**8.17 Event Model :-**  
  
People who are familiar with the MS-Windows or X-Windows SDK programming, remember that the Windows Procedure was braced against a flood of events and then a giant ‘switch’ statement was used to sort through these events.   
  
Similarly, in The Microsoft Foundation Classes programming model or The Motif programming model, each component has to be over-ridden to make it do what you want it to do - something similar to the older [Java 1.0 Inheritance Event Model](http://www.execpc.com/%7Egopalan/java/java10_emodel.html).  
  
Neither of the approaches discussed above are really elegant. The first approach is the easiest to implement, but any time you use a switch statement, you somehow have this nagging feeling that perhaps there is something un-object-oriented in the design or that a different class heirarchy and design would have made this go away.  
  
The second approach is more object-oriented, but involves a great deal of work.  
  
There were a few flaws with the Java 1.0 Inheritance Event Model. One major problem was that an event could only be handled by the component which generated it or by one of the containers that contained the original component.  
  
Another drawback with the Java 1.0 Inheritance Event Model was that a large number of CPU cycles were wasted on uninteresting events. Any event in which a program had no interest would just flow through the containment heirarchy before it was eventually discarded. The original model provided no way to disable irrelevant events.  
  
With the new 1.1 Delegation Event Model, a component can be told which object or objects should be notified when the component generates a particular type of event. If a component is not interested in an event type, then events of that type will not be propagated.  
  
According to Sun, the reasons for coming out with the Java 1.1 Delegation Event Model are:

The 1.1 Event model is based on the concept of an ‘Event Source’ and ‘Event Listeners’. Any object that is interested in receiving messages (or events ) is called an Event Listener. Any object that generates these messages ( or events ) is called an Event Source.

|  |
| --- |
|  |

Figure: Java 1.1 Event Handling mechanism

The Event Source Object maintains a list of listeners who are interested in receiving events that it produces. The Event Source Object provides methods that allow the listeners to add themselves ( ‘Register’ ) or remove themselves from this list of ‘interested’ objects. When the Event Source Object generates an event, or when a user input event occurs on the Event Source Object, the Event Source Object notifies all the listeners that the event has occurred.

|  |
| --- |
| **//This is one typical Event Source Object implementation** public class EventSource extends Applet { public Label m\_Label; public Button m\_Button; public int m\_nCounter;  public void init() { m\_Label = new Label( "Go ahead and click away." ); m\_Button = new Button( "Click Here" ); EventListener listenerObject = new EventListener( this ); **//Register with event source using addXListener** m\_Button.addActionListener( listenerObject );  add( m\_Label ); add( m\_Button ); }  } |
|  |
| **//This is one typical Event Listener Object implementation** class EventListener implements ActionListener { private EventSource m\_sourceObject;  public EventListener( EventSource sourceObject ) { m\_sourceObject = sourceObject; }  **//This method handles the EventObjects fired by the EventSource Object //Note: An ActionEvent is an EventObject** public void actionPerformed( ActionEvent e ) { m\_sourceObject.m\_Label.setText( "That was click " +  ( ++m\_sourceObject.m\_nCounter ) +  " buddy." ); }  } |

An Event Source Object notifies an Event Listener Object by invoking a method on it and passing it an EventObject ( an instance of a subclass of java.util.EventObject ). In order for the source to invoke a method on a listener, all listeners must implement the required method. This is ensured by requiring that all event listeners for a particular type of event implement a corresponding interface.  
  
The Java 1.1 Event Delegation Model is based on four concepts:

|  |  |
| --- | --- |
|  | The Event Classes |
|  | The Event Listeners |
|  | Explicit Event Enabling |
|  | Adapters |

The Event Classes

The Java 1.1 event model defines a large number of event classes. The figure below shows the heirarchy of these event classes.

|  |
| --- |
|  |

Figure: The Event Class Heirarchy

If you will notice carefully, different classes of events are represented by different Java classes.   
  
**java.util.EventObject** - Every event is a subclass of **java.util.EventObject**. It is a very general class with only one method of interest:  
**Object getSource()**  
This method returns the object that originated the event. Every event has a source object, from which the event originated. This method returns a reference to that source.  
  
**java.awt.AWTEvent -** AWT Events, which is what we are concerned here, are subclasses of java.awt.AWTEvent. This is the superclass of all the delegation model event classes. The most interesting method in this class is  
**int getID()**   
This method returns the ID of the event. An event’s ID is an int that specifies the exact nature of the event. This value is used to distinguish the various types of events that are represented by any event class.  
  
**java.awt.event -** The subclasses of **java.awt.AWTEvent** represent the various event types that can be generated by the various AWT components. All the various types of AWT events, are placed in a separate package called **java.awt.event** for the sake of convenience.

|  |  |  |
| --- | --- | --- |
|  | **ActionEvent** | generated by component activation |
|  | **AdjustmentEvent** | generated by adjustment of adjustable components such as scroll bars |
|  | **ContainerEvent** | generated when components are added to or removed from a container |
|  | **FocusEvent** | generated when a component receives input focus |
|  | **ItemEvent** | generated when an item is selected from a list, choice or check box |
|  | **KeyEvent** | generated by keyboard activity |
|  | **MouseEvent** | generated by mouse activity |
|  | **PaintEvent** | generated when a component is painted |
|  | **TextEvent** | generated when a text component is modified |
|  | **WindowEvent** | generated by window activity like minimising or maximising |

There are a number of ways to handle the events listed above. One way is to delegate event handling to a listener object. Another way is to explicitly enable the originating component to handle its own events.