The KeyEventDispatcher

Baldwin explains how to use KeyEventDispatcher objects to cause a KeyEvent fired by one component to be dispatched to a different component for processing.

Published: December 21, 2004
By Richard G. Baldwin

Java Programming, Notes # 1855

- Preface
- Preview
- Discussion and Sample Code
- Run the Program
- Summary
- What's Next
- Complete Program Listing

Preface

The focus subsystem

This lesson is part of a series of lessons designed to teach you how to use several important features the new 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 Opposite Focus Component.

Previous topics

Previous lessons in this series have dealt with several aspects 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.

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

What do we mean by focus?

One of the ways to describe *focus* is as follows. Among all of the applications showing on the desktop at any point in time, only one will respond to the keyboard. We will refer to that application as the *active* application.

If the *active* 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.

What does Sun have to say?

Before getting into the technical details, I will provide some quotations from Sun that describe a **KeyEventDispatcher**. In discussing the changes to the focus subsystem, Sun has this to say:

*"While the user's **KeyEvents** should generally be delivered to the focus owner, there are rare cases where this is not desirable. ...*

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

*Instances of classes that implement the interface and are registered with the current **KeyboardFocusManager** will receive **KeyEvents** before they are dispatched to the focus owner, 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**, regardless of whether it actually did so, the **KeyboardFocusManager** will take no further action with regard to the **KeyEvent**."*

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 **KeyEventDispatcher** objects to cause a **KeyEvent** fired by one component to be dispatched to a different component for processing.

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 are typed.

Discussion and Sample Code

Description of the program

This lesson presents and explains a sample program named **KeyEventDispatch01**.

This program illustrates the use of a **KeyEventDispatcher** as an alternative to the default dispatching of key events.

The program GUI

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

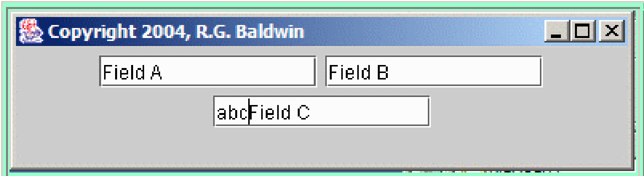

Figure 1 Program GUI.

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 *(this is one case in Java where the order of registration is important):*

- 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 **true**. This causes keystrokes entered into *Field A* to appear in *Field C* instead of appearing in *Field A.*

Returning **true** prevents any further dispatching of the event by the **KeyboardFocusManager**.

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

If the source of the event is not *Field A,* the **AlternateDispatcherA** object returns **false**. This causes the current **KeyboardFocusManager** to deliver the event to **AlternateDispatcherB** for dispatching.

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 **true**. This causes keystrokes entered into *Field B* to appear in *Field C* instead of appearing in *Field B.*

Returning **true** prevents any further dispatching of the event.

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 returns **false**.

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

Must be *Field C*

In this program, that 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 entered into any of the three fields in the GUI in Figure 1 will appear in *Field C.*

This program was tested using J2SE 1.4.2 under WinXP.

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.*

Will discuss in fragments

I will discuss this program in fragments. A complete listing of the program is provided in Listing 10 near the end of the lesson.

Listing 1 shows the class named **KeyEventDispatch01**, including the **main** method.

```
class KeyEventDispatch01{
  public static void main(String[]
args){
    new GUI();
  }//end main
}//end class KeyEventDispatch01
```

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;
```

Listing 2

Listing 2 declares and initializes the three **JTextField** objects shown in Figure 1.

Listing 2 also declares a reference variable named **manager** that will be used to hold a reference to the current **KeyboardFocusManager** object.

The constructor

The constructor for the **GUI** class begins in Listing 3.

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

    setSize(390,100);
    setTitle("Copyright 2004, R.G.
Baldwin");
    setDefaultCloseOperation(

JFrame.EXIT_ON_CLOSE);
    setVisible(true);
```

Listing 3

The code in Listing 3 is completely straightforward. Code like this has been included in hundreds of previous tutorials to create a graphical user interface and to cause it to appear on the screen.

The current KeyboardFocusManager

Continuing with the constructor, the statement in Listing 4 is new to this lesson.

```
    manager = KeyboardFocusManager.

getCurrentKeyboardFocusManager();
```

Listing 4

Listing 4 invokes the static **getCurrentKeyboardFocusManager** method of the **KeyboardFocusManager** class to gain access to the current **KeyboardFocusManager** object.

The object's reference is saved in the variable named **manager** that was declared in Listing 2.

What is a KeyboardFocusManager object

According to Sun,

*"The **KeyboardFocusManager** is responsible for managing the active and focused **Windows**, and the current focus owner.*

*The focus owner is defined as the **Component** in an application that will typically receive all **KeyEvents** generated by the user.*

*The focused **Window** is the **Window** that is, or contains, the focus owner. Only a **Frame** or a **Dialog** can be the active **Window**.*

*(Note that **JFrame** extends **Frame**, and **JDialog** extends **Dialog** - Baldwin.)*

*The native windowing system may denote the active **Window** or its children with special decorations, such as a highlighted title bar. The active **Window** is always either the focused **Window**, or the first **Frame** or **Dialog** that is an owner of the focused Window.*

*The **KeyboardFocusManager** is both a centralized location for client code to query for the focus owner and initiate focus changes, and an event dispatcher for all **FocusEvents**, **WindowEvents** related to focus, and **KeyEvents**."*

Many methods are available

A **KeyboardFocusManager** object provides more than fifty different methods that can be used by client code for *"managing the active and focused **Windows**, and the current focus owner."*

I will be using several of those methods in this lesson. The first such method that I will use is the static **getCurrentKeyboardFocusManager** method shown in Listing 4. As explained earlier, this method returns a reference to the current **KeyboardFocusManager** object.

In addition, I will also be using the following methods in subsequent code:

- addKeyEventDispatcher
- redispatchEvent

Registering KeyEventDispatcher objects

This program defines two different classes that implement the **KeyEventDispatcher** interface:

- AlternateDispatcherA
- AlternateDispatcherB

Continuing with the constructor, the code in Listing 5 registers one object of each of those two classes on the **KeyboardFocusManager**. Registration makes it possible to *"pre-listen to all **KeyEvents**"* and to take appropriate action when a **KeyEvent** occurs.

```
    manager.addKeyEventDispatcher(
                                new
AlternateDispatcherA(

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

fieldB,fieldC,manager));

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

Listing 5

I will explain what I mean by *appropriate action* later.

End of the GUI class

Listing 5 also signals the end of the constructor and the end of the **GUI** class.

The AlternateDispatcherA class

One of the classes that implement the **KeyEventDispatcher** interface is named **AlternateDispatcherA**.

Listing 6 shows the beginning of the class named **AlternateDispatcherA**.

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

Listing 6

(An object instantiated from the ***AlternateDispatcherA*** *class is registered on the* ***KeyboardFocusManager*** *in Listing 5.)*

Listing 6 declares three instance variables that will be populated by the constructor.

References to the three text fields shown in Figure 1, as well as a reference to the **KeyboardFocusManager** are required to make it possible for an object of this class to provide the *appropriate action* mentioned earlier.

The constructor

Listing 7 shows the constructor for the **AlternateDispatcherA** class.

```
  AlternateDispatcherA(
                   JTextField fieldA,
                   JTextField fieldC,

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

Listing 7

The code in the constructor receives the incoming references and saves them in the instance variables declared in Listing 6.

The dispatchKeyEvent method

The **KeyEventDispatcher** interface declares a single method named **dispatchKeyEvent**. All classes that implement the interface must provide a concrete definition of this method. This is the method whose behavior performs the *appropriate action* mentioned earlier.

What does Sun have to say?

Here is part of what Sun has to say about the **dispatchKeyEvent** method.

*"This method is called by the current **KeyboardFocusManager** requesting that this **KeyEventDispatcher** dispatch the specified event on its behalf. This **KeyEventDispatcher** is free to retarget the event, consume it, dispatch it itself, or make other changes.*

*This capability is typically used to deliver **KeyEvents** to **Components** other than the focus owner.*

(That is what is done in this sample program - Baldwin.)

*... Note that if a **KeyEventDispatcher** dispatches the **KeyEvent** itself, it must use **redispatchEvent** to prevent the current **KeyboardFocusManager** from recursively requesting that this **KeyEventDispatcher** dispatch the event again.*

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

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

Definition of the dispatchKeyEvent method

Listing 8 shows the definition of the **dispatchKeyEvent** method in the **AlternateDispatcherA** class.

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

manager.redispatchEvent(fieldC,e);
      return true;
    }else{
      return false;
    }//end else
  }//end dispatchKeyEvent
}//end class AlternateDispatcherA

Listing 8
```

*(Note that the dispatchKeyEvent method receives a reference to the **KeyEvent** object.)*

Is *Field A* the source of the event?

The code in Listing 8 begins by testing to see if the source of the **KeyEvent** is the **JTextField** object referred to by **fieldA**.

(This is the text field identified as Field A in Figure 1.)

If so, the code in Listing 8 invokes the **redispatchEvent** method to *re-dispatch* the event to the **JTextField** object referred to by **fieldC**.

(This is the text field identified as Field C in Figure 1.)

This causes *Field C* to respond to the event instead of *Field A*.

The normal response of a text field to a **KeyEvent** is to display the character associated with the key that was pressed to cause the **KeyEvent** to be fired.

As a result, characters typed into *Field A* in Figure 1 appear in *Field C* instead of *Field A*.

The redispatchEvent method

What does it mean to *dispatch* an event to a component? One way to think of this is to deliver the event object to the component and to ask the component to deal with it in whatever way is appropriate for that component.

For example, when a **KeyEvent** of a particular type is dispatched to a text field, the appropriate behavior is for the text field to extract the character from the event object and to display that character in the text field. That character is also combined with other characters and saved in such a way that it is possible later to get the character content of the text field by invoking a method on the text field.

> *(Experimentation indicates that this happens for a **KeyEvent** of type **keyPressed**, as opposed to type **keyReleased** or type **keyTyped**.)*

Figure 2 contains some of what Sun has to say about the **redispatchEvent** method of the **KeyboardFocusManager** class.

public final void redispatchEvent(
Component target,
AWTEvent e)

Re-dispatches an **AWTEvent** in such a way that the AWT event dispatcher will not recursively request that the **KeyboardFocusManager**, or any installed **KeyEventDispatchers**, dispatch the event again.

Client implementations of **dispatchEvent** and client-defined **KeyEventDispatchers** must call **redispatchEvent(target, e)** instead of **target.dispatchEvent(e)** to dispatch an event.

Parameters:

- **target** - the Component to which the event should be dispatched
- **e** - the event to dispatch

Figure 2

Given the above discussion, Figure 2 should be relatively self-explanatory.

The return value

Returning now to the code in Listing 8, after re-dispatching the **KeyEvent** to *Field C*, the **dispatchKeyEvent** method returns **true**. This causes the **KeyboardFocusManager** to assume that the **KeyEvent** has been dispatched. It will take no further action with respect to that **KeyEvent**.

If the source of the **KeyEvent** in Listing 8 was not *Field A*, the **dispatchKeyEvent** method returns **false**. This causes the **KeyboardFocusManager** to pass the **KeyEvent** to the

KeyEventDispatcher of type **AlternateDispatcherB** that was also registered on the **KeyboardFocusManager**.

> *(Recall that events are delivered to multiple registered **KeyEventDispatcher** objects in the order in which they were registered on the **KeyboardFocusManager**.)*

The AlternateDispatcherB class

The **AlternateDispatcherB** class is shown in Listing 9. This class is very similar to the **AlternateDispatcherA** class shown in Listings 6, 7, and 8.

```
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);
      return true;
    }else{
      return false;
    }//end else
  }//end dispatchKeyEvent
}//end class AlternateDispatcherB
```

Listing 9

The main difference between the **AlternateDispatcherA** class and the **AlternateDispatcherB** class is:

> The **dispatchKeyEvent** method in the **AlternateDispatcherB** class re-dispatches the event to *Field C* if the source of the **KeyEvent** is ***Field B***. In contrast, the **dispatchKeyEvent** method in the **AlternateDispatcherA** class re-dispatches the event to *Field C* if the source of the **KeyEvent** is ***Field A***.

This causes characters entered into *Field B* to appear in *Field C* instead of appearing in *Field B.*

The end of the program

Listing 9 signals the end of the program.

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 **KeyEventDispatcher** objects to cause a **KeyEvent** fired by one component to be dispatched to a different component for processing. As a result, characters typed into one text field appear in another text field, and do not appear in the text field into which they are typed.

What's Next?

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

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

The next lesson in the series will explain the use of the **KeyEventPostProcessor** interface to cause a final stage of customized event processing to take place after a **KeyEvent** has already been dispatched to a component for default processing.

Complete Program Listing

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

```
/*File KeyEventDispatch01.java
Copyright 2004,R.G.Baldwin
Revised 07/27/04

This program illustrates the use of an
alternate KeyEventDispatcher.
```

```
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

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
true.  That causes keystrokes entered into
Field A to appear in Field C instead.  Returning
true prevents any further dispatching of the
event.

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
true.  That causes keystrokes entered into
Field B to appear in Field C instead.  Returning
true prevents any further dispatching of the
event.

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.

Tested using J2SE 1.4.2 under WinXP.
```

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

class KeyEventDispatch01{
  public static void main(String[] args){
    new GUI();
  }//end main
}//end class KeyEventDispatch01
//==============================================//

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 2004, 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));

  }//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);
      return true;
    }else{
      return false;
    }//end else
  }//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);
      return true;
    }else{
      return false;
    }//end else
  }//end dispatchKeyEvent
}//end class AlternateDispatcherB
```

Listing 10

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-