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

Event Handling in JDK 1.1, Posting Synthetic Events to the System Event Queue

Java Programming, Lecture Notes # 104, Revised 04/06/98.

- Preface
- Introduction
- Key Ingredients for Posting Events
- Sample Program
    - Interesting Code Fragments
    - Program Listing
- A Timer Program
- Review

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 released on February 18, 1997 and JDK 1.1.1 was released on March 27, 1997. This lesson was originally written on March 29, 1997 using the software and documentation in the JDK 1.1.1 download package.

Introduction

In my experience with JDK 1.1, one of the most difficult aspects of understanding the new *Delegation Event Model* has to do with the ability to create synthetic events and post them to the **SystemEventQueue** for delivery to some specified component. The concept is not difficult. However, as of the original date of this writing (March 29, 1997) the available documentation is very meager. I have spent many hours working to understand how to accomplish this..

I believe that I have finally come to an understanding of how this process works, and that is the topic of this lesson.

Before embarking on the lesson however, let me express sincere gratitude to **Roedy Green** of Canadian Mind Products, **<http://oberon.ark.com/~roedy/>** for serving as an advisor and sounding board near the end of the learning process and for helping me to sort this all out. You should visit his site often. You will find much that is useful there including an enormous glossary explaining many aspects of Java.

Key Ingredients for Posting Events

The key ingredients for creating and posting event objects to the **SystemEventQueue** are shown below.

While the approach to any particular programming problem may involve several key ingredients, insofar as the creation and posting of the events is concerned, there is only one key ingredient:

You must define and invoke a method that will execute statements similar to the following. Note that this is all one statement.

```
Toolkit.getDefaultToolkit().
    getSystemEventQueue().
    postEvent(new MouseEvent(myCustomComp,
                             MouseEvent.MOUSE_CLICKED,
                             0,0,-1,-1,2,false));
```

We will discuss this statement in more detail later in this lesson. However, as you can see, working from the inside out, it contains the following components:

- Instantiation of a new event object of the desired event type (MouseEvent in this case).
- Posting that object to the **SystemEventQueue** that is returned by
- Invoking the **getSystemEventQueue()** method which is a method of the default toolkit returned by
- Invoking **getDefaultToolkit()** which is a static method of the **Toolkit** class.

Sample Program

This program was designed to be compiled and executed under JDK 1.1.1. It demonstrates the use of the **postEvent()** method to post events to the **SystemEventQueue**. The program also continues illustrating the creation of new component types which extend existing types. This topic was discussed in detail in an earlier set of lecture notes.

The ability to post events to the **SystemEventQueue** is used in this sample to intercept *key* events and convert them into *mouse* events.

The main **GUI** object is instantiated from a class that extends **Frame**.

A custom component class named **CustomComponent** is defined which *extends* the **Label** class. Objects of this class are capable of responding to *mouse* events and *key* event. An object of this class is added to the main **Frame** object.

Extending the **Label** class makes it possible to override the **processMouseEvent()** method for **Label** objects.

An overridden **processMouseEvent()** method is provided for the **CustomComponent** class.

Mouse events are *enabled* on objects of the class so that any **MouseEvent** on the object will be delivered to the **processMouseEvent()** method. As is always required in situations like this, the overridden **processMouseEvent()** method passes the **MouseEvent** object to the method of the same name in the *superclass* before it terminates.

Physical mouse clicks on the **CustomComponent** object are delivered to the **processMouseEvent()** method where information about the events is displayed on the screen.

KeyEvents are trapped by a **KeyListener** object. Whenever a key event is trapped, a synthetic **MouseEvent** object is created and posted to the system **EventQueue**.

The **KeyListener** object creates synthetic mouse events according to the following JDK 1.1.1 definition.

```
public MouseEvent(Component source,
                  int id,
                  long when,
                  int modifiers,
                  int x,
                  int y,
                  int clickCount,
                  boolean popupTrigger)

   Constructs a MouseEvent object with the specified
   source component, type, modifiers, coordinates, and
   click count.

   Parameters:
        source - the object where the event originated
```

In this case, the *"source"* parameter is a reference to the **CustomComponent** object. Values of -1 for *x* and *y* are provided to make the object easy to recognize when it emerges later in the **processMouseEvent()** method. (True click events won't have negative coordinate values.)

Arbitrary values are provided for the *when*, *modifiers*, *clickCount*, and *popupTrigger* parameters.

An arbitrary values was not assigned to the *id* parameter. It is critical that in constructing the **MouseEvent** object, the value of the *id* field match one of the following symbolic constants defined in the **MouseEvent** class. (It may be satisfactory to simply assure that the value is within the range of values for mouse events. There is currently no documentation on this and I haven't tested it.)

- MOUSE_CLICKED The mouse clicked event type.
- MOUSE_DRAGGED The mouse dragged event type.
- MOUSE_ENTERED The mouse entered event type.

- MOUSE_EXITED The mouse exited event type.
- MOUSE_FIRST Marks the first integer id for the range of mouse event ids.
- MOUSE_LAST Marks the last integer id for the range of mouse event ids.
- MOUSE_MOVED The mouse moved event type.
- MOUSE_PRESSED The mouse pressed event type.
- MOUSE_RELEASED The mouse released event type.

If the value of the *id* field doesn't match one of these values (or possibly fall within the allowable range), the system will not deliver the object to its intended receiver. There is no warning and an exception is not thrown. The event simply isn't delivered, (somewhat reminiscent of C and C++).

When the program starts, a **Frame** object appears on the screen filled by the **CustomComponent** object.

Typical program output while the mouse is moved around and clicked on the **CustomComponent** object and when the *x* and *y* keys are pressed is shown below. (Note the *x* and *y* coordinates of -1 for the synthetic mouse events generated inside the **KeyEventListener** object.) Note that manual line breaks were inserted to force this material to fit on the printed page.

```
In processMouseEvent in CustomComponent.
  ID = 504 java.awt.Point[x=156,y=70]
In processMouseEvent in CustomComponent.
  ID = 505 java.awt.Point[x=0,y=32]
In processMouseEvent in CustomComponent.
  ID = 504 java.awt.Point[x=4,y=9]
In processMouseEvent in CustomComponent.
  ID = 501 java.awt.Point[x=25,y=13]
In processMouseEvent in CustomComponent.
  ID = 502 java.awt.Point[x=25,y=13]
In processMouseEvent in CustomComponent.
  ID = 500 java.awt.Point[x=25,y=13]
In keyPressed() method, the key pressed was x
In processMouseEvent in CustomComponent.
  ID = 500 java.awt.Point[x=-1,y=-1]
In keyPressed() method, the key pressed was y
In processMouseEvent in CustomComponent.
  ID = 500 java.awt.Point[x=-1,y=-1]
In processMouseEvent in CustomComponent.
  ID = 505 java.awt.Point[x=105,y=96]
```

The program was tested using JDK 1.1.1 and Win95.

As an interesting side note, this program can also be used to demonstrate that the coordinate offset error that existed in JDK 1.1 (at least for the Win95 version) has been resolved in JDK 1.1.1.

Interesting Code Fragments

Much of the material in this program has been included and discussed in sample programs in earlier lessons. Those portions of the code won't be highlighted here.

The first interesting code fragment consists of some selected statements from the constructor that have to do with instantiating an object of the **CustomComponent** type and registering a **KeyListener** object on that component.

It is the code in the **KeyListener** object that we will see later that traps the **KeyEvent** objects, creates synthetic **MouseEvent** objects, and posts those **MouseEvent** objects for delivery to the **CustomComponent** object. Thus, this program might be thought of as one that intercepts **KeyEvent** objects and converts them into **MouseEvent** objects.

```
    CustomComponent myCustomComp = new CustomComponent();
    this.add(myCustomComp);
    ...
    myCustomComp.addKeyListener(
                         new MyKeyListener(myCustomComp));
```

The next interesting code fragment is the statement inside the **KeyListener** class that creates the synthetic **MouseEvent** object and posts it to the **SystemEventQueue**. As you can see, the syntax of this statement is rather involved.

Earlier in this lesson, we provided a definition of each of the parameters in the **MouseEvent** constructor so you should be able to pick them out. For example, the first parameter is **myCustomComp** which is a reference to the custom component. This is the parameter that was referred to as the *source* in the previous description. You should also recognize the second parameter as **MouseEvent.MOUSE_CLICKED** from the allowable list of *id* values presented earlier.

```
Toolkit.getDefaultToolkit().
     getSystemEventQueue().
     postEvent(new MouseEvent(myCustomComp,
                              MouseEvent.MOUSE_CLICKED,0,0,
                              -1,-1,2,false));
```

As mentioned earlier, the *x* and *y* coordinate values were both set to -1 to make objects of this type easy to recognize when they emerge in the **processMouseEvent()** method. (The real click events won't have negative coordinate values.)

The best (and possibly only) documentation regarding the remainder of this syntax can be found in the JDK 1.1.1 AWT Enhancements Design Specification under Delegation Event Model in the section entitled The Event Queue. You can download that document from JavaSoft or read it online.

That section (which is all of the documentation that I have been able to locate on this topic) is repeated below for your enlightenment.

The Event Queue

Another feature of the 1.1 event model is the addition of an event queue class:

```
java.awt.EventQueue
```

This class provides a number of public instance methods to manipulate the queue:

```
public synchronized void postEvent(AWTEvent e)

public synchronized AWTEvent getNextEvent()

public synchronized AWTEvent peekEvent()

public synchronized AWTEvent peekEvent(int eventID)
```

Programs can actually use this class to instantiate their own event queue instances for the purpose of asynchronously posting events. The EventQueue class automatically instantiates an internal thread for dispatching the events appropriately.

In the default JDK implementation, all events generated on components are first posted to a special "system" EventQueue instance before being dispatched to their target component.

The Toolkit class provides a method to access the handle of the system EventQueue instance:

```
public final EventQueue getSystemEventQueue()
```

It would obviously be a security problem to allow untrusted applets to freely manipulate the system event queue, therefore the getSystemEventQueue() method is protected by a SecurityManager check which disallows applets direct access to the system queue. We realize that applets would also like access to an event queue which is scoped to their own containment hierarchies and we are working on an architecture to allow that for a follow-on release.

Hopefully the folks at JavaSoft won't be too upset about us lifting this information verbatim from their specification.

The next interesting code fragment is a statement contained in the constructor for the **CustomComponent** class. We have seen this statement before. It is repeated here for emphasis. This statement is required in order for the overridden **processMouseEvent()** method to be invoked whenever a mouse event is delivered to an object of the **CustomComponent** class.

```
    enableEvents(AWTEvent.MOUSE_EVENT_MASK );
```

The final interesting code fragment is the overridden **processMouseEvent()** method that announces that the method has been invoked and displays the *id* value and the coordinate values from the object.

Note once again the call to **super.processMouseEvent()** at the end of the method. If you forget to do this, the outcome will be indeterminate, but probably not good.

```
  public void processMouseEvent(MouseEvent e) {
    System.out.println(
      "In processMouseEvent in CustomComponent. ID = "
                    + e.getID() + " " + e.getPoint());
    //ALWAYS DO THIS IF YOU OVERRIDE the method.
    super.processMouseEvent(e);
  }//end processMouseEvent
```

All that's left is a completely standard **WindowListener** class that we have seen many times before.

These code fragments have illustrated the the essential ingredients of creating and posting synthetic events. A complete listing of the program follows 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 Event30.java Copyright 1997, R.G.Baldwin
Reformatted on 10/4/97.
This program was designed to be compiled and executed under
JDK 1.1.1.  The reformatted version was tested
under JDK 1.1.3

This program demonstrates the use of the postEvent() method
to post events to the system EventQueue.

These features are used to intercept key events and convert
them into mouse events.

The main GUI object is of a class that extends Frame.

A custom component class named CustomComponent is defined.
Objects of this class are capable of responding to mouse
events and key events.  An object of this class is added
to the main Frame object.

An overridden processMouseEvent() method is provided for
the CustomComponent class. Mouse events are enabled on
objects of the class so that any mouse event on the object
will be delivered to the processMouseEvent() method.  As is
always required, the overridden processMouseEvent() method
passes the object to the method of the same name in the
superclass before it terminates.

Physical mouse clicks on the CustomComponent object are
delivered to the processMouseEvent() method where
```

```
information about the events is displayed on the screen.

Key events are trapped by a KeyListener object.  Whenever
a key event is trapped, a synthetic MouseEvent object is
created and posted to the system EventQueue.

The KeyListener object creates synthetic mouse events
according to the following JDK 1.1.1 definition.
-------------------------------------------
  public MouseEvent(Component source,
                    int id,
                    long when,
                    int modifiers,
                    int x,
                    int y,
                    int clickCount,
                    boolean popupTrigger)

     Constructs a MouseEvent object with the specified
       source component, type, modifiers, coordinates, and
       click count.

     Parameters:
          source - the object where the event originated
-------------------------------------------
In this case, the "source" parameter is a reference to the
CustomComponent object. Values of -1 for x and y are
provided to make the object easy to recognize when it
emerges in the processEvent()method. Arbitrary values are
provided for the when, modifiers, clickCount, and
popupTrigger parameters.

Arbitrary values were not assigned to the id parameter.  It
is absolutely critical that in constructing the mouse event
object, the value of the id field match one of the
following symbolic constants defined in the MouseEvent
class:

  MOUSE_CLICKED
     The mouse clicked event type.
  MOUSE_DRAGGED
     The mouse dragged event type.
  MOUSE_ENTERED
     The mouse entered event type.
  MOUSE_EXITED
     The mouse exited event type.
  MOUSE_FIRST
     Marks the first integer id for the range of
       mouse event ids.
  MOUSE_LAST
     Marks the last integer id for the range of
       mouse event ids.
  MOUSE_MOVED
     The mouse moved event type.
  MOUSE_PRESSED
     The mouse pressed event type.
```

```
  MOUSE_RELEASED
     The mouse released event type.

If the value of the id field doesn't match one of these
values, the system will not deliver the object to its
intended receiver.

Typical program output while the mouse is moved around and
clicked on the component and while the x and y keys are
pressed is shown below.  Note the x and y coordinates of -1
for the synthetic mouse events generated inside the
KeyEvent Listener object.

Note also that line breaks were manually inserted in this
replica of the output to cause the material to fit easily
on the width of the page.

In processMouseEvent in CustomComponent.
  ID = 504 java.awt.Point[x=156,y=70]
In processMouseEvent in CustomComponent.
  ID = 505 java.awt.Point[x=0,y=32]
In processMouseEvent in CustomComponent.
  ID = 504 java.awt.Point[x=4,y=9]
In processMouseEvent in CustomComponent.
  ID = 501 java.awt.Point[x=25,y=13]
In processMouseEvent in CustomComponent.
  ID = 502 java.awt.Point[x=25,y=13]
In processMouseEvent in CustomComponent.
  ID = 500 java.awt.Point[x=25,y=13]
In keyPressed() method, the key pressed was x
In processMouseEvent in CustomComponent.
  ID = 500 java.awt.Point[x=-1,y=-1]
In keyPressed() method, the key pressed was y
In processMouseEvent in CustomComponent.
  ID = 500 java.awt.Point[x=-1,y=-1]
In processMouseEvent in CustomComponent.
  ID = 505 java.awt.Point[x=105,y=96]

The program was originally tested using JDK 1.1.1 and Win95
and the reformatted version was tested using JDK 1.1.3.
*/

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

//=========================================================
public class Event30 extends Frame{
  public static void main(String[] args){
    Event30 displayWindow = new Event30();
  }//end main
  //---------------------------------------------------------

  public Event30(){//constructor
    setTitle("Copyright 1997, R.G.Baldwin");
    CustomComponent myCustomComp = new CustomComponent();
    this.add(myCustomComp);
```

```
    setSize(250,100);//set frame size
    setVisible(true);//display the frame

    //The following KeyListener object converts key events
    // to mouse events.
    myCustomComp.addKeyListener(
                         new MyKeyListener(myCustomComp));

    //terminate when Frame is closed
    this.addWindowListener(new Terminate());
  }//end constructor
}//end class Event30
//=========================================================

//This class listens for key events on a custom components.
// Whenever a key event is trapped, code in the overridden
// keyPressed() method displays the character (some keys
// don't generate characters that can be displayed). Then
// it creates a synthetic MouseEvent object and posts it in
// the system EventQueue to be delivered to the same
// custom component.  Thus, Listener objects of this type
// convert key events to mouse events.
class MyKeyListener extends KeyAdapter{
  //reference to the custom component
  CustomComponent myCustomComp;
  //---------------------------------------------------

  //constructor
  MyKeyListener(CustomComponent inCustomComponent){
    //save reference to custom component
    myCustomComp = inCustomComponent;
  }//end constructor
  //---------------------------------------------------

  public void keyPressed(KeyEvent e){//overridden method
    System.out.println(
            "In keyPressed() method, the key pressed was "
            + e.getKeyChar());

    //Note, the id parameter in the construction of the
    // following MouseEvent object must be a valid
    // MouseEvent id.  This event is constructed with
    // x and y coordinate values of -1 to make the event
    // easily identifiable. Note the reference to the
    // custom component as the first parameter to the
    // constructor.  That is where the object will be
    // delivered.  Note also that the following several
    // lines of code comprise a single statement.
    Toolkit.getDefaultToolkit().
        getSystemEventQueue().
        postEvent(new MouseEvent(myCustomComp,
                                 MouseEvent.MOUSE_CLICKED,
                                 0,0,-1,-1,2,false));
  }//end overridden keyPressed() method
}//end MyKeyListener
//=========================================================
```

```
//This class defines a custom component created by
// extending Label.  It can respond to key events if an
// appropriate KeyListener object is registered on its
// behalf.  It overrides processMouseEvent() for the
// purpose of capturing and displaying the MouseEvent
// objects created and posted by the KeyListener object
// with a reference to an object of this type as the
// first parameter in the MouseEvent constructor.
class CustomComponent extends Label{
  CustomComponent(){//constructor
    this.setText("Custom Component");
    //The following statement is required to cause the
    // processMouseEvent() method to be invoked whenever a
    // mouse event is queued for an object of this class.
    enableEvents(AWTEvent.MOUSE_EVENT_MASK );
  }//end constructor

//-----------------------------------------------------------
  public void processMouseEvent(MouseEvent e) {
    //Announce that the method has been invoked and display
    // the ID and coordinate values of the MouseEvent
    // object passed in as a parameter.
    System.out.println(
           "In processMouseEvent in CustomComponent. ID = "
           + e.getID() + " " + e.getPoint());

    //ALWAYS DO THIS IF YOU OVERRIDE the
    // processMouseEvent() method.
    super.processMouseEvent(e);
  }//end processMouseEvent
}//end class CustomComponent
//===========================================================

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
//===========================================================
```

A Timer Program

Now lets examine a practical example of the use of the system **Event Queue**. This is a simplified version of a program that I wrote for one of my consulting clients early in 1997.

The actual software for which I developed this program was used with a public terminal. The user of the terminal could work through a series of dialog boxes to obtain information pertinent to the use of the terminal. A **Quit** button was provided on each dialog box so that when the user was finished, or at any time along the way to completion, the user could click the **Quit** button which would return the system to its *idle* state ready for the next user.

Since this was a public terminal, there was no guarantee that a user would bother to click the **Quit** button. Sometimes, the user would simply abandon the system in the middle of one of the dialog boxes. For that reason, it was necessary for the system to automatically return to the *idle* state after a certain amount of time had elapsed with no user activity.

The version of the program shown in the following example is one where the user was expected to enter some information in **TextField** objects. As long as the user entered at least one character within a ten-second interval, the system was considered to be active (the actual time used was longer than ten seconds, but it has been shortened in this example for illustration purposes).

If the user failed to enter at least one character during a ten-second interval, the system was considered to have been abandoned. In that case, a **Timer** thread created a synthetic **actionPerformed()** event, attributed it to the **Quit** button, and placed it in the system **Event Queue**. This had the same effect as if the user had actually clicked the **Quit** button.

Quite a lot of processing was performed at this point in the actual system as it prepared itself for the next user. In this simplified example, the **actionPerformed()** event on the **Quit** button simply causes some text to be displayed and then causes the program to terminate. The actual source of the event; the **Quit** button or the **Timer** is displayed, is also displayed.

Although quite simple, this example is adequate to illustrate the importance of being able to generate specific kinds of events and attribute them to specific components completely under program control. Stated differently, this simple program illustrates how to write program code that can simulate the physical action of a user.

The operational aspects of the program are described in the comments.

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

This application illustrates the use of the system event
queue for the posting of synthetic events.

A Frame object appears on the screen containing a TextField
object and a Quit Button object.

If the user clicks the Quit button, the program terminates
by invoking the ActionPerformed() method on the Button.

If the user enters text into the TextField object, a ten-
second timer is reset on each keystroke and the ten-second
interval starts over.

If the user fails to enter text into the TextField object
for ten seconds, the ten-second elapsed time programmed
into the timer expires, causing the timer object to
create a synthetic ActionPerformed event, attribute it to
the Quit button, and place it in the system event queue.
```

Because the timer creates a synthetic ActionPerformed() event and attributes it to the Quit button, the end result of a timeout is exactly the same as if the user were to click the Quit button. Either action causes the same ActionPerformed() method to be invoked. This makes it possible to place code in the ActionPerformed method to do any cleanup or other useful work that might be appropriate before the program actually terminates.

This is a case of program code simulating the action of a human user by creating a synthetic click event on a button.

The general purpose timer class used in this program is designed to accept a timeout interval and a component as parameters. When the timeout interval expires, a synthetic ActionPerformed() event is generated. That event is attributed to the component that is passed in as a parameter.

This is accomplished using a separate thread object of type Timer. When the Timer object is instantiated, a reference to the Quit button and the ten-second timeout interval in milliseconds are passed in as parameters.

When the Timer object is instantiated, it goes to sleep for the prescribed time interval. It will wake up under either of two conditions:

One condition is that it is interrupted by code in another thread. In this case, it throws an InterruptedException object which is caught and processed to implement the reset logic of the Timer class.

The other case is that the elapsed time expires. In this case, it doesn't throw an InterruptedException object and the Timer object is not reset.

If it's sleep is interrupted, it resets and goes back to sleep for the prescribed period.

If it wakes up without being interrupted, it generates a synthetic actionPerformed() event on the component that is passed in as a parameter and places the synthetic event in the system event queue. The synthetic event contains a command that can be extracted in the ActionPerformed() method to determine that the event was actually generated by the timer if such determination is needed. In this simple example, the command is simply displayed on the screen to indicate whether the actionPerformed() method was invoked by a click on the button, or was invoked by the Timer object.

Placing the synthetic event in the system event queue causes the actionPerformed() method to be invoked on the specified component just as though the user had clicked the Quit button.

```
A KeyListener object is registered for the text field and
an ActionListener object is registered for the button.

The only function of the KeyListener object in this simple
example is to interrupt the Timer object and cause it
to wake up, reset the time interval, and go back to sleep.

The ActionListener object is used to simulate cleanup and
shutdown whenever the user clicks the Quit button or allows
the system to time out.

A WindowListener object is also instantiated to support the
close button on the Frame object just in case it is needed.
*/
//========================================================

import  java.applet.Applet;
import  java.awt.*;
import  java.awt.event.*;

//========================================================
class TimerTest{//controlling class
  static public void main(String[] args){
    new PresentationGUI();
  }//end main
}//end class TimerTest

//========================================================
//Note that this class implements the listener interfaces
// which means that an object of this class is a listener
// object.  An object of this class is added to the
// lists of listener objects later using the keyword this.
class PresentationGUI
                    implements ActionListener, KeyListener{
  Frame myFrame;  //references to various objects
  Button quitButton;
  TextField inputField;
  Thread myTimer;
  //--------------------------------------------------------

  PresentationGUI(){//constructor
    myFrame = new Frame("Copyright 1997, R.G.Baldwin");
    myFrame.setLayout(new FlowLayout());
    myFrame.setSize(400,100);

    //Add the TextField to the Frame
    myFrame.add(inputField = new TextField(10));

    //Add a KeyListener object to the TextField object
    //See the above note regarding use of the keyword this
    // as a parameter to the add...Listener method.  The
    // code in the KeyListener object causes the timer to
    // be reset on each keystroke on the TextField object.
    inputField.addKeyListener(this);
```

```
    //Add the Button to the Frame
    quitButton = new Button("Quit");
    //The following ActionCommand is displayed by the
    // ActionPerformed method if the user clicks the
    // quitButton.
    quitButton.setActionCommand("Quit Button was Clicked");

    myFrame.add(quitButton);

    //Add an ActionListener object to the Button.
    //Note that because this class implements the action
    // listener interface and the ActionPerformed()
    // method is defined in this class, an object of this
    // class is an ActionListener object.  Thus the keyword
    // this is registered to listen for Action events on
    // the quitButton by the following statement.
    quitButton.addActionListener(this);

    //make visible and redraw everything to proper size
    myFrame.setVisible(true);
    myFrame.validate();

    //Instantiate the timer object for a prescribed ten-
    // second delay period.  Pass a reference to the Quit
    // button so that the timer object can create a
    // synthetic actionPerformed event on that button
    // whenever it times out.
    //The timer object is instantiated to run in a
    // separate unsynchronized thread.
    myTimer = new Thread(new Timer(quitButton,10000));
    //start the Timer thread
    myTimer.start();

    //The following listener object services the close
    // button on the Frame object if needed, but it is
    // really not needed because the quitButton can be
    // used to terminate the program.
    WProc1 winProcCmd1 = new WProc1();
    myFrame.addWindowListener(winProcCmd1);
  };  // end constructor
  //-----------------------------------------------------

  //The next two empty methods are provided simply to
  // satisfy the requirement to implement all methods
  // declared in the KeyListener interface.
  public void keyPressed(KeyEvent e){}
  public void keyReleased(KeyEvent e){}

  //The following method responds to keyTyped events and
  // resets the timer on each keystroke.
  public void keyTyped(KeyEvent e){
    myTimer.interrupt();//reset the timer
  }//end method keyTyped
  //-----------------------------------------------------

  //The following actionPerformed() method is invoked
```

```
  // whenever an actionPerformed event is placed in the
  // system event queue.  In this program, that can happen
  // either by the user clicking on the quitButton or by
  // the timer object experiencing a timeout.
  public void actionPerformed(ActionEvent  e){
    //Display the ActionCommand that identifies the actual
    // source of the event:  quitButton or timer.
    System.out.println(e.getActionCommand());

    //Do something useful here prior to termination
    System.out.println(
                   "This method will be invoked if the\n" +
                    "user clicks on the Quit button, or\n" +
                    "if the system is allowed to time \n" +
                    "out.  In either case, the necessary\n"+
                    "cleanup can be performed before\n" +
                    "actually terminating the program.");

    System.exit(0);//terminate the program
  };  // end actionPerformed()
}; // end class PresentationGUI
//=========================================================

//The following listener class is used to terminate the
// program when the user closes the frame object.
class WProc1 extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    System.exit(0);
  }//end windowClosing()
}//end class WProc1
//=========================================================

/*This is a custom Timer class that generates a synthetic
actionPerformed() event on the source object passed in
as a parameter after sleeping for a specified period of
time.  The specified period of time to sleep is also passed
in as a parameter.

In the event that sleep is interrupted by another thread
invoking the interrupt() method on this thread, the timer
is reset and then goes back to sleep.

If sleep is not interrupted, a synthetic actionPerformed()
event is placed in the system event queue and attributed
to the source object passed in as a parameter.

Note that this class implements the Runnable interface, and
defines a run() method.  Therefore, it can be run in its
own thread.
*/

class Timer implements Runnable{
  Object source;//reference to the source component
  int delay;//time interval to sleep
  //-------------------------------------------------------
```

```
  Timer(Object inSource, int inDelay){//constructor
    source = inSource;//save references
    delay = inDelay;
  }//end constructor
  //-------------------------------------------------------

  //The significant functionality of all thread objects is
  // written into the run() method for the object.
  public void run(){
    boolean keepLooping = true;

    //Keep looping and resetting as long as the
    // keepLooping variable is true.  Keystrokes in the
    // TextField object have the effect of setting the
    // variable named keepLooping to true.
    while(keepLooping){//while keepLooping is true
      keepLooping = false;
      //In order to avoid exiting the loop, it is necessary
      // that sleep be interrupted which will cause the
      // keepLooping variable to be restored to true.

      try{
        Thread.currentThread().sleep(delay);
      }catch(InterruptedException e){
        //Control is transferred here when interrupt()
        // is invoked on this thread

        //Display the InterruptedException object
        System.out.println("" + e.toString());

        //Reset the timer when thread is interrupted by
        // restoring keepLooping to true and looping back
        // to the top of the loop
        keepLooping = true;
      }//end catch

      //Exit the loop if keepLooping is still false.
      // Go back to the top of the loop if keepLooping has
      // been restored to true in the catch block.
    }//end while(keepLooping)

    //Control is transferred here when the loop is
    // allowed to terminate indicating that sleep was not
    // interrupted by a keystroke during the prescribed
    // timeout period.
    //Create a synthetic actionPerformed() event on the
    // source object.  Note the continuation of the
    // following very long statement on the next several
    // lines.
    Toolkit.getDefaultToolkit().
        getSystemEventQueue().
        postEvent(
          new ActionEvent(source,
                          ActionEvent.ACTION_PERFORMED,
                          "Timeout Period Elapsed"));
  }//end run method
```

```
}//end Timer class
//=============================================================
```

Review

Q - Write a Java application that meets the specifications given below.

A - See the specifications and the solution below.

```
/*File SampProg132.java from lesson 104
Copyright 1997, R.G.Baldwin

Without viewing the solution that follows, write a Java
application that meets the following specifications.

When the program starts, a Frame object appears on the
screen with a width of about 300 pixels and a height of
about 100 pixels.

Your name must appear in the banner at the top of the Frame
object.

A TextField object about 10 characters wide appears in the
center of the Frame object.  When the program starts, the
TextField object has the focus.

If you click in the TextField object, a mouseClicked()
method in a MouseListener object is invoked and announces
that it has been invoked. It also displays the name
extracted from a reference to the TextField object. It also
displays the identification of the MouseListener object
which shows the name of the MouseListener class and the
address of the object.

If you press a key to enter a character into the TextField
object, a keyPressed() method in a KeyListener object is
invoked.  This method announces that it has been invoked
and displays the character. It also displays the name
extracted from a reference to the TextField object. It also
displays the identification of the KeyListener object
which shows the name of the KeyListener class and the
address of the object.

In addition, when you press a key to enter a character in
the TextField object, this causes the same mouseClicked()
method in the same MouseListener object to be invoked as
described earlier, and to behave in the same way that it
behaves when you click the mouse in the TextField object.

When you click the close button in the upper right-hand
corner of the Frame object, the program terminates and
control is returned to the operating system.
```

```
This program was tested with JDK 1.1.3 under Win95.

The output produced by first clicking the mouse in the
TextField object and then entering the character "a" is
shown below.  You probably won't get the same addresses
following the "@".  However, it is necessary that each time
the mouseClicked() method is invoked, the same address is
displayed.

In mouseClicked() method
textfield0
MyMouseListener@1ccbec
In keyPressed() method, the key pressed was a
textfield0
MyKeyListener@1ccbdb
In mouseClicked() method
textfield0
MyMouseListener@1ccbec

*/

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

//============================================================
public class SampProg132 extends Frame{
  public static void main(String[] args){
    SampProg132 displayWindow = new SampProg132();
  }//end main
  //----------------------------------------------------------

  public SampProg132(){//constructor
    this.setTitle("Copyright 1997, R.G.Baldwin");
    this.setLayout(new FlowLayout());
    TextField myTextField = new TextField(10);

    this.add(myTextField);
    setSize(300,100);//set frame size
    setVisible(true);//display the frame

    myTextField.addKeyListener(
                          new MyKeyListener(myTextField));

    myTextField.addMouseListener(
                        new MyMouseListener(myTextField));

    //terminate when Frame is closed
    this.addWindowListener(new Terminate());
  }//end constructor
}//end class SampProg132
//============================================================

class MyMouseListener extends MouseAdapter{
  TextField myTextFieldRef;
  //----------------------------------------------------------
```

```
  //constructor
  MyMouseListener(TextField inTextField){
    //save reference to TextField component
    myTextFieldRef = inTextField;
  }//end constructor
  //-------------------------------------------------------

  public void mouseClicked(MouseEvent e){
    System.out.println("In mouseClicked() method");
    System.out.println(myTextFieldRef.getName());
    System.out.println(this);
  }//end overridden mouseClicked() method
}//end MyMouseListener

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

//This class listens for key events on a TextField.
// Whenever a key event is trapped, code in the overridden
// keyPressed() method displays the character (some keys
// don't generate characters that can be displayed). Then
// it creates a synthetic MouseEvent object and posts it in
// the system EventQueue to be delivered to the TextField
// object.
class MyKeyListener extends KeyAdapter{
  //reference to the TextField object
  TextField myTextFieldRef;
  //-------------------------------------------------------

  //constructor
  MyKeyListener(TextField inTextField){
    //save reference to TextField object
    myTextFieldRef = inTextField;
  }//end constructor
  //-------------------------------------------------------

  public void keyPressed(KeyEvent e){//overridden method
    System.out.println(
            "In keyPressed() method, the key pressed was "
            + e.getKeyChar());
    System.out.println(myTextFieldRef.getName());
    System.out.println(this);

    Toolkit.getDefaultToolkit().
        getSystemEventQueue().
        postEvent(new MouseEvent(myTextFieldRef,
                                 MouseEvent.MOUSE_CLICKED,
                                 0,0,-1,-1,2,false));
  }//end overridden keyPressed() method
}//end MyKeyListener
//=========================================================

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-