

This section was originally published in the first edition of Learning Java as part of the AWT/Swing chapters. It describes how events were handled prior to Java 1.1 and how to enable legacy support for systems using this style of events. Since this is not of wide interest, we have moved this material to the CD.

Old-Style and New-Style Event Processing

Although Java is still a youngster, it has a bit of a legacy. Versions of Java before 1.1 used a different style of event delivery. Back in the old days (a couple of years ago), event types were limited, and an event was delivered only to the `Component` that generated it or one of its parent containers. The old style component event-handler methods (now deprecated) returned a `boolean` value declaring whether or not they had handled the event:

```
boolean handleEvent( Event e ) {  
    ...  
}
```

If the method returns `false`, the event is automatically redelivered to the component's container to give it a chance. If the container does not handle it, it is passed on to its parent container, and so on. In this way, events were propagated up the containment hierarchy until they were either consumed or passed over to the component peer, just as current `InputEvents` are ultimately interpreted using the peer if no registered event listeners have consumed them.

Note that this older style of event processing applies only to AWT components. The newer Swing components handle only events with the new model.

Although this style of event delivery was convenient for some simple applications, it is not very flexible. Events could be handled only by components, which meant that you always had to subclass a `Component` or `Container` type to handle events. This was a degenerate use of inheritance (i.e., bad design) that led to the creation of lots of unnecessary classes.

We could, alternatively, receive the events for many embedded components in a single parent container, but that would often lead to very convoluted logic in the container's event-handling methods. It is also very costly to run every single AWT event through a gauntlet of (often empty) tests as it traverses its way up the tree of containers. This is why Swing now provides the more dynamic and general event source/listener model that we have described in this chapter. The old-style events and event-handler methods are, however, still with us.

Java is not allowed to simply change and break an established API. Instead, older ways of doing things are simply deprecated in favor of the new ones. This means that code using the old-style event handler methods will still work; you may see old-style code around for a long time. The problem with relying on old-style event delivery, however, is that the old and new ways of doing things cannot be mixed.

By default, Java is obligated to perform the old behavior—offering events to the component and each of its parent containers. However, Java turns off the old-style delivery whenever it thinks that we have elected to use the new style. Java determines whether a `Component` should receive old-style or new-style events based on whether any event listeners are registered, or whether new style events have been explicitly enabled. When an event listener is registered with a `Component`, new-style events are implicitly turned on (a flag is set). Additionally, a mask is set telling the component the types of events it should process. The mask allows components to be more selective about which events they process.

`processEvent()`

When new-style events are enabled, all events are first routed to the `dispatchEvent()` method of the `Component` class. The `dispatchEvent()` method examines the component's event mask and

decides whether the event should be processed or ignored. Events that have been enabled are sent to the `processEvent()` method, which looks at the event's type and delegates it to a helper processing method named for its type. The helper processing method finally dispatches the event to the set of registered listeners for its type.

This process closely parallels the way in which old-style events are processed and the way in which events are first directed to a single `handleEvent()` method that dispatches them to more specific handler methods in the `Component` class. The differences are that new-style events are not delivered unless someone is listening for them, and the listener registration mechanism means that we don't have to subclass the component in order to override its event-handler methods and insert our own code.

Enabling new-style events explicitly

Still, if you are subclassing a `Component` or you really want to process all events in a single method, you should be aware that it is possible to emulate the old-style event handling and override your component's event-processing methods. Call the `Component`'s `enableEvents()` method with the appropriate mask value to turn on processing for the given type of event. You can then override the corresponding method and insert your code. The mask values, listed in the following table, are found in the `java.awt.AWTEvent` class.

AWT Event Masks

<code>java.awt.AWTEvent</code> mask	Method
<code>COMPONENT_EVENT_MASK</code>	<code>ProcessComponentEvent()</code>
<code>FOCUS_EVENT_MASK</code>	<code>processFocusEvent()</code>
<code>KEY_EVENT_MASK</code>	<code>processKeyEvent()</code>
<code>MOUSE_EVENT_MASK</code>	<code>processMouseEvent()</code>
<code>MOUSE_MOTION_EVENT_MASK</code>	<code>ProcessMouseMotionEvent()</code>

For example:

```
public void init( ) {  
    ...  
    enableEvent( AWTEvent.KEY_EVENT_MASK );  
}  
  
public void processKeyEvent(KeyEvent e) {  
    if ( e.getID( ) == KeyEvent.KEY_TYPED ) {  
        // do work  
    }  
}
```

```

    }
    super.processKeyEvent(e);
}

```

If you do this, it is very important that you remember to make a call to `super.process...Event()` in order to allow normal event delegation to continue. Of course, by emulating old-style event handling, we're giving up the virtues of the new style; this code is a lot less flexible than the code we could write with the new event model. As we've seen, the user interface is hopelessly tangled with the actual work your program does. A compromise solution would be to have your subclass declare that it implements the appropriate listener interface and register itself, as we have done in the simpler examples in this book:

```

class MyPanel implements KeyListener {
    public void init( ) {
        addKeyListener( this );
        ...
    }

    public void keyTyped(KeyEvent e) {
        // do work
    }
}

```