OOP reference material for MT3

Exception Handling

Limitations of Error Handling

In programming, traditional error handling methods, such as using conditional statements or return codes, have limitations in managing exceptions effectively. They lack centralized control, making it difficult to handle unexpected errors that can occur at runtime.

Advantages of Exception Handling

Exception handling provides a structured mechanism to manage errors and exceptional conditions during program execution. It enhances code readability by segregating error handling code from regular code. Exception handling also offers better error reporting, making it easier to trace and identify issues in the code.

Types of Errors

- **Compile-time errors:** Detected by the compiler during compilation, such as syntax errors.
- **Runtime errors:** Occur during program execution, like division by zero or null pointer exceptions.
- **Logical errors:** Flaws in the program's logic, leading to incorrect outputs despite successful compilation.

Basics of Exception Handling

In Java, exceptions are objects that represent errors or exceptional conditions that may arise during program execution. The core components of exception handling include:

- **try block:** Encloses the code that might throw exceptions.
- **catch block:** Catches and handles exceptions thrown within the try block.
- **finally block:** Executes code irrespective of whether an exception occurred or not.

```
java
try {
    // Code that might throw exceptions
    // ...
} catch (ExceptionType e) {
    // Exception handling code
    // ...
} finally {
    // Cleanup code or necessary operations
    // ...
}
```

Try Blocks, Throwing an Exception, Catching an Exception, Finally Statement

```
java
try {
    // Code that might throw exceptions
    int result = divide(10, 0); // This can throw an ArithmeticException
} catch (ArithmeticException e) {
    // Handling specific exceptions
    System.out.println("Cannot divide by zero.");
    // e.printStackTrace(); // Optionally print stack trace
} finally {
    // Cleanup or resource release code
}
```

Multi-Threading

Creating Threads

In Java, threads can be created by extending the `Thread` class or implementing the `Runnable` interface. Using the `Runnable` interface is generally preferred for better flexibility.

Life of a Thread

A thread in Java goes through several states:

- **New:** When a thread is created.
- **Runnable:** When the thread is ready to run.
- **Blocked/Waiting:** When a thread is waiting for a resource or event.
- **Terminated:** When a thread completes its task or is stopped.

Thread Life Cycle

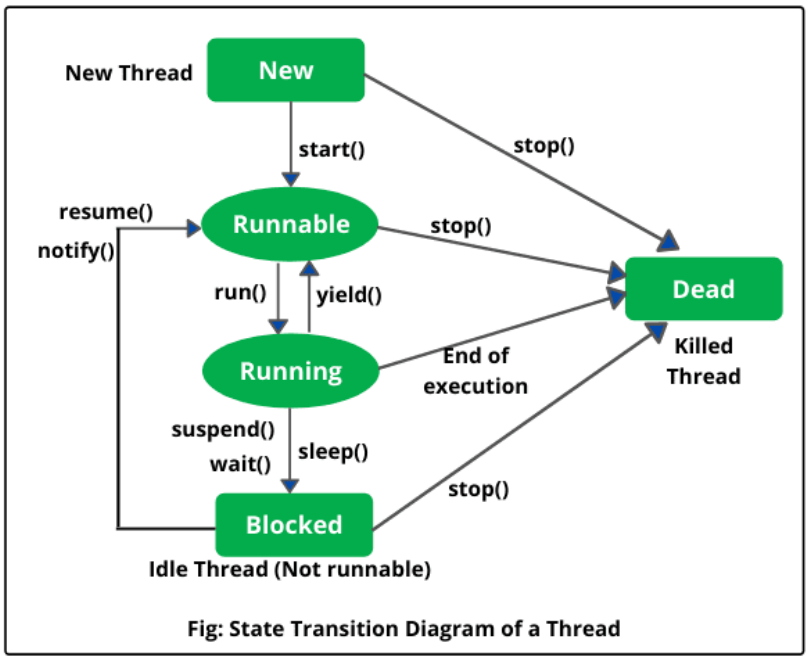


Fig: State Transition Diagram of a Thread

Importance of understanding the life cycle of Thread in Java and its states

The thread life cycle in Java is an important concept in multithreaded applications. Let's see the importance of understanding life cycle of thread in java:

1. Understanding the life cycle of a thread in Java and the states of a thread is essential because it helps identify potential issues that can arise when creating or manipulating threads.
2. It allows developers to utilize resources more effectively and prevent errors related to multiple threads accessing shared data simultaneously.
3. Knowing the thread states in Java helps predict a program's behaviour and debug any issues that may arise.
4. It also guides the developer on properly suspending, resuming, and stopping a thread as required for a specific task.

The Life Cycle of Thread in Java - Threads State

In Java, the life cycle of Thread goes through various states. These states represent different stages of execution. Here are examples of each stage of the life cycle of Thread in Java with real-life use cases:

- **New (born) state**
  - Example: Creating a new thread using the Thread class constructor.
  - Use case: Creating a new thread to perform a background task while the main Thread continues with other operations
- **Runnable state**
  - Example: After calling the start() method on a thread, it enters the runnable state.

    - Use case: Multiple threads competing for CPU time to perform their tasks concurrently.
- **Running state**
    - Example: When a thread executes its code inside the run() method.
    - Use case: A thread executing a complex computation or performing a time-consuming task.
- **Blocked state:**
    - Example: When a thread tries to access a synchronized block or method, but another thread already holds the lock.
    - Use case: Multiple threads accessing a shared resource can only be obtained by a single Thread, such as a database or a file.
- **Waiting state:**
    - Example: Using the wait method inside a synchronized block, a thread can wait until another thread calls the notify() or notifyAll() methods to wake it up.
    - Use case: Implementing the producer-consumer pattern, where a thread waits for a specific condition to be met before continuing its execution
- **Timed waiting state:**
    - Example: Using methods like sleep(milliseconds) or join(milliseconds) causes a thread to enter the timed waiting state for the specified duration.
    - Use case: Adding delays between consecutive actions or waiting for the completion of other threads before proceeding.
- **Terminated state:**
    - Example: When the run() method finishes its execution or when the stop() method is called on the Thread.
    - Use case: Completing a task or explicitly stopping a thread's execution.

Transitions Between Thread States

Threads in a multithreaded environment can transition between different thread states in Java as they execute. The Java Thread class defines these states and represents different stages in the lifecycle of a thread. The possible states include:

1. **New**: When a thread has just been created, it is in the "New" state. At this point, the Thread is not yet scheduled for execution and has not started running.
2. **Runnable**: A thread enters the "Runnable" state after invoking the start() method. In this state, the Thread is eligible to be scheduled by the operating system and can start executing its code.
3. **Blocked/Waiting**: A thread can enter the "Blocked" or "Waiting" state under certain circumstances. For example, if a thread is waiting for a lock to be released by another thread, it goes into the "Blocked" state. Similarly, if a thread waits for a specific condition to be satisfied, it enters the "Waiting" state. In these states, the Thread is not actively executing its code and is not eligible for scheduling.
4. **Timed Waiting**: Threads can also enter the "Timed Waiting" state, similar to the "Waiting" state but with a time constraint. For instance, a thread can enter the "Timed Waiting" state when it calls methods like Thread.sleep() or Object.wait() with a specific timeout value. The Thread remains in this specific state until the timeout expires or until it receives a notification from another thread.

5. **Terminated**: The final state in the thread lifecycle is the "Terminated" state. A thread enters this state when it completes its execution or when an unhandled exception occurs within the Thread. Once a thread is terminated, it cannot transition to any other state.

Example: Thread Life Cycle in Java:

Here's an example that demonstrates the life cycle of Thread in Java:

```
public class ThreadLifecycleExample {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            System.out.println("Thread is running.");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Thread is terminating.");
        });
        System.out.println("Thread is in the New state.");
        thread.start();
        System.out.println("Thread is in the Runnable state.");

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Thread is in the Timed Waiting state.");
        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Thread is in the Terminated");
```

Defining & Running Threads, Thread Methods

```
java
// Using Thread class
class MyThread extends Thread {
    public void run() {
        // Code to execute in the thread
    }
}
// Creating and starting the thread
MyThread thread = new MyThread();
thread.start();

// Using Runnable interface
class MyRunnable implements Runnable {
    public void run() {
        // Code to execute in the thread
    }
```

```
}
// Creating and starting the thread using Runnable
Thread runnableThread = new Thread(new MyRunnable());
runnableThread.start();
```

Thread Priority, Synchronization

```
java
// Setting thread priority
thread.setPriority(Thread.MAX_PRIORITY); // Highest priority

// Synchronization example
class SharedResource {
    synchronized void sharedMethod() {
        // Code that needs synchronization
    }
}
```

Implementing Runnable Interface, Thread Scheduling

```
java
// Implementing Runnable interface
class MyRunnable implements Runnable {
    public void run() {
        // Code to execute in the thread
    }
}
Thread runnableThread = new Thread(new MyRunnable());
runnableThread.start();

// Thread scheduling
Thread.yield(); // Yield the current thread's execution to other threads
```

I/O Streams

File, Streams, Advantages

In Java, a file represents a sequence of bytes stored on a storage device. Streams, such as `InputStream` and `OutputStream`, are used for reading from and writing to these files, providing a uniform interface.

The advantages of using streams include abstraction (handling I/O operations irrespective of the data source), ease of use, and efficient data processing.

The Stream Classes

- **Byte Streams (`InputStream`, `OutputStream`):** Used for handling raw binary data.
- **Character Streams (`Reader`, `Writer`):** Used for handling text-based data, automatically handling character encoding and decoding.

Byte Streams, Character Streams

```
java
// Byte Stream example
try (FileInputStream inputStream = new FileInputStream("file.txt")) {
    int data;
    while ((data = inputStream.read()) != -1) {
        // Process byte data
    }
} catch (IOException e) {
    e.printStackTrace();
}

// Character Stream example
try (FileReader reader = new FileReader("file.txt")) {
    int data;
    while ((data = reader.read()) != -1) {
        // Process character data
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

JDBC (Java Database Connectivity)

ODBC Drivers, JDBC-ODBC Bridges

- **ODBC Drivers:** Enable Java programs to interact with databases using the Open Database Connectivity (ODBC) standard.
- **JDBC-ODBC Bridges:** Allow Java applications to access ODBC databases.

Seven Steps to JDBC

1. **Importing Java SQL Packages:** `import java.sql.*;`
2. **Loading & 3. Registering the Drivers:**
   `Class.forName("com.mysql.jdbc.Driver");`

4. **Establishing Connection:**

```
            Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/database",
"username", "password");
```

5. **Creating & Executing Statements:**

```
    Statement statement = connection.createStatement();
    ResultSet resultSet = statement.executeQuery("SELECT * FROM table");
```

6. **Process the Result**

```
    while (resultSet.next()) {
        // Process each row of the ResultSet
```

```
// Example: Retrieve data using resultSet.getXXX() methods
}
```

This loop iterates through the retrieved data in the ResultSet.

7. Close the Connection and Resources

```
resultSet.close();
statement.close();
 connection.close();
```

It's essential to close the resources (ResultSet, Statement, Connection) to release them and avoid memory leaks.

Importing Java SQL Packages, Establishing Connection, Creating & Executing the Statement

```
java
// Importing Java SQL Packages
import java.sql.*;

// Establishing Connection
try {
    Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/database",
"username", "password");

    // Creating Statement and Executing
    Statement statement = connection.createStatement();
    ResultSet resultSet = statement.executeQuery("SELECT * FROM table");

    // Processing ResultSet
    while (resultSet.next()) {
        // Access data using resultSet.getXXX() methods
    }

    // Closing resources
    resultSet.close();
    statement.close();
    connection.close();
} catch (SQLException e) {
    e.printStackTrace();
}
```

In JDBC (Java Database Connectivity), there are four types of drivers used to connect Java applications with different types of databases. These drivers are known as JDBC driver types:

Type 1: JDBC-ODBC Bridge Driver

- **Description:** This driver acts as a bridge between the JDBC API and the ODBC (Open Database Connectivity) API. It translates JDBC method calls into ODBC calls.
- **Advantages:** Provides connectivity to any database for which an ODBC driver is available. Platform-independent as it uses the ODBC driver provided by the OS.
- **Disadvantages:** Performance overhead due to translation between JDBC and ODBC. Needs the ODBC driver to be installed on the client machine.

Type 2: Native-API Driver

- **Description:** Also known as the Native API driver, it uses a database-specific native API to communicate with the database.
- **Advantages:** Better performance than the Type 1 driver. It is database-specific but can be platform-independent.
- **Disadvantages:** Needs the database-specific native libraries to be installed on the client machine. Not entirely Java-based.

Type 3: Network Protocol Driver (Middleware Driver)

- **Description:** Uses a middle-tier server to translate JDBC calls into a database-independent network protocol that the middle-tier server can then translate into the database-specific protocol.
- **Advantages:** Database-independent, allowing connectivity to multiple databases. Can be platform-independent.
- **Disadvantages:** Relies on a middle-tier server which could be a bottleneck for performance.

Type 4: Thin Driver (Pure Java Driver)

- **Description:** A completely Java-based driver that communicates directly with the database using a vendor-specific protocol. It is platform-independent.
- **Advantages:** High performance, as it doesn't require an intermediate translation layer. Doesn't need native database libraries.
- **Disadvantages:** Database-specific, requiring different drivers for different databases.

These driver types vary in their architecture, performance, and dependencies. Choosing the appropriate driver type depends on factors such as performance requirements, database compatibility, and the desired level of platform independence.

AWT Components and Event Handlers

Abstract Window Toolkit (AWT)

AWT is the original Java GUI toolkit providing a set of classes for building graphical user interfaces in Java. It includes components like buttons, labels, text fields, etc.

Event Handlers and Event Listeners

- **Event Handlers:** Represent code that responds to specific events occurring during program execution.
- **Event Listeners:** Interfaces in Java that handle events triggered by user interactions with GUI components.

AWT Controls and Event Handling

AWT offers various controls:

Labels

Labels are used to display text or images. They provide information to the user but cannot be edited.

Example:

```
java
import java.awt.*;

public class LabelExample {
    public static void main(String[] args) {
        Frame frame = new Frame("Label Example");
        Label label = new Label("Hello, World!");
        frame.add(label);
        frame.setSize(300, 200);
        frame.setLayout(new FlowLayout());
        frame.setVisible(true);
    }
}
```

TextComponent

TextComponents are user-editable components, such as `TextField` and `TextArea`, used for entering and displaying text.

Example:

```
java
import java.awt.*;

public class TextFieldExample {
    public static void main(String[] args) {
        Frame frame = new Frame("TextField Example");
        TextField textField = new TextField("Type here");
        frame.add(textField);
```

```
        frame.setSize(300, 200);
        frame.setLayout(new FlowLayout());
        frame.setVisible(true);
    }
}
```

ActionEvent, Buttons

ActionEvent occurs when a button is clicked. Buttons allow users to trigger specific actions when clicked.

Example:

```
java
import java.awt.*;
import java.awt.event.*;

public class ButtonExample {
    public static void main(String[] args) {
        Frame frame = new Frame("Button Example");
        Button button = new Button("Click me");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button clicked!");
            }
        });
        frame.add(button);
        frame.setSize(300, 200);
        frame.setLayout(new FlowLayout());
        frame.setVisible(true);
    }
}
```

CheckBoxes, ItemEvent

CheckBoxes are used for selecting multiple options. ItemEvent occurs when the state of a checkbox changes.

Example:

```
java
import java.awt.*;
import java.awt.event.*;

public class CheckboxExample {
    public static void main(String[] args) {
        Frame frame = new Frame("Checkbox Example");
        Checkbox checkbox = new Checkbox("Check me");
        checkbox.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                System.out.println("Checkbox state changed");
            }
        });
        frame.add(checkbox);
```

```
        frame.setSize(300, 200);
        frame.setLayout(new FlowLayout());
        frame.setVisible(true);
    }
}
```

Choice, Scrollbars

Choice component provides a dropdown list of items. Scrollbars are used to scroll through content when it exceeds the visible area.

Example:

```
java
import java.awt.*;

public class ChoiceExample {
    public static void main(String[] args) {
        Frame frame = new Frame("Choice Example");
        Choice choice = new Choice();
        choice.add("Option 1");
        choice.add("Option 2");
        choice.add("Option 3");
        frame.add(choice);
        frame.setSize(300, 200);
        frame.setLayout(new FlowLayout());
        frame.setVisible(true);
    }
}
```

Layout Managers

Layout Managers help in arranging and positioning components within containers. Common layout managers include `FlowLayout`, `BorderLayout`, `GridLayout`, etc.

Example:

```
java
import java.awt.*;

public class LayoutExample {
    public static void main(String[] args) {
        Frame frame = new Frame("Layout Example");
        Button button1 = new Button("Button 1");
        Button button2 = new Button("Button 2");
        frame.add(button1);
        frame.add(button2);
        frame.setLayout(new FlowLayout());
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

Input Events, Menus, Programs

Input Events involve interactions like mouse clicks, key presses, etc. Menus provide options for users to perform various actions. Programs utilize AWT components and event handling to create interactive applications.

Introduction

In this article, we will discuss the Delegation Event Model in Java. In Delegation Event Model in Java, let's first discuss what you mean by Event Model.

The Event model is based on two things, i.e., Event Source and Event Listeners.

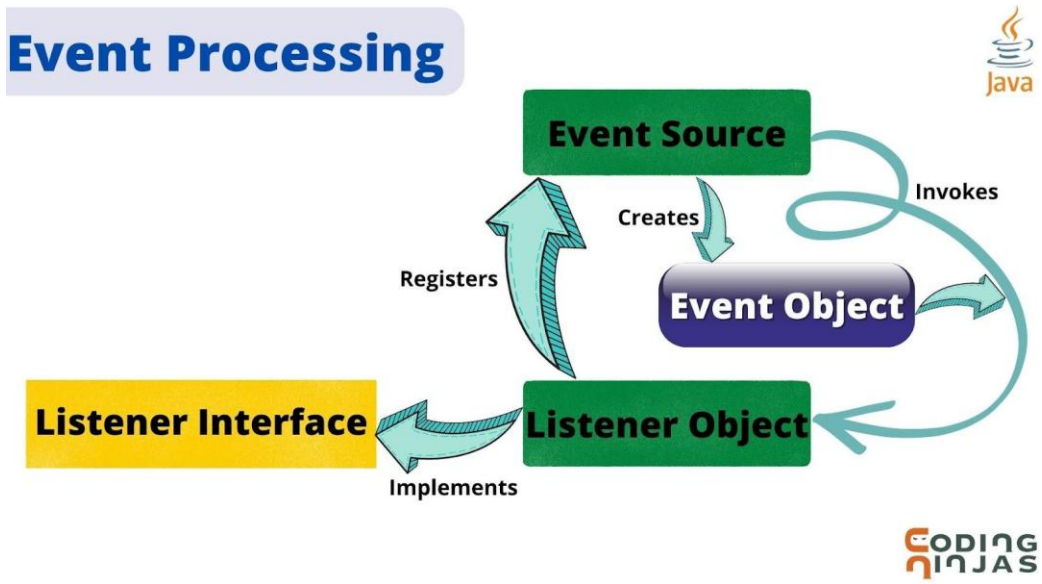
- Event Source means any object which creates the message or event.
- Event Listeners are the object which receives the message or event.

Now, coming to the Delegation Event Model in Java, the Delegation event model is based upon Event Source, Event Listeners, and Event Objects.

- Event Source is the class used to broadcast the events.
- Event Listeners are the classes that receive notifications of events.
- Event Object is the class object which describes the event.

Let's discuss how to implement the Delegation Event Model in Java.

What is delegation event model in Java?



Before discussing the Delegation Event Model in Java, let's discuss the Event Processing in Java.

In the Delegation model, a source generates an event and forwards it to one or more listeners, where the listener waits until they receive an event. Once the listener gets the event, it is processed by the listener, and then they return it. The UI elements can delegate an event's processing to a separate function.

This approach is more convenient than the event model (Java 1.0) because the events will only be received by the listeners who want to receive them in the delegation event model.

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

Registering the Source With Listener in Delegation Event Model

The different Classes provide different registration methods.

Practice by yourself on java online compiler.

Syntax:

```
addTypeListener()
```

where Type represents the Type of event.

For Example: For KeyEvent we use addKeyListener() to register, For ActionEvent we use addActionListener() to register.

Let's discuss some design goals of the event delegation model.

Event Processing in Java

Java supports event processing and supports the API used to develop the Desktop application, i.e., Abstract Window Toolkit.

Let's understand it graphically.

The delegation Model is the modern event processing approach, which has been available in Java since Java 1.1. It defines a standard and convenient mechanism to generate and process events.

Event Model includes the following three components:

- Events
- Events Sources
- Events Listeners

1. Events

An event refers to an action that occurs within a graphical user interface (GUI), such as a button click, a key press, or a mouse movement. When an event is triggered, it is typically handled by an event listener or handler, which is registered to the graphical component that generates the event.

2. Event Sources

An event source is an object that generates an event. It is typically a graphical user interface (GUI) component, such as a button, a text field, or a menu item, but it can also be other types of objects that generate events, such as timers or sockets.

When an event is triggered on an event source, the event is encapsulated in an event object and passed to all the registered event listeners or handlers. The event source is responsible for notifying the event listeners or handlers of the event and providing them with the information they need to handle the event.

3. Event Listeners

An event listener is an object that is registered to an event source and is responsible for handling events that are generated by that source. When an event occurs on the event source, the event listener is notified and performs the necessary actions to respond to the event.

An event listener in Java is typically implemented as a class that implements a specific event listener interface, such as ActionListener, MouseListener, or KeyListener. Each interface specifies a set of methods that the event listener must implement to handle the corresponding

type of event. For example, the ActionListener interface requires the implementation of a single method called actionPerformed, which is called when an action event occurs on the event source.

Types of Events

There are several types of events in the delegation event model, each with its own corresponding listener interfaces and methods for handling the events. Some common types of events in Java are:

1. The Foreground Events:

Foreground events typically refer to events that require immediate attention and are directly related to the user interaction or user interface. These events are important for the applications functioning and generally require user input or trigger visible changes in the interface of the application. Examples of foreground events include mouse movements, key presses, or button clicks.

2. The Background Events :

Background events occur in the background or behind the scenes, usually without requiring direct user interaction. These events may include tasks like network communication, data processing, or automated system operations. Background events are typically handled independently of the user interface. They may not have immediate visible effects but play a significant role in the overall functionality of an application.

The Delegation Model

The delegation model is a programming design pattern that is used to handle events and event-driven programming in graphical user interfaces (GUIs). The delegation model works by separating the concerns of event generation and event handling, allowing for greater modularity and flexibility in GUI programming.

In the delegation model, the objects responsible for generating events, such as buttons or menu items, are referred to as event sources. When an event occurs on an event source, the source creates an event object that encapsulates information about the event, such as its type and any associated data.

The delegation model has several advantages over other event-handling models. It allows for greater modularity and flexibility in GUI programming, as the event generation and event handling are separated into distinct objects. This makes it easier to change the behaviour of a GUI component without affecting other parts of the program.

Additionally, the delegation model supports multiple event listeners for a single event source, allowing for greater customization and flexibility in handling events. Overall, the delegation model is a powerful and flexible approach to handling events in GUI programming.

Delegation Event model

It has Sources and Listeners.

- **Source:** Event sources are objects that generate events. They are responsible for detecting specific occurrences or changes in the application and notifying interested listeners about these events. An event source could be a GUI component like a button, text field, or menu item, or it could be any object that generates events based on user interactions or other application-specific triggers.

- **Listeners:** Event listeners are objects that "listen" to specific events generated by event sources. When an event occurs, the corresponding listener's method(s) are invoked to handle that event.

Registering the Source With Listener

Registering the source with a listener is a crucial step in the Java Delegation Event Model. It establishes a connection between an event source and a listener, allowing the listener to receive and handle events generated by the source.

The following are the steps for registering the source with a listener:-

1. **Create the Event Listener:** Define a class that implements the appropriate listener interface for the type of event you want to handle. For example, if you're dealing with GUI events, you might implement the ActionListener interface.

2. **Implement Listener Methods:** Implement the methods specified by the listener interface within the listener class. These methods will be called when the associated event occurs.

3. **Create the Event Source:** Instantiate the event source object. This could be a GUI component like a button, text field, or menu item, depending on the type of event you're handling.

4. **Instantiate the Listener:** Create an instance of the listener class you implemented in step 1.

5. **Register the Listener with the Event Source:** Use a method provided by the event source to register the listener. This method varies depending on the event source and the type of event you're handling.

Also see, Access modifiers in java

In the next section, you will learn about the different event classes in Java.

Event Classes in Java

The following are some commonly used event classes in Java:-

Event Class	Listener Interface	Description
ActionEvent	ActionListener	Represents an action, such as a button click, triggered by a GUI component.
MouseEvent	MouseListener	Represents mouse events like clicks, enters, exits, and button presses on a GUI component.
KeyEvent	KeyListener	Represents keyboard events, such as key presses and releases, from a GUI component.
WindowEvent	WindowListener	Represents window-related events, like opening, closing, or resizing a GUI window.
FocusEvent	FocusListener	Represents focus-related events, including gaining and losing focus on a GUI component.

In the following section, we will look at a Java program that implements the event delegation model.

GUI Programming with Java (Swing)

Introduction to Swing

Swing is a part of Java's Standard Edition (SE) and provides a rich set of GUI components. It's an advanced and more flexible toolkit than AWT for creating graphical user interfaces in Java. Swing components are lightweight and platform-independent.

Limitations of AWT

- **Heavyweight Components:** AWT uses heavyweight components that are directly associated with the native operating system's GUI widgets, limiting customization and flexibility.
- **Platform Dependency:** AWT components are platform-dependent, which can cause inconsistencies across different platforms.

Swing vs. AWT

- **Swing:** Provides lightweight components that are painted by Java, making them consistent across different platforms. Offers more extensive and customizable components compared to AWT.
- **AWT:** Uses heavyweight components directly from the native operating system.

MVC Architecture

- **Model-View-Controller (MVC):** A design pattern that separates an application into three components: Model (data), View (user interface), and Controller (handles user input).

Hierarchy for Swing Components

Swing components form a hierarchy:

- **JFrame:** A top-level window used to create the main window of a Java application.
- **JApplet:** A window with an embedded applet for running applets within a browser or applet viewer.
- **JDialog:** A dialog box that pops up to interact with the user.
- **JPanel:** A lightweight container that holds other components.

Overview of Some Swing Components

JButton

```
java
import javax.swing.*;

public class JButtonExample {
    public static void main(String[] args) {
```

```
        JFrame frame = new JFrame("Button Example");
        JButton button = new JButton("Click Me");
        frame.add(button);
        frame.setSize(300, 200);
        frame.setLayout(null);
        frame.setVisible(true);
    }
}
```

JLabel

```
java
import javax.swing.*;

public class JLabelExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Label Example");
        JLabel label = new JLabel("Hello, Swing!");
        frame.add(label);
        frame.setSize(300, 200);
        frame.setLayout(null);
        frame.setVisible(true);
    }
}
```

JTextField and JTextArea

```
java
import javax.swing.*;

public class JTextFieldTextAreaExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("TextField and TextArea Example");
        JTextField textField = new JTextField("Enter text here");
        JTextArea textArea = new JTextArea("Enter text here");
        frame.add(textField);
        frame.add(textArea);
        frame.setSize(300, 200);
        frame.setLayout(new FlowLayout());
        frame.setVisible(true);
    }
}
```

Simple Swing Applications

Creating a simple Swing application involves creating a JFrame and adding various Swing components to it. For instance, a basic login form might contain JLabels, JTextFields, JButtons, etc., arranged using layout managers like BorderLayout, GridLayout, or FlowLayout.
