The Delegation Event Model

The modern approach to handling events is based on the *delegation event model,* whic
defines standard and consistent mechanisms to generate and process events. Its concept
quite simple: a *source* generates an event and sends it to one or more *listeners.* In this schem
the listener simply waits until it receives an event. Once an event is received, the listene
processes the event and then returns. The advantage of this design is that the application log
that processes events is cleanly separated from the user interface logic that generates thos
events. A user interface element is able to "delegate" the processing of an event to a separat
piece of code.

In the delegation event model, listeners must register with a source in order to receive a
event notification. This provides an important benefit: notifications are sent only to listener
that want to receive them. This is a more efficient way to handle events than the design use
by the original Java 1.0 approach.

The following sections define events and describe the roles of sources and listeners.

Events

In the delegation model, an *event* is an object that describes a state change in a source. a
event can be generated as a consequence of a person interacting with the elements in
graphical user interface. Some of the activities that cause events to be generated are pressin

a button, entering a character via the keyboard, selecting an item in a list, and clicking th mouse.

Events may also occur that are not directly caused by interactions with a user interface. Fo example, an event may be generated when a timer expires, a counter exceeds a value, software or hardware failure occurs, or an operation is completed. You are free to defin events that are appropriate for your application.

Event Sources

A *source* is an object that generates an event. This occurs when the internal state of that objec changes in some way. Sources may generate more than one type of event.

A source must register listeners in order for the listeners to receive notifications about specific type of event. Each type of event has its own registration method. Here is the genera form:

public void add*Type*Listener (*Type*Listener *el*)

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For exampl the method that registers a keyboard event listener is called **addKeyListener( )**. The metho that registers a mouse motion listener is called **addMouseMotionListener( )**. When an even occurs, all registered listeners are notified and receive a copy of the event object. This is know as *multicasting* the event. In all cases, notifications are sent only to listeners that register t receive them.

Some sources may allow only one listener to register.

. This is known as *unicasting* the event.

A source must also provide a method that allows a listener to unregister an interest in a specif type of event. The general form of such a method is this:

public void remove*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For exampl to remove a keyboard listener, you would call **removeKeyListener( )**.

The methods that add or remove listeners are provided by the source that generates event For example, the **Component** class provides methods to add and remove keyboard and mous event listeners.

Event Listeners

- Event listeners are objects that are notified as soon as a specific event occurs. Event listeners must define the methods to process the notification they are interested to receive.

The methods that receive and process events are defined in a set of interfaces, such as thos found in **java.awt.event**. For example, the **MouseMotionListener** interface defines tw methods to receive notifications when the mouse is dragged or moved. Any object may receiv and process one or both of these events if it provides an implementation of this interfac Other listener interfaces are discussed later in this and other chapters.