# Delegation Event Model in Java

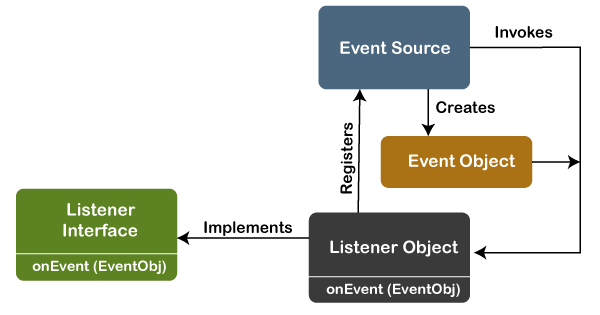
The Delegation Event model is defined to handle events in GUI [programming languages](https://www.tpointtech.com/programming-language). The [GUI](https://www.tpointtech.com/gui-full-form) stands for Graphical User Interface, where a user graphically/visually interacts with the system.

The GUI programming is inherently event-driven; whenever a user initiates an activity such as a mouse activity, clicks, scrolling, etc., each is known as an event that is mapped to a code to respond to functionality to the user. This is known as event handling.

In this section, we will discuss event processing and how to implement the delegation event model in [Java](https://www.tpointtech.com/java-tutorial). We will also discuss the different components of an Event Model.

## Event Processing in Java

Java support event processing since Java 1.0. It provides support for [AWT ( Abstract Window Toolkit)](https://www.tpointtech.com/java-awt), which is an API used to develop the Desktop application. In Java 1.0, the AWT was based on inheritance. To catch and process GUI events for a program, it should hold subclass GUI components and override action() or handleEvent() methods. The below image demonstrates the event processing.



But, the modern approach for event processing is based on the Delegation Model. It defines a standard and compatible mechanism to generate and process events. In this model, a source generates an event and forwards it to one or more listeners. The listener waits until it receives an event. Once it receives the event, it is processed by the listener and returns it. The UI elements are able to delegate the processing of an event to a separate function.

The key advantage of the Delegation Event Model is that the application logic is completely separated from the interface logic.

In this model, the listener must be connected with a source to receive the event notifications. Thus, the events will only be received by the listeners who wish to receive them. So, this approach is more convenient than the inheritance-based event model (in Java 1.0).

In the older model, an event was propagated up the containment until a component was handled. This needed components to receive events that were not processed, and it took lots of time. The Delegation Event model overcame this issue.

Basically, an Event Model is based on the following three components:

* Events
* Events Sources
* Events Listeners

### Events

The Events are the objects that define state change in a source. An event can be generated as a reaction of a user while interacting with GUI elements. Some of the event generation activities are moving the mouse pointer, clicking on a button, pressing the keyboard key, selecting an item from the list, and so on. We can also consider many other user operations as events.

The Events may also occur that may be not related to user interaction, such as a timer expires, counter exceeded, system failures, or a task is completed, etc. We can define events for any of the applied actions.

### Event Sources

A source is an object that causes and generates an event. It generates an event when the internal state of the object is changed. The sources are allowed to generate several different types of events.

A source must register a listener to receive notifications for a specific event. Each event contains its registration method. Below is an example:

1. **public** **void** addTypeListener (TypeListener e1)

From the above syntax, the Type is the name of the event, and e1 is a reference to the event listener. For example, for a keyboard event listener, the method will be called as ****addKeyListener()****. For the mouse event listener, the method will be called as ****addMouseMotionListener()****. When an event is triggered using the respected source, all the events will be notified to registered listeners and receive the event object. This process is known as event multicasting. In few cases, the event notification will only be sent to listeners that register to receive them.

Some listeners allow only one listener to register. Below is an example:

1. **public** **void** addTypeListener(TypeListener e2) **throws** java.util.TooManyListenersException

From the above syntax, the Type is the name of the event, and e2 is the event listener's reference. When the specified event occurs, it will be notified to the registered listener. This process is known as ****unicasting**** events.

A source should contain a method that unregisters a specific type of event from the listener if not needed. Below is an example of the method that will remove the event from the listener.

1. **public** **void** removeTypeListener(TypeListener e2?)

From the above syntax, the Type is an event name, and e2 is the reference of the listener. For example, to remove the keyboard listener, the ****removeKeyListener()**** method will be called.

The source provides the methods to add or remove listeners that generate the events. For example, the Component class contains the methods to operate on the different types of events, such as adding or removing them from the listener.

### Event Listeners

An event listener is an object that is invoked when an event triggers. The listeners require two things; first, it must be registered with a source; however, it can be registered with several resources to receive notification about the events. Second, it must implement the methods to receive and process the received notifications.

The methods that deal with the events are defined in a set of interfaces. These interfaces can be found in the java.awt.event package.

For example, the ****MouseMotionListener**** interface provides two methods when the mouse is dragged and moved. Any object can receive and process these events if it implements the MouseMotionListener interface.

## Types of Events

The events are categories into the following two categories:

****The Foreground Events:****

The foreground events are those events that require direct interaction of the user. These types of events are generated as a result of user interaction with the GUI component. For example, clicking on a button, mouse movement, pressing a keyboard key, selecting an option from the list, etc.

****The Background Events :****

The Background events are those events that result from the interaction of the end-user. For example, an Operating system interrupts system failure (Hardware or Software).

To handle these events, we need an event handling mechanism that provides control over the events and responses.

## The Delegation Model

The Delegation Model is available in Java since Java 1.1. it provides a new delegation-based event model using AWT to resolve the event problems. It provides a convenient mechanism to support complex Java programs.

### Design Goals

The design goals of the event delegation model are as following:

* It is easy to learn and implement
* It supports a clean separation between application and GUI code.
* It provides robust event handling program code which is less error-prone (strong compile-time checking)
* It is Flexible, can enable different types of application models for event flow and propagation.
* It enables run-time discovery of both the component-generated events as well as observable events.
* It provides support for the backward binary compatibility with the previous model.

Let's implement it with an example:

### Java Program to Implement the Event Deligation Model

The below is a Java program to handle events implementing the event deligation model:

****TestApp.java:****

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** TestApp {

**public** **void** search() {

*// For searching*

        System.out.println("Searching...");

    }

**public** **void** sort() {

*// for sorting*

        System.out.println("Sorting....");

    }

**static** **public** **void** main(String args[]) {

       TestApp app = **new** TestApp();

       GUI gui = **new** GUI(app);

    }

}

**class** Command **implements** ActionListener  {

**static** **final** **int** SEARCH = 0;

**static** **final** **int** SORT = 1;

**int** id;

    TestApp app;

**public** Command(**int** id, TestApp app) {

**this**.id = id;

**this**.app = app;

    }

**public** **void** actionPerformed(ActionEvent e) {

**switch**(id) {

**case** SEARCH:

            app.search();

**break**;

**case** SORT:

            app.sort();

**break**;

        }

    }

}

**class** GUI {

**public** GUI(TestApp app) {

        Frame f = **new** Frame();

        f.setLayout(**new** FlowLayout());

        Command searchCmd = **new** Command(Command.SEARCH, app);

        Command sortCmd = **new** Command(Command.SORT, app);

        Button b;

        f.add(b = **new** Button("Search"));

        b.addActionListener(searchCmd);

        f.add(b = **new** Button("Sort"));

        b.addActionListener(sortCmd);

        List l;

        f.add(l = **new** List());

        l.add("Alphabetical");

        l.add("Chronological");

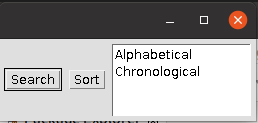
        l.addActionListener(sortCmd);

        f.pack();

        f.show();

    }

}  ****Output:****



Searching...