The KeyEventPostProcessor in Java (Capturing Keyboard Strokes in Java)

The processing of a key event by the focus owner is not necessarily the end of the road for that event. Learn how to cause a chain of post processors to spring into action and to perform additional processing on the event before it finally dies.

Published: January 25, 2005
By Richard G. Baldwin

Java Programming, Notes # 1856

Preface

The focus subsystem

When Java 2 SDK, Standard Edition Version 1.4 was released, it contained a large number of new features. This included many changes and additions to the focus subsystem, some of which were incompatible with earlier versions.

This lesson is part of a series of lessons designed to teach you how to use some of those features of the focus subsystem.

The first lesson in the series was entitled Focus Traversal Policies in Java Version 1.4. The previous lesson was entitled The KeyEventDispatcher in Java.

Previous topics

Previous lessons in this series have taught you how to use several features of the new focus subsystem, including:

- Defining new focus traversal keys
- How to control focusability at runtime.
- The ability to query for the currently focused Component.
- The default Focus Traversal Policy.

- How to establish a focus traversal policy and modify it at runtime.
- How to control the focus programmatically.
- Opposite components.
- The KeyEventDispatcher.

This lesson will show you how to use the **KeyEventPostProcessor** interface.

Builds on previous lesson

The sample program that I will explain in this lesson is an upgrade of the program presented in the previous lesson entitled The KeyEventDispatcher in Java. Thus, the material in this lesson builds upon the material that I presented in that lesson. As a result, much of the material discussed early in this lesson will be very brief, simply reminding you of what you learned in the previous lesson.

It is strongly recommended that you study and understand the material in the previous lesson entitled The KeyEventDispatcher in Java before embarking on the material in this lesson.

What do we mean by focus?

Among all of the applications showing on the desktop at any point in time, only one will respond to the keyboard.

If that application is a Java application, only one component within that application's graphical user interface *(GUI)* will respond to the keyboard. That is the component that has the *focus* at that point in time.

A Java component that has the focus also has the ability to fire **KeyEvents** when it responds to the keyboard.

KeyEventDispatcher and KeyEventPostProcessor

The program that I will explain in this lesson makes heavy use of both the **KeyEventDispatcher** interface and the **KeyEventPostProcessor** interface.

The previous lesson provided detailed information about the **KeyEventDispatcher** interface, so I won't repeat that information here. Rather, I will concentrate on the **KeyEventPostProcessor** interface in this lesson.

What does Sun have to say?

Before getting into the technical details, I will provide some quotations from Sun that describe both the **KeyEventDispatcher** interface and **KeyEventPostProcessor** interface.

*"A **KeyEventDispatcher** is a lightweight interface that allows client code to pre-listen to all **KeyEvents** ...*

*... allowing the **KeyEventDispatcher** to retarget the event, consume it, dispatch it itself, or make other changes.*

*... if a **KeyEventDispatcher** reports that it dispatched the KeyEvent, ... the **KeyboardFocusManager** will take no further action with regard to the KeyEvent.*

*Client-code may also post-listen to **KeyEvents** ...using the **KeyEventPostProcessor** interface.*

***KeyEventPostProcessors** registered with the current **KeyboardFocusManager** will receive **KeyEvents** after the **KeyEvents** have been dispatched to and handled by the focus owner."*

In layman's terms ...

A **KeyEventDispatcher** object has an opportunity to process all **KeyEvents** before they are dispatched to the components that fired them, and can prevent those events from being dispatched to the components that fired them.

A **KeyEventPostProcessor** object has an opportunity to process all **KeyEvents** after they have been processed by **KeyEventDispatchers** or by the components that fired the events.

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.

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. 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 they are difficult to locate there. You will find a consolidated index at www.DickBaldwin.com.

Preview

In this lesson, I will show you how to use a **KeyEventPostProcessor** to process **KeyEvents** after they have been processed by **KeyEventDispatchers** or by the components that fired the events.

Using a sample program, characters typed into one text field in a GUI will appear in another text field belonging to the same GUI. Those characters will not appear in the text field into which they were typed.

In addition, a **KeyEventPostProcessor** will display information about the events on the screen.

Discussion and Sample Code

Description of the program

This lesson presents and explains a sample program named **KeyEventPostProc01**.

This program illustrates the use of a **KeyEventPostProcessor** object in addition to a pair of **KeyEventDispatcher** objects.

This program is an upgrade of the program named **KeyEventDispatch01**, which was discussed in the previous lesson. This version adds a **KeyEventPostProcessor** to process **KeyEvents** after they have been handled either by the focus owner or by a **KeyEventDispatcher** object.

The program GUI

The program causes a single **JFrame** object to appear on the screen as shown in Figure 1.

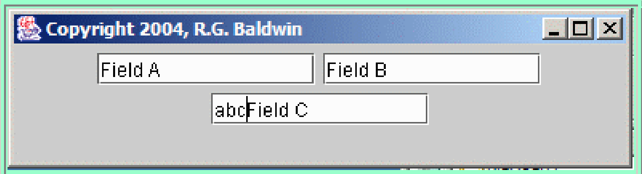


Figure 1 Program GUI.

The **JFrame** object contains three **JTextField** objects. The initial text in the three **JTextField** objects is respectively:

- Field A
- Field B
- Field C

Two KeyEventDispatcher objects

Two different **KeyEventDispatcher** objects are registered on the current **KeyboardFocusManager** in the following order:

- AlternateDispatcherA
- AlternateDispatcherB

When a KeyEvent occurs ...

When a **KeyEvent** occurs, the current **KeyboardFocusManager** delivers that event to the **AlternateDispatcherA** object for dispatching.

*(Events are delivered to multiple registered **KeyEventDispatcher** objects in the order in which they are registered.)*

If the focus owner is *Field A* ...

If the owner of the focus at that point in time *(the source of the event)* is *Field A*, the **AlternateDispatcherA** object *re-dispatches* that event to *Field C* and returns **false**.

*(Note that the program in the previous lesson returned **true** at this point.)*

Re-dispatching the event to *Field C* causes keystrokes entered into *Field A* to appear in *Field C* instead of appearing in *Field A.*

If the source of the event is not *Field A* ...

If the source of the event is not *Field A,* the **AlternateDispatcherA** object still returns **false**. The object returns **false** regardless of whether the source of the event is *Field A.*

Returning false ...

Returning **false** causes the event to be delivered to **AlternateDispatcherB** as described below, and also causes the event to be delivered to the **KeyEventPostProcessor** later.

If the source of the event is *Field B* ...

If the owner of the focus at that point in time is *Field B,* the **AlternateDispatcherB** object *re-dispatches* that event to *Field C* and returns **false**.

*(Once again, the program in the previous lesson returned **true** at this point.)*

Re-dispatching the event to *Field C* causes keystrokes entered into *Field B* to appear in *Field C* instead of appearing in *Field B.*

If the source of the event is not *Field B* ...

If the owner of the focus at that point in time is not *Field B,* the **AlternateDispatcherB** object still returns **false**.

Returning **false** causes the current **KeyboardFocusManager** to dispatch the event to the component that owns the focus. This is what normally happens in the absence of registered **KeyEventDispatcher** objects.

Returning **false** also causes the event to be delivered to the **KeyEventPostProcessor** later.

Must be *Field C*

In this program, the component that owns the focus at this point would have to be *Field C* because it has already been determined that neither *Field A* nor *Field B* owns the focus. This causes keystrokes entered into *Field C* to appear in *Field C* as normal.

All keystrokes appear in *Field C*

Thus, all keystrokes typed into any of the three fields in the GUI in Figure 1 will appear in *Field C*.

The state of the GUI

Figure 1 shows the state of the GUI after the characters a, b, and c, were entered respectively into fields A, B, and C, in that order. Note that all three characters, abc, appear in *Field C.*

The KeyEventPostProcessor

After the event has been dispatched to the focus owner or delivered to the **KeyEventDispatcher** objects, the **KeyboardFocusManager** will deliver the event to the registered **KeyEventPostProcessor** objects for final processing.

> *(There is only one registered* ***KeyEventPostProcessor*** *in this program.)*

Display the character and the event type

The **KeyEventPostProcessor** determines and displays the character that was entered into the text field and the type of the event. Three types of **KeyEvent** are possible:

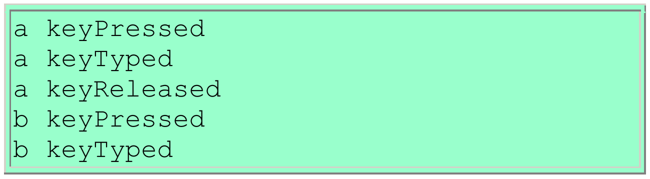
- The "key pressed" event.
- The "key released" event.
- The "key typed" event.

The character and the type of the event are displayed on the screen as each event is processed by the **KeyEventPostProcessor**.

> *(Note that typing a single character into a text field causes all three of the event types listed above to be fired.)*

The screen output

The screen output produced by entering the characters a, b, and c into fields A, B, and C respectively is shown in Figure 2.



```
a keyPressed
a keyTyped
a keyReleased
b keyPressed
b keyTyped
```

```
b keyReleased
c keyPressed
c keyTyped
c keyReleased
```

Figure 2

A chain of KeyEventPostProcessors

A **KeyboardFocusManager** can register more than one **KeyEventPostProcessors**, creating a chain of **KeyEventPostProcessors**, ending with the current **KeyboardFocusManager**.

Each **KeyEventPostProcessor** can return either **true** or **false**. If a **KeyEventPostProcessor** returns false, then the **KeyEvent** is passed to the next **KeyEventPostProcessor** in the chain.

If the **KeyEventPostProcessor** returns **true,** the **KeyEvent** is assumed to have been fully handled and the **KeyboardFocusManager** will take no further action with regard to the **KeyEvent**.

Note that the **KeyEventPostProcessor** in this program returns **true**. However, it doesn't matter whether the return value is **true** or **false** because only one **KeyEventPostProcessor** is registered on the **KeyboardFocusManager** in this program.

Will discuss the program in fragments

I will discuss this program in fragments. A complete listing of the program is provided in Listing 7 near the end of the lesson.

Listing 1 shows the class named **KeyEventPostProc01**, including the **main** method.

```
class KeyEventPostProc01{
  public static void main(String[]
args){
    new GUI();
  }//end main
}//end class KeyEventPostProc01
```

Listing 1

In this case, the **main** method simply instantiates an object of the **GUI** class. The object of the **GUI** class appears in Figure 1.

The GUI class

Listing 2 shows the beginning of the **GUI** class, which extends the **JFrame** class. Thus, a **GUI** object is also a **JFrame** object.

```
class GUI extends JFrame{
  JTextField fieldA =
                    new
JTextField("Field A",12);
  JTextField fieldB =
                    new
JTextField("Field B",12);
  JTextField fieldC =
                    new
JTextField("Field C",12);

  KeyboardFocusManager manager;

  GUI(){//constructor
    getContentPane().setLayout(new
FlowLayout());
    getContentPane().add(fieldA);
    getContentPane().add(fieldB);
    getContentPane().add(fieldC);

    setSize(390,100);
    setTitle("Copyright 2005, R.G.
Baldwin");
    setDefaultCloseOperation(

JFrame.EXIT_ON_CLOSE);
    setVisible(true);
    manager = KeyboardFocusManager.

getCurrentKeyboardFocusManager();

    manager.addKeyEventDispatcher(
                        new
AlternateDispatcherA(

fieldA,fieldC,manager));
    manager.addKeyEventDispatcher(
                        new
AlternateDispatcherB(

fieldB,fieldC,manager));
```

Listing 2

The code in Listing 2 is very similar to the code that I explained in the previous lesson, so I won't repeat that explanation here.

(Note that the code in Listing 2 ends in the middle of the constructor.)

Register a KeyEventPostProcessor

Continuing with the constructor, the statement in Listing 3 is new to this lesson.

```
    manager.addKeyEventPostProcessor(
                            new
PostProcessor());

  }//end constructor
}//end class GUI
```

Listing 3

The statement in Listing 3 instantiates a new object of the **PostProcessor** class and registers it on the **KeyboardFocusManager**.

The **PostProcessor** class implements the **KeyEventPostProcessor** interface. Therefore, an object of the **PostProcessor** class is a **KeyEventPostProcessor**.

*(I will explain the **PostProcessor** class later in this lesson.)*

End of the GUI class

Listing 3 also signals the end of the constructor and the end of the **GUI** class.

The AlternateDispatcherA class

Listing 4 shows the complete **AlternateDispatcherA** class.

```
class AlternateDispatcherA
                   implements
KeyEventDispatcher{
  JTextField fieldA;
  JTextField fieldC;
  KeyboardFocusManager manager;

  //Constructor
  AlternateDispatcherA(
                   JTextField fieldA,
                   JTextField fieldC,

KeyboardFocusManager manager){
    this.fieldA = fieldA;
    this.fieldC = fieldC;
    this.manager = manager;
  }//end constructor

  public boolean
dispatchKeyEvent(KeyEvent e){
    if(e.getSource() == fieldA){

manager.redispatchEvent(fieldC,e);
    }//end if
    return false;
  }//end dispatchKeyEvent
}//end class AlternateDispatcherA
```

Listing 4

The **AlternateDispatcherA** class implements the **KeyEventDispatcher** interface. Therefore, an object of the **AlternateDispatcherA** class is an **KeyEventDispatcher**.

> *(An object of the **AlternateDispatcherA** class was registered on the **KeyboardFocusManager** in Listing 2.)*

Similar to previous code

The code in Listing 4 is almost identical to code in a class having the same name that I explained in the previous lesson. The only difference involves some minor logic changes having to do with the return value from the method named **dispatchKeyEvent**.

> *(This version of the **dispatchKeyEvent** method always returns **false**, whereas the previous version returned **true** if the source of the event was Field A and returned **false** otherwise.)*

Therefore, I won't repeat that explanation in this lesson.

The AlternateDispatcherB class

Listing 7 near the end of the lesson also defines a class named **AlternateDispatcherB**.

Once again, the code in the **AlternateDispatcherB** class **is** almost identical to code in a class having the same name that I explained in the previous lesson. The only difference involves minor logic changes having to do with the return value from the method named **dispatchKeyEvent**.

As a result, I won't show the **AlternateDispatcherB** class as a code fragment in this lesson, and I won't repeat the previous explanation in this lesson.

The PostProcessor class

The beginning of the **PostProcessor** class and the entire method named **postProcessKeyEvent** are shown in Listing 5.

```

class PostProcessor
                implements
KeyEventPostProcessor{

  public boolean
postProcessKeyEvent(KeyEvent e){
    System.out.println(
                e.getKeyChar() + " " +

getEventType(e.getID()));
    return true;
  }//end postProcessKeyEvent
```

Listing 5

The **PostProcessor** class implements the **KeyEventPostProcessor** interface. Therefore, an object of the **PostProcessor** class is a **KeyEventPostProcessor.**

An object of the **PostProcessor** class was registered on the **KeyboardFocusManager** in Listing 3 by invoking the **addKeyEventPostProcessor** method on the **KeyboardFocusManager** and passing the **PostProcessor** object as a parameter.

The addKeyEventPostProcessor method

Here is some of what Sun has to say about the behavior of the **addKeyEventPostProcessor** method and the subsequent behavior of the **KeyboardFocusManager** with respect to registered **KeyEventPostProcessors**.

> *"This method adds a **KeyEventPostProcessor** to this **KeyboardFocusManager's** post- processor chain.*
>
> *After a **KeyEvent** has been dispatched to and handled by its target, **KeyboardFocusManager** will request that each **KeyEventPostProcessor** perform any necessary post-processing as part of the **KeyEvent's** final resolution.*
>
> ***KeyEventPostProcessors** will be notified in the order in which they were added; the current **KeyboardFocusManager** will be notified last.*
>
> *Notifications will halt as soon as one KeyEventPostProcessor returns **true** from its **postProcessKeyEvent** method.*
>
> *There is no limit to the total number of **KeyEventPostProcessors** that can be added, nor to the number of times that a particular **KeyEventPostProcessor** instance can be added.*
>
> *If a null post-processor is specified, no action is taken and no exception is thrown."*

The KeyEventPostProcessor interface

The **PostProcessor** class implements the **KeyEventPostProcessor** interface. Here is some of what Sun has to say about that interface.

*"A **KeyEventPostProcessor** cooperates with the current **KeyboardFocusManager** in the final resolution of all unconsumed **KeyEvents**.*

***KeyEventPostProcessors** registered with the current **KeyboardFocusManager** will receive **KeyEvents** after the **KeyEvents** have been dispatched to and handled by their targets."*

The **KeyEventPostProcessor** interface declares a single method named **postProcessKeyEvent**, which must be defined by classes that implement the interface.

The postProcessKeyEvent method

Here is some of what Sun has to say about the **postProcessKeyEvent** method.

*"This method is called by the current **KeyboardFocusManager**, requesting that this **KeyEventPostProcessor** perform any necessary post-processing which should be part of the **KeyEvent's** final resolution.*

*At the time this method is invoked, typically the **KeyEvent** has already been dispatched to and handled by its target. ...*

*Note that if a **KeyEventPostProcessor** wishes to dispatch the **KeyEvent**, it must use **redispatchEvent** to prevent the AWT from recursively requesting that this **KeyEventPostProcessor** perform post-processing of the event again.*

*If an implementation of this method returns **false**, then the **KeyEvent** is passed to the next **KeyEventPostProcessor** in the chain, ending with the current **KeyboardFocusManager**.*

*If an implementation returns **true**, the **KeyEvent** is assumed to have been fully handled (although this need not be the case), and the AWT will take no further action with regard to the **KeyEvent**."*

The code for the postProcessKeyEvent method

Now refer back to the definition of the **postProcessKeyEvent** method in Listing 5.

*(Note that the method receives a reference to the **KeyEvent** object as an incoming parameter.)*

The code invokes the following two methods on the **KeyEvent** object:

- getKeyChar
- getID

Behavior of these methods

The **getKeyChar** method returns the character associated with the key in the **KeyEvent** object.

The **getID** method is inherited from the **AWTEvent** class. It returns the event type as a numeric value. For a **KeyEvent**, **t**he event type is returned as a value of type **int**, which corresponds to one of the following three constants defined in the **KeyEvent** class:

- KEY_PRESSED
- KEY_RELEASED
- KEY_TYPED

Event type as a descriptive String

The purpose of the code in Listing 5 is to display the character and the event type for all **KeyEvents**.

Because the **getID** method returns the type as a numeric value of type **int**, it is useful to convert that type to a descriptive string before displaying it. This is accomplished by the method named **getEventType.**

The getEventType method

The **getEventType** method is shown in Listing 6.

```
String getEventType(int ID){
  if(ID == KeyEvent.KEY_PRESSED){
    return "keyPressed";
  }else if(ID ==
KeyEvent.KEY_RELEASED){
    return "keyReleased";
  }else if(ID == KeyEvent.KEY_TYPED){
    return "keyTyped";
  }else{
    return "Unknown event type";
  }//end else
}//end getEventType
```

Listing 6

This method simply

- Receives the numeric type identifier as an incoming parameter.
- Compares the value of the parameter with the values of the three constants listed earlier.
- Returns a **String** that describes the event type.

  (As an aside, the strings returned by this method are actually the names of the corresponding methods declared in the ***KeyListener*** *interface. This is the interface that is implemented by code designed to handle* ***KeyEvents*** *in the*

JavaBeans event model. I have published numerous tutorials on this topic on my web site.)

The screen output

Now refer back to Figure 2, which shows the screen output produced by entering the characters a, b, and c into fields A, B, and C respectively in the GUI shown in Figure 1.

(Note that all three event types are fired each time a single character is typed into one of the text fields.)

Run the Program

I encourage you to copy, compile, and run the program provided in this lesson. Experiment with it, making changes and observing the results of your changes.

Summary

I showed you how to use a **KeyEventPostProcessor** object to process **KeyEvents** after they have been processed by **KeyEventDispatchers** or by the components that fired the events.

What's Next?

Future lessons will discuss other features of the focus subsystem including the following:

- FocusEvent and WindowEvent
- Event delivery
- Temporary focus events
- Focus and PropertyChangeListener
- Focus and VetoableChangeListener

Complete Program Listing

A complete listing of the program discussed in this lesson is provided below.

```
/*File KeyEventPostProc01.java
Copyright 2005,R.G.Baldwin
Revised 07/27/04

This program illustrates the use of a
KeyEventPostProcessor in addition to an
alternate KeyEventDispatcher.

This is an upgrade to the program named
KeyEventDispatch01. This version adds a
```

```
KeyEventPostProcessor to process key events after
they have been handled either by the focus owner
or by an alternate KeyEventDispatcher object.

A JFrame object appears on the screen containing
three JTextField objects. The initial text in
the three JTextField objects is respectively:

Field A
Field B
Field C

Two alternative KeyEventDispatcher objects are
registered on the current KeyboardFocusManager
in the following order:

AlternateDispatcherA
AlternateDispatcherB

In addition a KeyEventPostProcessor object is
registered on the current KeyboardFocusManager.

When a key event occurs, the current
KeyboardFocusManager delivers that event to
AlternateDispatcherA for dispatching. If the
owner of the focus at that point in time is
Field A, the AlternateDispatcherA object
re-dispatches that event to Field C and returns
false. That causes keystrokes entered into
Field A to appear in Field C instead. Returning
false causes the event to be delivered to
AlternateDispatcherB as described below, and also
causes the event to be delivered to the
KeyEventPostProcessor later.

If the owner of the focus at that point in time
is not Field A, the AlternateDispatcherA object
returns false. This causes the current
KeyboardFocusManager to deliver the event to
AlternateDispatcherB for dispatching. If the
owner of the focus at that point in time is
Field B, the AlternateDispatcherB object
re-dispatches that event to Field C and returns
false. That causes keystrokes entered into
Field B to appear in Field C instead. Returning
false causes the event to be delivered to the
KeyEventPostProcessor later.

If the owner of the focus at that point in time
is not Field B, the AlternateDispatcherB object
returns false. This causes the current
KeyboardFocusManager to dispatch the event to
the component that owns the focus. In this
simple program, that would have to be Field C
because it has already been determined that
neither Field A nor Field B own the focus.
```

```
That causes keystrokes entered into Field C to
appear in Field C as is normally the case.

Thus, all keystrokes entered into any of the
three fields in the GUI will appear in field C.

In addition, after the event has been dispatched
to the focus owner or delivered to the alternate
KeyEventDispatcher object, the
KeyboardFocusManager will deliver the event to
the KeyEventPostProcessor object for final
processing. The KeyEventPostProcessor determines
and displays the character that was entered into
the text field and the type of the event. Three
types of event are possible:

The "key pressed" event.
The "key released" event.
The "key typed" event.

The character and the type of the event are
displayed on the screen as each event is
processed by the KeyEventPostProcessor. Note
that entering a single character into a text
field causes all three of the event types listed
above to be fired. The screen output produced
by entering the characters a, b, and c into
fields A, B, and C respectively is shown below.

a keyPressed
a keyTyped
a keyReleased
b keyPressed
b keyTyped
b keyReleased
c keyPressed
c keyTyped
c keyReleased

Note that the KeyEventPostProcessor returns true,
but in this program, it doesn't matter whether
the return value is true or false. This is
because this program only has one
KeyEventPostProcessor object registered on the
KeyboardFocusManager. If a KeyEventPostProcessor
object returns false, then the KeyEvent is passed
to the next KeyEventPostProcessor in the chain,
ending with the current KeyboardFocusManager. If
true, the KeyEvent is assumed to have been fully
handled (although this need not be the case), and
the AWT will take no further action with regard
to the KeyEvent.

Tested using J2SE 1.4.2 under WinXP
************************************************/
```

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class KeyEventPostProc01{
  public static void main(String[] args){
    new GUI();
  }//end main
}//end class KeyEventPostProc01
//===============================================//

class GUI extends JFrame{
  JTextField fieldA =
                    new JTextField("Field A",12);
  JTextField fieldB =
                    new JTextField("Field B",12);
  JTextField fieldC =
                    new JTextField("Field C",12);

  KeyboardFocusManager manager;

  GUI(){//constructor
    getContentPane().setLayout(new FlowLayout());
    getContentPane().add(fieldA);
    getContentPane().add(fieldB);
    getContentPane().add(fieldC);

    setSize(390,100);
    setTitle("Copyright 2005, R.G. Baldwin");
    setDefaultCloseOperation(
                           JFrame.EXIT_ON_CLOSE);
    setVisible(true);
    manager = KeyboardFocusManager.
                getCurrentKeyboardFocusManager();

    manager.addKeyEventDispatcher(
                        new AlternateDispatcherA(
                         fieldA,fieldC,manager));
    manager.addKeyEventDispatcher(
                        new AlternateDispatcherB(
                         fieldB,fieldC,manager));
    manager.addKeyEventPostProcessor(
                            new PostProcessor());

  }//end constructor
}//end class GUI
//===============================================//

class AlternateDispatcherA
                   implements KeyEventDispatcher{
  JTextField fieldA;
  JTextField fieldC;
  KeyboardFocusManager manager;

  //Constructor
  AlternateDispatcherA(
                   JTextField fieldA,
                   JTextField fieldC,
                   KeyboardFocusManager manager){
    this.fieldA = fieldA;
    this.fieldC = fieldC;
    this.manager = manager;
  }//end constructor

  public boolean dispatchKeyEvent(KeyEvent e){
    if(e.getSource() == fieldA){
      manager.redispatchEvent(fieldC,e);
    }//end if
    return false;
  }//end dispatchKeyEvent
}//end class AlternateDispatcherA
//============================================//

class AlternateDispatcherB
                   implements KeyEventDispatcher{
  JTextField fieldB;
  JTextField fieldC;
  KeyboardFocusManager manager;

  //Constructor
  AlternateDispatcherB(
                   JTextField fieldB,
                   JTextField fieldC,
                   KeyboardFocusManager manager){
    this.fieldB = fieldB;
    this.fieldC = fieldC;
    this.manager = manager;
  }//end constructor

  public boolean dispatchKeyEvent(KeyEvent e){
    if(e.getSource() == fieldB){
      manager.redispatchEvent(fieldC,e);
    }//end if
    return false;
  }//end dispatchKeyEvent
}//end class AlternateDispatcherB
//============================================//

class PostProcessor
                implements KeyEventPostProcessor{

  public boolean postProcessKeyEvent(KeyEvent e){
    System.out.println(
               e.getKeyChar() + " " +
                        getEventType(e.getID()));
    return true;
  }//end postProcessKeyEvent
//--------------------------------------------//

String getEventType(int ID){
  if(ID == KeyEvent.KEY_PRESSED){
    return "keyPressed";
  }else if(ID == KeyEvent.KEY_RELEASED){
    return "keyReleased";
  }else if(ID == KeyEvent.KEY_TYPED){
    return "keyTyped";
  }else{
    return "Unknown event type";
  }//end else
}//end getEventType

}//end class PostProcessor
```

Listing 7

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.*

Richard has participated in numerous consulting projects, and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which has gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

In addition to his programming expertise, Richard has many years of practical experience in Digital Signal Processing (DSP). His first job after he earned his Bachelor's degree was doing DSP in the Seismic Research Department of Texas Instruments. (TI is still a world leader in DSP.) In the following years, he applied his programming and DSP expertise to other interesting areas including sonar and underwater acoustics.

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@DickBaldwin.com

-end-