Listeners in Java

by
Peter Lavin

August 1, 2003

Overview

This article seeks to demystify listener interfaces in Java. For example, if you have come to Java from Visual Basic, listeners can seem confusing. If you use a button in VB it comes ready-made with a click event. The programmer doesn't have to worry about how or where mouse or keyboard events will be processed. Not so in Java. Objects are not already event-enabled.

In what follows we will assume some familiarity with one of the most commonly used listeners, "ActionListener". We will then create our own listener interface. This is a lot easier than you might think. Any beginner to intermediate level Java programmer should benefit from this discussion. Java conventions for capitalisation of classes and interfaces will be used for all built-in and programmer created classes. When referring to generic classes, for example, "listeners", lower case names will be used.

Introduction

Listeners are “interfaces” so first a brief discussion of what their function is in Java. Unlike some other object-oriented languages, Java does not allow for multiple inheritance of classes. While this generally seems to be a wise decision, in some cases it is a distinct disadvantage. A class may only inherit from one class. Java compensates for this by allowing for the implementation of any number of interfaces. Classes are “extended” and “interfaces” are “implemented”. This terminology is well chosen. None of the methods of an interface are implemented – they are empty or to be more exact “abstract”.

When a listener is created all the methods of that interface must be implemented. Some listeners, like the ActionListener, have only one method. Creating a class that is an ActionListener is done in the following way:

```
public class MyClass implements ActionListener{
      …
}
```

This class must implement the “actionPerformed” method. The implementation might be something like the following:

```
public void actionPerformed(ActionEvent e){
      //catch button press here
      String straction = e.getActionCommand();
            if (straction == "Add"){
                  //do things related to add
                  …
            }
            if (straction == "Edit"){
                  //do things related to edit
                  …
            }
            if (straction == "Save"){
                  //do things related to save
                  …
            }
}
```

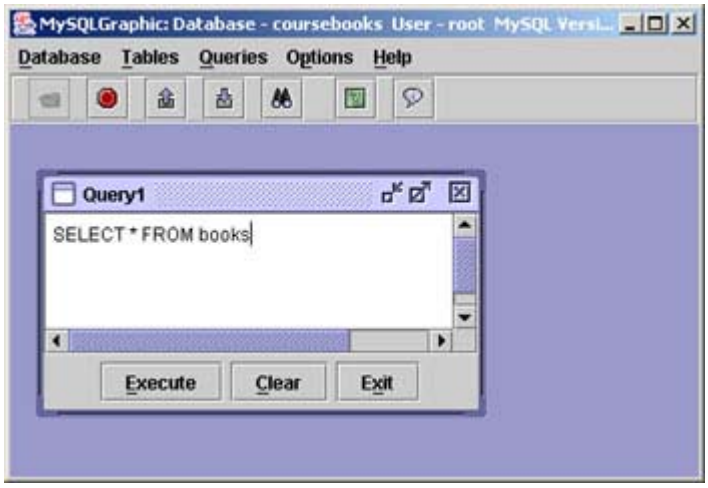
Assume that there are three JButtons in MyClass and these buttons display the text “Add”, “Edit”, “Save” . I can now capture the click event in the above method by adding MyClass as an ActionListener to the individual JButtons.

Following these steps will create a basic listener. The interface and its methods are implemented and then it is added to the appropriate objects.

However, the process whereby a listener is notified can seem mysterious. How does an event get sent to the Listener class? This will be clarified by creating our own listener interface from scratch. You'll see that it is surprisingly easy.

Set the Scene

Let's examine a situation where it might make sense to create our own listener. Assume the following - A main window needs to take some action based on a dependent window. The main window of this application is a multi-document interface. This simply means that you can have many documents open simultaneously just as you can with any word processor. To clarify have a look at the picture below.



When the "Execute" button in the window entitled "Query1" is pressed, the main window needs to display the result of the select query. Please note that you do not need to know SQL to follow this example. You only need to understand that this select query retrieves information from a database and this information will then be displayed. Because the main window is a multi-document interface it needs to be the "parent" of all windows created. For this reason a dependent window should not create another window itself but inform the main window and let it do so. Creating a listener is the best solution to this problem.

Create A Listener

The first step is to create our listener. Let's first look at the necessary code.

```
//SQLListener interface
////////////////////////////////////////////////
```

```
//put package first
package dbpack;
//put imports here

public interface SQLListener{
////////////////////////////////////////////////
//public functions
////////////////////////////////////////////////
      //abstract method that must be implemented
      public void SQLAlert(SQLEvent sqle);

}
```

That's all we need. Take out the comments and we have three lines of code. An interface is simply made up of one or more undefined or "abstract" methods. This method must be defined by any class that implements this interface. This method will receive an event object that will be the basis for further action.

Create An Event Object

In the code above we reference a class that does not yet exist, SQLEvent. This is the event object that will tell the main window what to do. Let's see what this class looks like.

```
////////////////////////////////////////////////////////////////////////
//SQLEvent class
////////////////////////////////////////////////////////////////////////
//put package first
package dbpack;
//put imports here
import java.util.*;
public class SQLEvent extends EventObject{
      //data members
      private String name;
      private String SQLmessage;
////////////////////////////////////////////////////////////////////////
//constructor
////////////////////////////////////////////////////////////////////////
    public SQLEvent(Object source, String newname, String msg){
            super(source);
            name = newname;
            SQLmessage = msg;
    }
//end constructor
////////////////////////////////////////////////////////////////////////
//public functions
////////////////////////////////////////////////////////////////////////
      public String getName(){
            return name;
      }
```

```
      public String getMessage(){
            return SQLmessage;
}
////////////////////////////////////////////////////////////////////////
//end class SQLEvent
////////////////////////////////////////////////////////////////////////
```

Again we have a fairly simple class with only a constructor, two data members and two methods. Within the context of our example, this event class will communicate the nature and content of an SQL statement. This class will also, of course, inherit the functionality of its parent.

Main Window Implements Listener

We now need a class that will implement the listener we have created. We have already determined that it is the main window that needs to know about SQLEvents so that it can display and position a new dependent window. Here's how our listener class would be implemented:

```
public class MyMainWindow extends JFrame implements SQLListener{
      …
}
```

If we immediately recompile our class after adding the listener, the compiler will object. Why? Because we claimed the main window class was an SQLListener, but we have not defined a method called "SQLAlert". We didn't complete the "contract" that a programmer enters into when he implements an interface. That is, all methods of the interface must be defined. We can do this by adding the following code to the main window class:

```
public void SQLAlert(SQLEvent sqle){
      //code here to process the event
}
```

This method will process the event that is initiated when the execute key is pressed in the dependent window.

Adding the Listener to the Dependent Window

We haven't yet created a method in the dependent window to add an SQLListener. The requirements are not onerous. Firstly, we need a class level object to hold any and all SQLListeners. For this purpose we will use a Vector. The declaration is as follows:

```
private Vector SQLlisteners;
```

Now that we've got the container, we need a method in the dependent window that will allow us to add SQLListeners. Following Java naming conventions it looks like this:

```
public void addSQLListener(SQLListener sqll){
      //add main frame to vector of listeners
      if (SQLlisteners.contains(sqll))
            return;
      SQLlisteners.addElement(sqll);
}
```

This method must be public because it will be invoked outside the class. This method will be called when the dependent SQL window is created. The listener that will be added is the main window class. (It also makes sense to create a method to remove SQLListeners but this is not important for our current example.) Adding a listener will require code such as the following:

```
DependentWindow dw = new DependentWindow();
dw.addSQLListener(main);
```

So far what we've done doesn't really differ much from using one of the existing listener classes. However, through creating our own listener we can now more clearly understand how a message is relayed from a class to its listeners. Note that our "addSQLListener" method enables the dependent window to hold a reference to the main window.

How the Event Occurs

Let's do a short recap. A main window needs to take some action based on a dependent window. In this case the result of a select query needs to be displayed and positioned by the main window. When using one of the built-in listeners, for example an ActionListener, we simply put the code we need executed into the "actionPerformed" method. The details of how this method gets called are hidden inside the built-in class. With our own class we can see exactly what happens and how an event gets "sent". Find below the method that is called whenever the "Execute" button in our dependent window is clicked.

```
private void notifySQLAlert(){
      //code here to get text from textarea
      …
      //code for how the message gets sent
      if (!strmessage.equals("")){
```

```
                //create the message encapsulated in an SQLEvent
                sqle = new SQLEvent (this, type, strmessage);
                Vector vtemp = (Vector)SQLlisteners.clone();
                for (int x = 0; x < vtemp.size(); x++){
                      SQLListener target = null;
                      target = (SQLListener)vtemp.elementAt(x);
                      target.SQLAlert(sqle);
                }
        }
}
```

Because a reference to the SQLListener is stored in a Vector inside the dependent class the public method “SQLAlert” may be called against this reference. Information encapsulated inside an event object is passed into this method and processed by the listener. Our main window class can now decide what kind of new window to create in order to display the results of our query.

Conclusion

Events are not really “sent” to a listener rather a listener is sent to the event. The method “addSQLListener” is the means by which this is done – the reporter on the scene if you like. The event is communicated to the interested parties using the public method(s) of the listener. The compiler helps the programmer use interfaces properly by enforcing the implementation of all method(s) of the interface.

About the Author

Peter Lavin runs a Web Design/Development firm in Toronto, Canada. For more information visit http://www.softcoded.com/. Peter may be reached at peterlavin@sympatico.ca.