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

Event Handling in JDK 1.1, Creating, Trapping, and Processing Custom Event Types

Java Programming, Lecture Notes # 100, Revised 02/24/99.

Preface

Students in Prof. Baldwin's **Intermediate Java Programming** classes at ACC are responsible for knowing and understanding all of the material in this lesson.

JDK 1.1 was formally released on February 18, 1997. This lesson was originally written on March 27, 1997 using the software and documentation in the JDK 1.1 download package.

On 2/24/99, it was confirmed that the sample program in this lesson works properly with JDK 1.2 under Win95.

Introduction

The purpose of this lesson is to illustrate one way that you can create your own custom event types. In other words, if you need a type of event that is not one of the standard types, it is possible for you to create a new type. Then you can cause events of that new type to be generated, trapped, and processed.

This is an important aspect of Java Beans technology, so it is worth learning how to do it while we are studying the various ways of working with events. We will be referring back to this material in a future lesson on Java Beans.

Essential Ingredients for Creating Custom Event Types

Using the techniques illustrated in this program, the essential ingredients for creating, trapping, and processing a new custom event type are:

- Define a class from which objects of the new custom event type can be instantiated. It should extend **EventObject**.
- Define an **EventListener** interface for the new custom event type. This interface should be *implemented* by **Listener** classes for the new type. The interface should extend **EventListener**.
- Define a **Listener** class that *implements* the interface for the new custom event type.
- Define a class for instantiating objects capable of generating the new custom event type. In this program, that class extends **Component** and is in the category of "program-generated events."
- Define a method, preferably named **add<CustomEventNameListener**, that can maintain a list of **Listener** objects registered to listen for events of the new custom type (In this bare-bones program, to avoid complexity, the length of the list is limited to only one **Listener** object. See discussion in the next section.)
- Define a method that dispatches an event object of the new custom event type to all **Listener** objects registered in the above list by invoking the central event-trapping method of those **Listener** objects. This event-trapping method is the method declared in the **EventListener** interface for the new custom event type. In this program, it is named **customEventTrapped()**.

Discussion of Multiple Listener Objects

The *delegation event model* can support the registration of multiple **Listener** objects on a single event-generating source object. The model can also support the registration of objects from a single **Listener** class on multiple event-generating source objects. Finally, the model can support various combinations of the two.

In an earlier lesson on "program-generated events", we learned how to use the **add()** method of the **AWTEventMulticaster** class to create and maintain a list of registered **Listener** objects. This method supports the capability to register multiple **Listener** objects on a single event-generating source object. When you then invoke an event dispatching method (such as **actionPerformed()**) on the reference to the list, the event is automatically dispatched to every **Listener** object in the list.

Unfortunately, the **add()** method of the **AWTEventMulticaster** class cannot be used directly to maintain a list of registered **Listener** objects for new custom event types because there is no overloaded version of the **add()** method whose signature matches the new event type.

One way to create and maintain such a list of registered **Listener** objects is to embark on a *data structures* programming exercise in *list processing*. Since *data structures* is not the thrust of this lesson, I decided not to take that approach. If you would like to see some example code showing how to create and maintain your own list of registered objects in a situation similar to this, see Chapter 15 of Inside Java by Siyan and Weaver, ISBN 1-56205-664-6, $55.00 USA/$77.95 CAN.

Another approach that would probably work would be to subclass the **AWTEventMulticaster** class and overload the **add()** method. The method would have a signature that matches the new type of event Of course, the body of the overloaded method would still have to provide the list-processing code so this could become a little complicated.. If I can find the time, I may try to do that and update this lesson later.

In the meantime, the primary thrust of this lesson on custom event types can be adequately illustrated by limiting the number of **Listener** objects to one object per object-registration list. Also, that limitation will probably satisfy a reasonable percentage of real-world situations. Therefore, I decided to avoid the complexity of *list-processing* code in the sample program in this lesson by limiting the number of **Listener** objects to one **Listener** object per object-registration list..

Sample Program

This program is designed to be compiled and run under JDK 1.1.

This is a bare-bones program that illustrates the ability to create, trap, and process events of new custom types.

The program also illustrates program-generated events, a capability that was also illustrated in an earlier sample program in a previous lesson. You should make certain that you understand that material before embarking on this lesson.

A non-visual object class (named **NonVisual**) is defined. Objects of this class are capable of generating events of the new custom event type.

The essential ingredients for creating, trapping, and processing events of new custom types were listed in an earlier section of this lesson.

In this program, two objects of the **NonVisual** class are instantiated and one **CustomEventListenerClass** class is defined. The **CustomEventListenerClass** class *implements* the **CustomEventListener** interface and defines a method for processing **CustomEvent** objects named **customEventTrapped()**.

Objects of the **CustomEventListenerClass** class are instantiated and registered to listen for custom events on both of the **NonVisual** objects mentioned above. Therefore, this program also illustrates the capability to instantiate and register **Listener** objects from a single **Listener** class on multiple event-generating source objects.

The **NonVisual** object contains an instance method named **generateCustomEvent()** This method is designed to dispatch an event of the new custom event type to the **Listener** object registered on the source object by invoking the **customEventTrapped()** method of the **Listener** object.

In the program, the **generateCustomEvent()** method is invoked on both of the **NonVisual** objects causing custom events to occur and to be trapped and processed by the registered **CustomEventListenerClass** objects.

The processing of the event in this case is fairly simple. Identification and source information is extracted from the **CustomEvent** object that is passed as a parameter to the **customEventTrapped()** method. That information is displayed on the standard output device.

The output from the program for one particular run was:

```
Copyright 1997, R.G.Baldwin
customEventTrapped() method invoked on First NonVisualObject
Source of event was NonVisual[,0,0,0x0,invalid]
customEventTrapped() method invoked on Second NonVisualObject
Source of event was NonVisual[,0,0,0x0,invalid]
```

The program was tested using JDK 1.1 running under Win95.

Interesting Code Fragments

The first interesting code fragment is the definition of the class from which event objects of a new custom event type can be instantiated. The entire class definition is shown below.

```
class CustomEvent extends EventObject{
  String id;//instance variable for an object of this type
  //------------------------------------------------------------
  CustomEvent(Object obj, String id){//constructor
    super(obj);//pass the Object parameter to the superclass
    this.id = id;//save the String parameter
  }//end constructor
  //------------------------------------------------------------
  //method to return the saved String parameter
  String getCustomID(){
    return id;
  }//end getCustomID
}//end class CustomEvent
```

As you can see, this class extends **java.util.EventObject**. The constructor for an object of this class receives two parameters. The first parameter is a reference to the object that generated the event. This parameter is passed to the constructor of the superclass, **EventObject**, where it is stored for later access using the **getSource()** method of class **EventObject.**

The second parameter is **String** identification information provided to the constructor when the **CustomEvent** object is instantiated. In this program, the **String** information is simply a **String** object stored as an instance variable in the source object when it was instantiated. You could pass virtually any identifying or command information in this manner. The **String** information is stored as an instance variable in the **CustomEvent** object.

The class also provides an accessor method that returns the **String** stored as an instance variable in the object.

The next interesting code fragment is the definition of a new **Listener** interface named **CustomEventListener** that extends **EventListener**. This interface declares a method named **customEventTrapped()** which is the primary event processing method of **Listener** objects of this class. It is through this method that events are dispatched to the listener objects.

```
interface CustomEventListener extends EventListener{
  void customEventTrapped(CustomEvent e);
}
```

The next interesting code fragment implements the above interface and defines a **Listener** class for events of this new custom type. It is a *plain-vanilla* **Listener** class that overrides the **customEventTrapped()** method declared in the interface. The overridden method uses the accessor methods to access and display the **String** data and the identification of the source object from the **CustomEvent** object.

```
class CustomEventListenerClass
                     implements CustomEventListener{
  public void customEventTrapped(CustomEvent e){
    System.out.println(
      "customEventTrapped() method invoked on " +
                                   e.getCustomID());
    System.out.println("Source of event was " +
                                     e.getSource());
  }//end customEventTrapped
}//end class CustomEventListenerClass
```

The remaining interesting code fragments come from the definition of a class designed to generate events of this new custom event type. This class is very similar to the class developed in a previous lesson on "program-generated events." As mentioned earlier, this bare-bones version can only support one **Listener** object registered on each event-generating source object.

We start with the beginning of the class definition that shows that this class extends **Component** and maintains two instance variables. One of the instance variables is the **String ID** information initialized by the constructor. The other instance variable is a reference to the **Listener** object registered on the source object. The constructor code is too simple to merit display in this section.

```
class NonVisual extends Component {
  String ID; // The ID of this object
  //Reference to single Listener object
  CustomEventListenerClass customListener;
```

The next interesting code fragment is the code that actually registers a **Listener** object. This is very similar to the code in the previous lesson except that this code doesn't use the **AWTEventMulticaster.add()** method to create and maintain a list of registered **Listener** objects. This code supports only one registered **Listener** object and contains logic to terminate the program if an attempt is made to register more than one **Listener** object.

```
 public void addCustomEventListener(
                         CustomEventListenerClass listener){
    //Do not attempt to add more than one Listener object.
    if(customListener == null)
      customListener = listener;//one listener only
    else{
      System.out.println(
                 "No support for multiple Listener objects");
      //terminate on attempt to register multiple listeners
      System.exit(0);
    }
  }//end addCustomEventListener()
```

The next interesting code fragment is the method that actually dispatches an event of the new custom type. In this case, the dispatching is accomplished by invoking the **customEventTrapped()** method of the registered **Listener** object and passing an object of the **CustomEvent** type as a parameter. This is also nearly identical to code in the previous lesson on "program-generated events" and is a typical way of dispatching events in JDK 1.1.

```
  public void generateCustomEvent() {
    customListener.customEventTrapped(
        new CustomEvent(this, ID));
  }//end generateCustomEvent
```

These code fragments have illustrated the essential ingredients of creating, trapping, and processing new custom event types. A complete listing of the program is contained in the next section.

Program Listing

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

```
/*File Event26.java Copyright 1997, R.G.Baldwin
This program is designed to be compiled and run under JDK 1.1

This is a bare-bones program that illustrates the ability to
create, trap, and process events of new custom types.

The output from the program for one particular run was:
```

```
Copyright 1997, R.G.Baldwin
customEventTrapped() method invoked on First NonVisualObject
Source of event was NonVisual[,0,0,0x0,invalid]
customEventTrapped() method invoked on Second NonVisualObject
Source of event was NonVisual[,0,0,0x0,invalid]

The program was tested using JDK 1.1 running under Win95.
*/
//=============================================================
import java.awt.*;
import java.awt.event.*;
import java.util.EventListener;
import java.util.EventObject;
//=============================================================

public class Event26 {
  public static void main(String[] args){
    new Event26();//instantiate an object of this type
  }//end main
//-------------------------------------------------------------
  public Event26(){//constructor
    System.out.println("Copyright 1997, R.G.Baldwin");
    NonVisual firstNonVisualObject = new NonVisual(
                                   "First NonVisualObject");
    firstNonVisualObject.addCustomEventListener(
                            new CustomEventListenerClass());
    //create an event
    firstNonVisualObject.generateCustomEvent();

    NonVisual secondNonVisualObject = new NonVisual(
                                  "Second NonVisualObject");
    secondNonVisualObject.addCustomEventListener(
                            new CustomEventListenerClass());
    //The following statement causes the program to terminate
    // with the message
    // "No support for multiple Listener objects" because it
    // violates the restriction that this bare-bones program
    // only allows one Listener object to be registered on
    // each NonVisual object.  Therefore, the statement
    // has been disabled by making it a comment. It is
    // included here for illustration purposes only.
    //secondNonVisualObject.addCustomEventListener(
                            new CustomEventListenerClass());
    //create an event
    secondNonVisualObject.generateCustomEvent();

  }//end constructor
}//end class Event26
//=============================================================

//Class to define a new type of event
class CustomEvent extends EventObject{
  String id;//instance variable for an object of this type
  //-----------------------------------------------------------
  CustomEvent(Object obj, String id){//constructor
```

```
    super(obj);//pass the Object parameter to the superclass
    this.id = id;//save the String parameter
  }//end constructor
  //------------------------------------------------------------
  //method to return the saved String parameter
  String getCustomID(){
    return id;
  }//end getCustomID
}//end class CustomEvent
//=============================================================

//Define interface for the new type of event listener
interface CustomEventListener extends EventListener{
  void customEventTrapped(CustomEvent e);
}//
//=============================================================

//Listener class to respond to custom events
class CustomEventListenerClass
                              implements CustomEventListener{
  public void customEventTrapped(CustomEvent e){
    System.out.println(
      "customEventTrapped() method invoked on " +
                                            e.getCustomID());
    System.out.println("Source of event was " +
                                              e.getSource());
  }//end customEventTrapped
}//end class CustomEventListenerClass
//=============================================================

//Class to create object capable of generating Custom events.
// Note:  This is a bare-bones version which can only support
// a single Listener object for the Custom event type.
class NonVisual extends Component {
  String ID; // The ID of this object
  //Reference to single Listener object
  CustomEventListenerClass customListener;
  //------------------------------------------------------------
  public NonVisual(String ID) {//Construct a NonVisual object
      this.ID = ID;
  }//end constructor
  //------------------------------------------------------------
  public void addCustomEventListener(
                          CustomEventListenerClass listener){
    //Do not attempt to add more than one Listener object.
    if(customListener == null)
      customListener = listener;//one listener only
    else{
      System.out.println(
                 "No support for multiple Listener objects");
      //terminate on attempt to register multiple Listeners
      System.exit(0);
    }
  }//end addCustomEventListener()
  //------------------------------------------------------------
  public void generateCustomEvent() {
    customListener.customEventTrapped(
        new CustomEvent(this, ID));
  }//end generateCustomEvent
}//end class NonVisual
//============================================================
```

Review

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

A - See solution below.

```
/*File SampProg130.java from lesson 100
Copyright 1997, R.G.Baldwin

Without viewing the solution that follows, write a Java
application that uses the definition given for the class
named SampProg130 and produces the output shown below.

Copyright 1997, R.G.Baldwin
CustomEvent[source=ClassA[,0,0,0x0,invalid]]
on ClassA Object
CustomEvent[source=ClassB[,0,0,0x0,invalid]]
on ClassB Object

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

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

public class SampProg130 {
  public static void main(String[] args){
    new SampProg130();//instantiate an object of this type
  }//end main
//-----------------------------------------------------------
  public SampProg130(){//constructor
    System.out.println("Copyright 1997, R.G.Baldwin");

    ClassA classAObject = new ClassA("ClassA Object");
    classAObject.addCustomEventListener(
                           new CustomEventListenerClass());
    classAObject.generateCustomEvent();//create an event

    ClassB classBObject = new ClassB("ClassB Object");
    classBObject.addCustomEventListener(
                           new CustomEventListenerClass());
    classBObject.generateCustomEvent();//create an event
  }//end constructor
}//end class SampProg130
//===========================================================

//One class to create object capable of generating Custom
// events. Note:  This is a bare-bones version which can
// only support a single Listener object for the Custom
// event type.
class ClassA extends Component {
  String ID; // The ID of this object
  //Reference to single Listener object
  CustomEventListenerClass customListener;
  //---------------------------------------------------------
  public ClassA(String ID) {//Constructs a ClassA object
      this.ID = ID;
  }//end constructor
  //---------------------------------------------------------
  public void addCustomEventListener(
                      CustomEventListenerClass listener) {
    //Do not attempt to add more than one Listener object.
    if(customListener == null) customListener = listener;
    else{
      System.out.println(
               "No support for multiple Listener objects");
      //terminate on attempt to register multiple Listeners
      System.exit(0);
    }//end else
  }//end addCustomEventListener()
  //---------------------------------------------------------
  public void generateCustomEvent() {
    customListener.customEventTrapped(
        new CustomEvent(this, ID));
  }//end generateCustomEvent
}//end class ClassA
//===========================================================


//Another Class to create object capable of generating
// Custom events. Note:  This is a bare-bones version which
// can only support a single Listener object for the Custom
// event type.
class ClassB extends Component {
  String ID; // The ID of this object
  //Reference to single Listener object
  CustomEventListenerClass customListener;
  //---------------------------------------------------------
  public ClassB(String ID) {//Constructs a ClassB object
      this.ID = ID;
  }//end constructor
  //---------------------------------------------------------
  public void addCustomEventListener(
                      CustomEventListenerClass listener) {
    //Do not attempt to add more than one Listener object.
    if(customListener == null) customListener = listener;
    else{
      System.out.println(
               "No support for multiple Listener objects");
      //terminate on attempt to register multiple Listeners
      System.exit(0);
    }//end else
  }//end addCustomEventListener()
  //-----------------------------------------------------------
  public void generateCustomEvent() {
    customListener.customEventTrapped(
        new CustomEvent(this, ID));
  }//end generateCustomEvent
}//end class ClassB
//===========================================================

//Class to define a new type of event
class CustomEvent extends EventObject{
  String id;//instance variable for an object of this type
  //-----------------------------------------------------------
  CustomEvent(Object obj, String id){//constructor
    //pass the Object parameter to the superclass
    super(obj);
    this.id = id;//save the String parameter
  }//end constructor
  //-----------------------------------------------------------
  //method to return the saved String parameter
  String getCustomID(){
    return id;
  }//end getCustomID
}//end class CustomEvent
//===========================================================

//Define interface for the new type of event listener
interface CustomEventListener extends EventListener{
  void customEventTrapped(CustomEvent e);
}//
//===========================================================

//Listener class to respond to custom events
class CustomEventListenerClass
                            implements CustomEventListener{
  public void customEventTrapped(CustomEvent e){
    System.out.println("" + e.toString());
    System.out.println("on " + e.getCustomID());
  }//end customEventTrapped
}//end class CustomEventListenerClass
//===========================================================
```

-end-