism for event-handling, you'll need to select for the specific
types of events your component wishes to receive (in case no listener is
registered). This can be done by using the following method on
java.awt.Component:

```
     protected final void enableEvents(long eventsToEnable)
```

The parameter to this method is a bitwise mask of the event types you
wish to enable. The event masks are defined in java.awt.AWTEvent. Note
that changing this mask will not affect the delivery of events to
listeners -- it only controls the delivery to the component's processing
methods. The bottom line is that the set of events which are delivered
to processEvent() is defined by the union of event types which have
listeners registered and event types explicitly turned on via enableEvents().

Example using Extension Mechanism

Following is an example of how this extension mechanism may be used. For
Example, if a subclass of java.awt.Canvas wishes to render some visual
feedback when it receives/loses keyboard focus, it could do the
following.

```
    public class TextCanvas extends Canvas {
          boolean haveFocus = false;
```

```
        public TextCanvas() {
            enableEvents(AWTEvent.FOCUS_EVENT_MASK);
            ...
        }
        protected void processFocusEvent(FocusEvent e) {
            switch(e.getID()) {
              case FocusEvent.FOCUS_GAINED:
                haveFocus = true;
                break;
              case FocusEvent.FOCUS_LOST:
                haveFocus = false;
            }
            repaint();

            // let superclass dispatch to listeners
            super.processFocusEvent(e);
        }
        public void paint(Graphics g) {
            if (haveFocus) {
                // render focus feedback...
            }
        }
        ...rest of TextCanvas class...
    }
```

Note that this is not the only way to handle events on extended components. Event handling in JDK 1.1 is extremely flexible. This is simply the way chosen for this particular lesson.

Sample Program

This program is designed to be compiled and run under JDK 1.1. It illustrates the ability to handle events in extended components without the requirement to use **Listener** objects.

The program extends **TextField** to create a new type of *text field* component named **NumericTextField**. Objects of the **NumericTextField** type will only accept numeric input. If the user attempts to enter any character other than *0 through 9*, an audible alarm sounds (assuming the system setup supports audible alarms) and the character is not accepted by the runtime system.

The controlling class for the program extends **Frame**. Thus, all the **GUI** action takes place inside a **Frame** object.

A total of three component objects are instantiated and added to the **Frame** object. One of the objects is an object of the new **NumericTextField** class described above.

A second object is a standard **Button** object. The third object is a **Label** object.

Whenever the user clicks on the **Button**, the **String** data inside the **NumericTextField** object is copied into the **Label** object.

If the user clicks on the *close* button on the **Frame**, the program terminates.

The extended component named **NumericTextField** is created by extending the **TextField** class and providing the capability to use **key** events to filter the characters entered into the **NumericTextField** object by the user. Only *numbers* are allowed through the filter.

Although the operation of the new component requires the use of *key events*, it does not operate on the basis of the *Source/Listener* mode of the *Delegation Event Model*.

Rather, *key events* are enabled on all objects of the class using the **enableEvents()** method with a **KEY_EVENT_MASK**. Once this is done, the method named **processKeyEvent()** is invoked by the runtime system whenever a *key event* occurs on an object of the class. The method named **processKeyEvent()** is overridden to provide all of the processing necessary to filter out non-numeric characters without the requirement to instantiate a separate **Listener** object.

The processing inside the **processKeyEvent()** method is straightforward. Whenever a **KEY_TYPED** event occurs, the character typed is intercepted and tested to confirm that it is one of the numeric characters. If not, it is replaced by a character with a zero value. This is not a legal character, so the runtime system beeps and refuses to accept it into the component.

Whenever you override one of the **processXxxxEvent()** methods, you should always invoke the same method in the superclass passing the event object as a parameter. This makes certain that all necessary default processing takes place.

The program was tested using JDK 1.1 running under Win95.

Interesting Code Fragments

The first interesting code fragment is the instantiation and adding of an object of the new extended type to the primary GUI **Frame** object.

```
    NumericTextField myNumericTextField;
    add(myNumericTextField =
          new NumericTextField());//add a
custom component
```

The next interesting code fragment is inside the constructor for the new extended component object. Whenever this statement has been executed, any key events that occur on an object of the class are automatically delivered to a method named **processKeyEvent()**.

```
   enableEvents(AWTEvent.KEY_EVENT_MASK);
```

Another way of describing the situation is that once the above statement has been executed, the occurrence of a key event on an object of the class will cause a method named **processKeyEvent()** to be automatically invoked and an event object will be passed to the method as a parameter. This method can be overridden to produce the behavior desired.

The next interesting code fragment shown below is the entire overridden version of the **processKeyEvent()** method.

```
  protected void processKeyEvent(KeyEvent e){
    if(e.getID() == KeyEvent.KEY_TYPED)//KEY_TYPED is key-down and
key-up
      if(!((e.getKeyChar() = '0') && (e.getKeyChar() <= '9')))
        e.setKeyChar('\000');
    super.processKeyEvent(e);//always do this when overriding
processXxEvent
  }//end processKeyEvent
```

There are a number of different kinds of key events that are delivered to this method. The code in the method ignores all except **KEY_TYPED**. This is the type of event that occurs when a key has been pressed and released.

There are also a number of methods available to work with the **KeyEvent** object passed in as a parameter. It is important to note that in this case, processing occurs before the character associated with the key event is actually deposited into the extend **TextField** object. Therefore, it is possible to intercept the character using the **KeyEvent** object and modify it before it is deposited.

The logic inside this method simply confirms that the character is a numeric key, and if not, replaces the character by a character having an octal value of '\000'. This is an illegal character insofar as the runtime system is concerned, so the runtime system beeps and refuses to deposit it into the extended **TextField** object.

This method also invokes the **processKeyEvent()** method of the *superclass* passing the event object as a parameter. You must never forget to do this because that method in the superclass performs a number of tasks critical to the overall operation of the system. The requirement to remember to do this is one of the reasons that Sun cautions against using this method of event handling.

The remaining code in the program simply creates some standard **Listener** classes for copying the data in the new extended component into a **Label** object, and for terminating the program when the user closes the **Frame**.

These code fragments have illustrated the essential ingredients of creating an extended component and handling the events for that component without the requirement to instantiate and register **Listener** objects. A complete listing of the program is contained in the next section.

Program Listing

This section contains a complete listing of the program. Refer to previous sections for an operational description of the program.

```
/*File Event29.java Copyright 1997, R.G.Baldwin
This program is designed to be compiled and run under
JDK 1.1

This program illustrates the ability to handle events in
extended components without the requirement to use Listener
objects.

The program extends TextField to create a new type of text
field named NumericTextField.  Objects of the new type will
only accept numeric input.  If the user attempts to enter
any character other than 0 thru 9, an audible alarm sounds
and the character is not accepted.

The controlling class for the program extends Frame.  Thus,
all the GUI action takes place inside a Frame object.

Three component objects are instantiated and added to the
Frame object. One of the objects is an object of the
NumericTextField class described above.

A second object is a standard Button object.  The third
object is a Label object.

Whenever the user clicks on the Button, the String data
inside the NumericTextField object is copied into the
Label object.

If the user clicks on the close button on the Frame, the
program terminates.

The custom component named NumericTextField is created by
extending the TextField class and providing the capability
to use key events to filter the characters entered into
the NumericTextField object.  Only numbers are allowed
through the filter.

Although the operation of the new component requires the
use of key events, it does not operate on the basis of the
Source/Listener model of the Delegation Event Model.

Rather, key events are enabled on all objects of the class
using the enableEvents() method with a KEY_EVENT_MASK.
Once this is done, the method named processKeyEvent() is
invoked whenever a key event occurs on an object of the
class.  The method named processKeyEvent()is overridden to
provide all of the processing necessary to filter out
non-numeric characters without the requirement to
instantiate a separate Listener object.
```

```
The processing inside the processKeyEvent() method is
straightforward. Whenever a KEY_TYPED event occurs, the
character typed is intercepted and tested to confirm that
it is one of the numeric characters.  If not, it is
replaced by a character with a zero value.  This is not a
legal character, so the runtime system beeps and refuses
to accept it into the component.

Whenever you override one of the processXxxxEvent()
methods, you should always invoke the same method in the
superclass passing the event object as a parameter.  This
makes certain that all necessary default processing takes
place.

The remaining code consists of a couple of standard
Listener classes. One is used with the Button object to
copy the contents of the NumericTextField object to the
Label object.  The other terminates the program when the
user clicks on the close box on the Frame.

The program was tested using JDK 1.1.3 running under Win95.
*/
//=======================================================
import java.awt.*;
import java.awt.event.*;

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

public class Event29 extends Frame{
  public static void main(String[] args){
    new Event29();//instantiate an object of this type
  }//end main
//-------------------------------------------------------
  public Event29(){//constructor
    this.setTitle("Copyright 1997, R.G.Baldwin");
    this.setLayout(new FlowLayout());
    this.setSize(250,100);

    Button myButton;
    add(myButton = new Button("Copy to Label"));

    NumericTextField myNumericTextField;
    //add a custom component
    add(myNumericTextField = new NumericTextField());

    Label myLabel;
    add(myLabel = new Label("Initial Text"));

    this.setVisible(true);
    myNumericTextField.requestFocus();

    myButton.addActionListener(
         new MyActionListener(myLabel,myNumericTextField));
    this.addWindowListener(new Terminate());
  }//end constructor
```

```
}//end class Event29
//===========================================================

//Class to define a new type of TextField.  This is a
// custom TextField component, extended from TextField.
// It will only accept numeric values.  Note that it
// depends on key events for its operation but it does not
// use the source/listener mode of the Delegation Event
// Model.
class NumericTextField extends TextField{

  NumericTextField(){//constructor
    this.setColumns(10);//set the size
    //Enable key events so that the processKeyEvent()
    // method will be invoked whenever a key event occurs
    // on an object of this class.
    enableEvents(AWTEvent.KEY_EVENT_MASK);
  }//end constructor
  //-----------------------------------------------------


  //Because key events are enabled, this overridden method
  // will automatically be invoked whenever a key event
  // occurs on an object of the class.
  protected void processKeyEvent(KeyEvent e){
    //KEY_TYPED is key-down and key-up
    if(e.getID() == KeyEvent.KEY_TYPED)
      //If the char is not numeric, substitute an illegal
      // character so that the runtime system will reject
      // it and beep.
      if(!((e.getKeyChar() = '0')
                                 && (e.getKeyChar() <= '9')))
        e.setKeyChar('\000');
    //always do this when overriding processXxEvent
    super.processKeyEvent(e);
  }//end processKeyEvent
}//end class NumericTextField
//===========================================================

//Class to create an ActionListener for the Button object.
// Transfers the data from the NumericTextField to the
// Label.
class MyActionListener implements ActionListener{
  Label myLabel;
  NumericTextField myNumericTextField;
  //-----------------------------------------------------

  MyActionListener( //constructor
                Label inLbl, NumericTextField inNumTxtFld){
    myLabel = inLbl;
    myNumericTextField = inNumTxtFld;
  }//end constructor
  //-----------------------------------------------------

  public void actionPerformed(ActionEvent e){
```

```
    myLabel.setText(myNumericTextField.getText());
  }//end actionPerformed()
}//end MyActionListener
//=========================================================

class Terminate extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    //terminate the program when the window is closed
    System.exit(0);
  }//end windowClosing
}//end class Terminate
//=========================================================
```

Review

Q - Write an application that meets the specifications given below.

A - See the application that follows:

```
/*File SampProg131.java Copyright 1997, R.G.Baldwin
This program is designed to be compiled and run under
JDK 1.1

Without viewing the solution that follows, write an
application that contains an object that will accept text
input, a Button object, and a Label object in a standard
frame.  For purposes of the remainder of this
specification, the first object listed above will be
referred to as the Text Object.

If you enter lower case letters into the Text Object, they
are automatically converted to upper case before being
displayed.

When you click on the Button, the data inside the Text
Object are copied into the Label object.

When you click on the close button on the Frame, the
program terminates and returns control to the operating
system.

The program was tested using JDK 1.1.3 running under Win95.
*/
//=========================================================
import java.awt.*;
import java.awt.event.*;

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

public class SampProg131 extends Frame{
  public static void main(String[] args){
    new SampProg131();//instantiate an object of this type
```

```
  }//end main
//-----------------------------------------------------------
  public SampProg131(){//constructor
    this.setTitle("Copyright 1997, R.G.Baldwin");
    this.setLayout(new FlowLayout());
    this.setSize(250,100);

    Button myButton;
    add(myButton = new Button("Copy to Label"));

    NumericTextField myCustomTextField;
    //add a custom component
    add(myCustomTextField = new NumericTextField());

    Label myLabel;
    add(myLabel = new Label("Initial Text"));

    this.setVisible(true);

    myButton.addActionListener(
         new
MyActionListener(myLabel,myCustomTextField));
    this.addWindowListener(new Terminate());

  }//end constructor

}//end class SampProg131
//===========================================================

//Class to define a new type of TextField.  This is a
// custom TextField component, extended from TextField.
// It will only accept numeric values.  Note that it
// depends on key events for its operation but it does not
// use the source/listener mode of the Delegation Event
// Model.
class NumericTextField extends TextField{

  NumericTextField(){//constructor
    this.setColumns(10);//set the size
    //Enable key events so that the processKeyEvent()
    // method will be invoked whenever a key event occurs
    // on an object of this class.
    enableEvents(AWTEvent.KEY_EVENT_MASK);
  }//end constructor
  //-----------------------------------------------------------


  //Because key events are enabled, this overridden method
  // will automatically be invoked whenever a key event
  // occurs on an object of the class.
  protected void processKeyEvent(KeyEvent e){
    //KEY_TYPED is key-down and key-up
    if(e.getID() == KeyEvent.KEY_TYPED)
      //If the char is lower case, convert it to upper
      // case.
      if(e.getKeyChar()  'Z')
```

```
        e.setKeyChar((char)(e.getKeyChar() -('a' - 'A')));
    //always do this when overriding
processXxEvent
    super.processKeyEvent(e);
  }//end processKeyEvent
}//end class NumericTextField
//===========================================================

//Class to create an ActionListener for the Button object.
// Transfers the data from the NumericTextField to the
// Label.
class MyActionListener implements ActionListener{
  Label myLabel;
  NumericTextField myCustomTextField;
  //-----------------------------------------------------------

  MyActionListener( //constructor
                Label inLbl, NumericTextField inNumTxtFld){
    myLabel = inLbl;
    myCustomTextField = inNumTxtFld;
  }//end constructor
  //-----------------------------------------------------------

  public void actionPerformed(ActionEvent e){
    myLabel.setText(myCustomTextField.getText());
  }//end actionPerformed()
}//end MyActionListener
//===========================================================

class Terminate extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    //terminate the program when the window is closed
    System.exit(0);
  }//end windowClosing
}//end class Terminate
//===========================================================
```

-end-