

UNIT 4

Java Packages: I/O classes: Byte Stream and Character Stream, Predefined Stream, reading console input, writing console output. Applet: Applet Life Cycle, creating an applet, Using image and sound in applet. Lang: Various classes, Importance class Definition, Util: Framework, Event Model, Scanner Class AWT: Exploring AWT, Event handling – The delegation-event model, Event classes. Source of event, Event listener interfaces, handling mouse and keyboard event, Adapter class.

Types of Streams

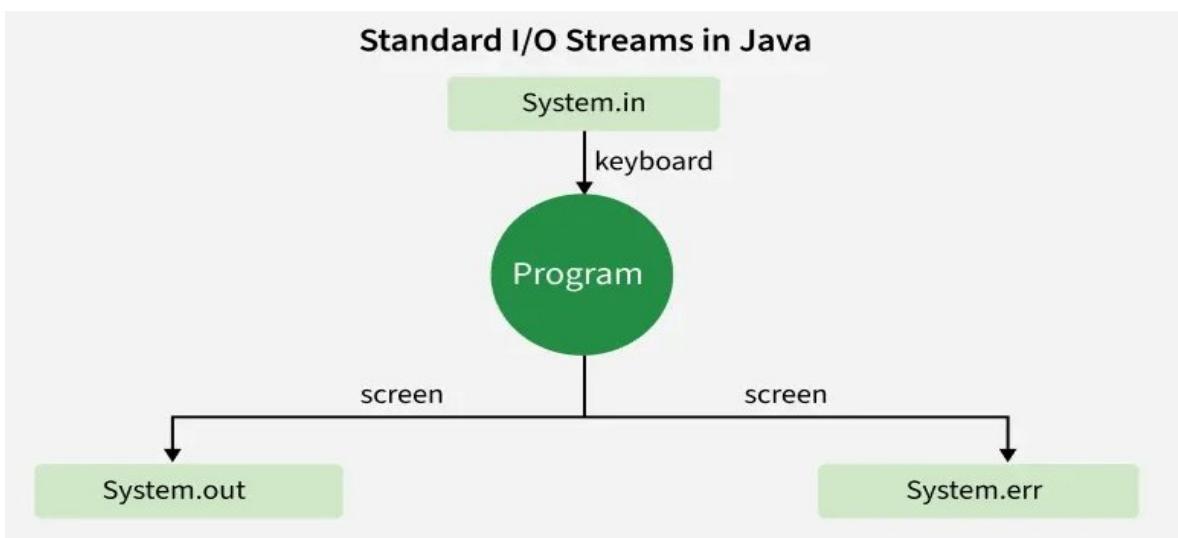
Depending on the type of operations, streams can be divided into two primary classes:

1. Input Stream: These streams are used to read data that must be taken as an input from a source array or file or any peripheral device. For eg., FileInputStream, BufferedInputStream, ByteArrayInputStream etc.



InputStream

2. Output Stream: These streams are used to write data as outputs into an array or file or any output peripheral device. For eg., FileOutputStream, BufferedOutputStream, ByteArrayOutputStream etc.

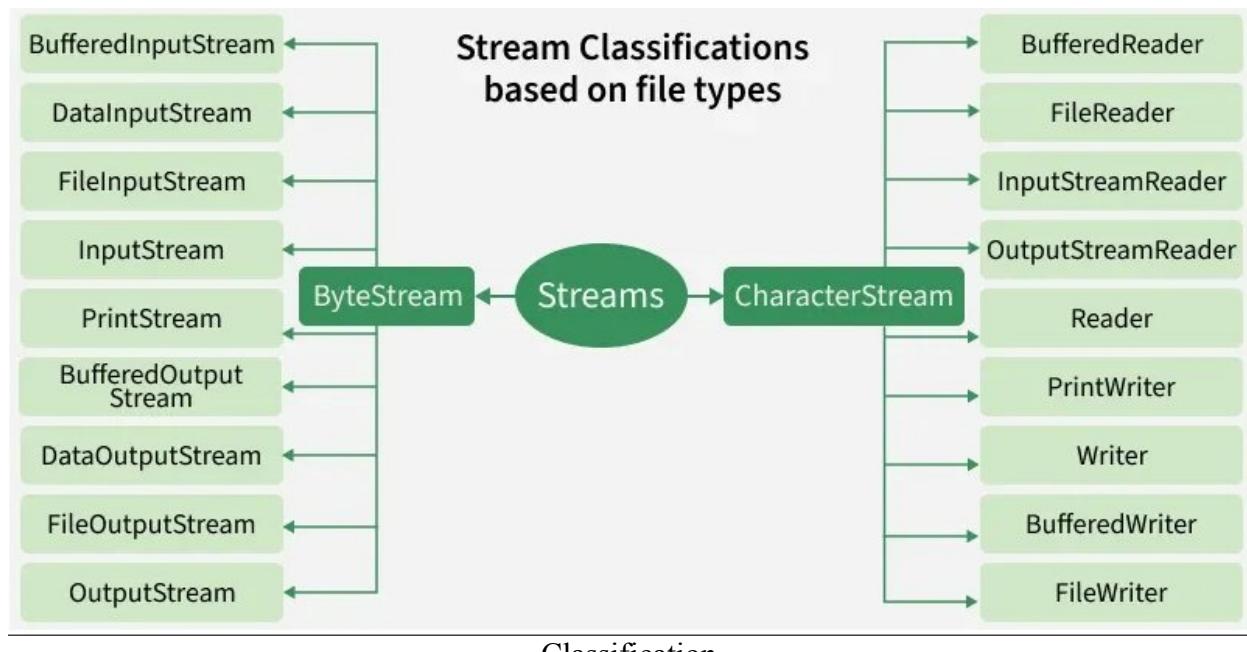


Output Stream

To know more about types of Streams, you can refer to: [Java.io.InputStream Class](#) & [Java.io.OutputStream class](#).

Types of Streams Depending on the Types of File

Depending on the types of file, Streams can be divided into two primary classes which can be further divided into other classes as can be seen through the diagram below followed by the explanations.



1. **ByteStream:**

Byte streams in Java are used to perform input and output of 8-bit bytes. They are suitable for handling raw binary data such as images, audio, and video, using classes like `InputStream` and `OutputStream`.

Here is the list of various `ByteStream` Classes:

Stream class	Description
BufferedInputStream	Used to read data more efficiently with buffering.
DataInputStream	Provides methods to read Java primitive data types.
FileInputStream	This is used to read from a file.

Stream class	Description
<u>InputStream</u>	This is an abstract class that describes stream input.
<u>PrintStream</u>	This contains the most used print() and println() method
<u>BufferedOutputStream</u>	This is used for Buffered Output Stream.
<u>DataOutputStream</u>	This contains method for writing java standard data types.
<u>FileOutputStream</u>	This is used to write to a file.
<u>OutputStream</u>	This is an abstract class that describes stream output.

Example: Java Program illustrating the Byte Stream to copy contents of one file to another file.

```
import java.io.*;
public class Geeks {
    public static void main(
        String[] args) throws IOException
    {

        FileInputStream sourceStream = null;
        FileOutputStream targetStream = null;

        try {
            sourceStream
                = new FileInputStream("sorcefile.txt");
            targetStream
                = new FileOutputStream("targetfile.txt");

            // Reading source file and writing content to target file byte by byte
            int temp;
            while ((temp =
                    sourceStream.read()) != -1)
                targetStream.write((byte)temp);
        }
        finally {
            if (sourceStream != null)
                sourceStream.close();
            if (targetStream != null)
                targetStream.close();
        }
    }
}
```

```
    }
}
}
```

Output:

Shows contents of file test.txt

2. Character Stream:

Character streams in Java are used to perform input and output of 16-bit Unicode characters. They are best suited for handling text data, using classes like Reader and Writer which automatically handle character encoding and decoding.

Here is the list of various CharacterStream Classes:

Stream class	Description
<u>BufferedReader</u>	It is used to handle buffered input stream.
<u>FileReader</u>	This is an input stream that reads from file.
<u>InputStreamReader</u>	This input stream is used to translate byte to character.
<u>OutputStreamWriter</u>	Converts character stream to byte stream.
<u>Reader</u>	This is an abstract class that define character stream input.
<u>PrintWriter</u>	This contains the most used print() and println() method
<u>Writer</u>	This is an abstract class that define character stream output.
<u>BufferedWriter</u>	This is used to handle buffered output stream.
<u>FileWriter</u>	This is used to output stream that writes to file.

Example:

```
import java.io.*;

public class Geeks
{
    public static void main(String[] args) throws IOException
    {
        FileReader sourceStream = null;
```

```

try {
    sourceStream = new FileReader("test.txt");

    // Reading sourcefile and writing content to target file character by character.
    int temp;

    while ((temp = sourceStream.read()) != -1)
        System.out.println((char)temp);
}
finally {
    // Closing stream as no longer in use
    if (sourceStream != null)
        sourceStream.close();
}
}
}

```

Reading console input in java

In java, there are three ways to read console input. Using the 3 following ways, we can read input data from the console.

- Using BufferedReader class
- Using Scanner class
- Using Console class

Let's explore the each method to read data with example.

1. Reading console input using BufferedReader class in java

Reading input data using the **BufferedReader** class is the traditional technique. This way of the reading method is used by wrapping the **System.in** (standard input stream) in an **InputStreamReader** which is wrapped in a **BufferedReader**, we can read input from the console.

The **BufferedReader** class has defined in the **java.io** package.

Consider the following example code to understand how to read console input using BufferedReader class.

Example

```
import java.io.*;
```

```
public class ReadingDemo {  
  
    public static void main(String[] args) throws IOException {  
  
        BufferedReader in = new BufferedReader(new  
InputStreamReader(System.in));  
  
        String name = "";  
  
        try {  
            System.out.print("Please enter your name : ");  
            name = in.readLine();  
            System.out.println("Hello, " + name + "!");  
        }  
        catch(Exception e) {  
            System.out.println(e);  
        }  
        finally {  
            in.close();  
        }  
    }  
}
```

When we run the above program, it produce the following output.

2. Reading console input using Scanner class in java

Reading input data using the **Scanner** class is the most commonly used method. This way of the reading method is used by wrapping the **System.in** (standard input stream) which is wrapped in a **Scanner**, we can read input from the console.

The **Scanner** class has defined in the **java.util** package.

Consider the following example code to understand how to read console input using Scanner class.

Example

```
import java.util.Scanner;

public class ReadingDemo {

    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);

        String name = "";
        System.out.print("Please enter your name : ");
        name = in.next();
        System.out.println("Hello, " + name + "!");

    }
}
```

3. Reading console input using Console class in java

Reading input data using the **Console** class is the most commonly used method. This class was introduced in Java 1.6 version.

The **Console** class has defined in the **java.io** package.

Consider the following example code to understand how to read console input using Console class.

Example

```
import java.io.*;

public class ReadingDemo {

    public static void main(String[] args) {

        String name;
        Console con = System.console();

        if(con != null) {
            name = con.readLine("Please enter your name : ");
            System.out.println("Hello, " + name + "!!");
        }
        else {
            System.out.println("Console not available.");
        }
    }
}
```

Writing console output in java

In java, there are two methods to write console output. Using the 2 following methods, we can write output data to the console.

- Using print() and println() methods
- Using write() method

Let's explore the each method to write data with example.

1. Writing console output using print() and println() methods

The PrintStream is a built-in class that provides two methods print() and println() to write console output. The print() and println() methods are the most widely used methods for console output.

Both print() and println() methods are used with System.out stream.

The print() method writes console output in the same line. This method can be used with console output only.

The println() method writes console output in a separate line (new line). This method can be used with console and also with other output sources.

Let's look at the following code to illustrate print() and println() methods.

Example

```
public class WritingDemo {  
    public static void main(String[] args) {  
        int[] list = new int[5];  
  
        for(int i = 0; i < 5; i++)  
            list[i] = i*10;  
        for(int i:list)  
            System.out.print(i);           //prints in same line  
  
        System.out.println("");  
        for(int i:list)  
            System.out.println(i);       //Prints in separate lines  
  
    }  
}
```

2. Writing console output using write() method

Alternatively, the PrintStream class provides a method write() to write console output.

The write() method takes integer as argument, and writes its ASCII equivalent character on to the console, it also accepts escape sequences.

Let's look at the following code to illustrate write() method.

Example

```
public class WritingDemo {
```

```
public static void main(String[] args) {  
  
    int[] list = new int[26];  
  
    for(int i = 0; i < 26; i++) {  
        list[i] = i + 65;  
    }  
  
    for(int i:list) {  
        System.out.write(i);  
        System.out.write('\n');  
    }  
}
```

Applet

Introduction

An Applet is the special type of Java program that is run on web browser. The Applet class provides the standard interface between applet and browser.

An applet class does not have main method and it extends the **java.applet.Applet** class. An applet program runs through **applet viewer**.

Advantages of Applets

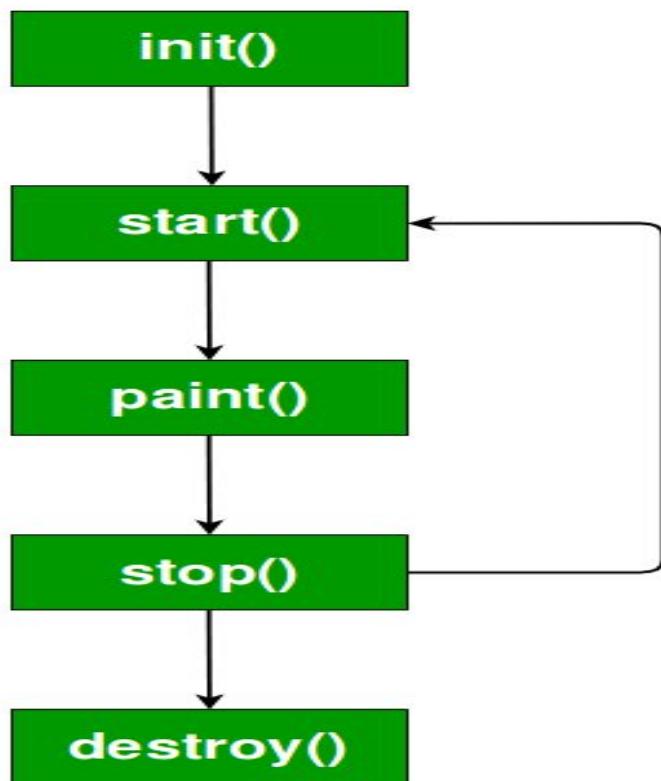
- Applets are supported by most web browser.
- Applets have very less response time, because it works at client side.
- Applets are quite secure because of their access of resources.
- An untrusted applet has no access to local machine.

- The size of applets is small, making it easier to transfer over network.

Limitations of Applets

- Applets require a Java plug-in to execute.
- Applets do not support file system.
- Applet itself cannot run or modify any application on the local system.
- It cannot work with native methods of system.
- An applet cannot access the client-side resources.

Applet Life Cycle



It is derived from the Applet class. When an Applet is created, it goes through different stages; it is known as applet life cycle.

1. Born or initialization state
2. Running state
3. Stopped state

4. Destroyed state

5. Display state

Applet Life Cycle Methods

1. init()

The init() method is responsible for applet initialization when it is loaded. It is called by browser or applet viewer only once.

Syntax

```
public void init()
{
    //actions
}
```

2. start()

It is invoked after the initialization state and applet entered into running state. It starts the execution of Applet and the applet becomes an active state.

Syntax

```
public void start()
{
    //actions
}
```

3. paint()

This method is used to display output on screen. It takes java.awt.Graphics object as a parameter. It helps to create Applet GUI such as a colored background, drawing etc.

Syntax

```
public void paint(Graphics g)
{
```

```
//Display statement  
}
```

4. stop()

This method is responsible to stop the execution and applet becomes temporarily inactive state. It is invoked when Applet is stopped.

Syntax

```
public void stop()  
{  
    //actions  
}
```

5. destroy()

It destroys and removes the applet from the memory. It is invoked only once. This is end of the life cycle of applet and it becomes dead.

Syntax

```
public void destroy()  
{  
    //actions  
}
```

Developing an Applet Program

Steps to develop an applet program

1. Create an applet (java) code i.e. .java file.
2. Compile to generate .class file.
3. Design a HTML file with <APPLET> tag.

Running the Applet Program

There are two ways to run the applet program

1. By HTML

2. Appletviewer tool

Example : Getting a message printed through Applet

```
import java.awt.*;
import java.applet.*;
public class AppletDemo
{
    public void paint(Graphics g)
    {
        g.drawString("Welcome to TutorialRide", 50, 50);
    }
}
```

1. Save the above program: **AppletDemo.java**

2. Compile: > javac AppletDemo.java

//AppletDemo.html

```
<html>
<head>
    <title> AppletDemo </title>
</head>
<body>
    <applet code="AppletDemo.class" width="200" height="250">
    </applet>
</body>
</html>
```

Run the complete program:

> appletviewer AppletDemo.html

NOTE: To execute the applet program through **appletviewer** tool, write it on command prompt.

```
C :\> javac AppletDemo.java  
C :\> appletviewer AppletDemo.java
```

Displaying Images in Java Applets

Java Applets can display images using the `Image` class and the `paint()` method. Here's how it works:

- **Steps to Display an Image:**

1. Load the image using the `getImage()` method of the `Applet` class.
2. Use the `paint()` method to draw the image on the applet window using the `drawImage()` method of the `Graphics` class.

- **Code Example:**

```
Copy code  
import java.applet.Applet;  
import java.awt.*;  
  
public class ImageApplet extends Applet {  
    Image img;  
  
    public void init() {  
        img = getImage(getDocumentBase(), "example.jpg"); // Load image  
    }  
  
    public void paint(Graphics g) {  
        g.drawImage(img, 50, 50, this); // Draw image at (50, 50)  
    }  
}
```

Playing Audio in Java Applets

Java Applets can play audio files using the `AudioClip` interface. The audio files must be in `.au` format or other supported formats.

- **Steps to Play Audio:**

1. Load the audio file using the `getAudioClip()` method of the `Applet` class.
2. Use methods like `play()`, `loop()`, or `stop()` to control the audio playback.

- **Code Example:**

```
Copy code  
import java.applet.Applet;  
import java.applet.AudioClip;  
  
public class AudioApplet extends Applet {  
    AudioClip audio;
```

```
public void init() {
    audio = getAudioClip(getDocumentBase(), "example.au"); // Load audio
}

public void start() {
    audio.play(); // Play audio once
}

public void stop() {
    audio.stop(); // Stop audio
}
```

Lang Package (java.lang)

Overview of the java.lang Package

The `java.lang` package is a fundamental part of the Java programming language, providing essential classes and interfaces that are automatically imported into every Java program. It includes classes for basic data types, mathematical operations, threading, string manipulation, and system-level operations.

Key Classes in `java.lang`

1. **Object**: The root of the class hierarchy. Every class in Java implicitly extends `Object`, which provides methods like `equals()`, `hashCode()`, and `toString()`.
2. **String**: Represents immutable sequences of characters. It is widely used for text manipulation and includes methods like `substring()`, `charAt()`, and `concat()`.
3. **StringBuilder and StringBuffer**: Mutable alternatives to `String`. `StringBuilder` is faster but not thread-safe, while `StringBuffer` is thread-safe.
4. **Math and StrictMath**: Provide methods for mathematical operations such as `sqrt()`, `log()`, and trigonometric functions.
5. **Wrapper Classes**: Include `Integer`, `Double`, `Boolean`, and others, which wrap primitive types into objects. These classes also provide utility methods for type conversion and parsing.
6. **Thread and ThreadGroup**: Facilitate multithreading by allowing the creation and management of threads. `ThreadGroup` organizes threads into groups for easier management.

7. **Runtime**: Provides methods to interact with the Java runtime environment, such as memory management and process execution.
8. **Class and ClassLoader**: Represent classes and interfaces at runtime. ClassLoader is responsible for dynamically loading classes.
9. **Throwable**: The superclass for all exceptions and errors in Java. Subclasses include Exception and Error.

Common Use Cases

- **Mathematical Operations**: Use Math or StrictMath for calculations like logarithms, trigonometry, and rounding.
- **String Manipulation**: Use String, StringBuilder, or StringBuffer for handling text data.
- **Threading**: Use Thread and related classes for concurrent programming.
- **System Operations**: Use System and Runtime for tasks like reading environment variables or executing external processes.

java.util package

The java.util package in Java is a fundamental part of the Java Standard Library, providing a collection of utility classes that support various data structures, algorithms, and other functionalities. This package is crucial for efficient data management and manipulation in Java applications. Here's a comprehensive overview of its key components:

1. Collections Framework

The java.util package includes the core interfaces and classes of the Collections Framework, which is used to store and manipulate groups of objects. The primary interfaces are:

- **Collection**: The root interface of the collection hierarchy. It represents a group of objects, known as elements. Common methods include add, remove, and size.
- **List**: Extends Collection and represents an ordered collection. Implementations like ArrayList and LinkedList provide different performance characteristics for list operations.
- **Set**: Extends Collection and represents a collection that does not allow duplicate elements. Implementations like HashSet and TreeSet offer different approaches to storing unique elements.
- **Queue**: Extends Collection and represents a collection designed for holding elements prior to processing. Implementations such as LinkedList and PriorityQueue support queue operations.
- **Map**: Represents a collection of key-value pairs. It does not extend Collection but is a part of the collections framework. Implementations like HashMap, TreeMap, and LinkedHashMap provide different ways to handle mappings between keys and values.

The Collection Framework includes several important classes that implement these interfaces, providing various data structures and functionalities.

1. **ArrayList**: A resizable array implementation of the List interface. It provides fast random access to elements and is suitable for scenarios where frequent retrievals are required.
2. **LinkedList**: Implements both the List and Deque interfaces, providing a doubly linked list structure. It is efficient for insertions and deletions but slower for random access compared to ArrayList.
3. **HashSet**: A hash table-based implementation of the Set interface that does not allow duplicate elements. It offers constant time complexity for basic operations like add, remove, and contains.
4. **TreeSet**: Implements the Set interface and is backed by a Red-Black tree. It maintains elements in a sorted order and allows efficient querying of range-based operations.
5. **PriorityQueue**: An implementation of the Queue interface that uses a priority heap. It orders elements based on their natural ordering or a specified comparator.
6. **HashMap**: Implements the Map interface using a hash table. It allows null values and keys, providing constant time complexity for get and put operations.
7. **TreeMap**: A Red-Black tree-based implementation of the Map interface that maintains keys in a sorted order. It provides a range of operations for dealing with sorted keys.

2. Collections Utility class Methods

The Collections class provides static methods for working with collections. Key methods include:

- **sort(List<T> list)**: Sorts the specified list into ascending order.
- **shuffle(List<?> list)**: Randomly permutes the elements of the specified list.
- **reverse(List<?> list)**: Reverses the order of the elements in the specified list.
- **max(Collection<? extends T> coll)**: Returns the maximum element of the given collection, according to the natural ordering of its elements.

Java Event Model

Introduction

The **Event Delegation Model** (or simply **Event Model**) in Java is part of the **Abstract Window Toolkit (AWT)** and **Swing**.

It provides a mechanism for handling events generated by GUI components such as buttons, text fields, checkboxes, etc.

When a user interacts with a GUI component (like clicking a button or pressing a key), an **event** is generated.

This event must be **handled** by some object that can respond appropriately.

Basic Concepts

Event

An **event** is an object that describes a change of state in a source component.

Examples of events:

- Mouse click
- Key press
- Button click
- Window closing
- Text change in a text field

Each event is an instance of a class that inherits from `java.util.EventObject`.

Event Source

The **source** is the object on which the event occurs.

Example:

- A Button is the source of an ActionEvent.
- A TextField is the source of a TextEvent.

Event Listener

A **listener** is an object that is notified when an event occurs.

Listeners implement specific **event listener interfaces** that define one or more methods.

Example:

- ActionListener for button clicks
- MouseListener for mouse events
- KeyListener for keyboard input

When an event occurs on the source, the listener's method is automatically called.

Delegation Event Model

Definition

The **Delegation Event Model** is the mechanism by which events are handled in Java.

In this model:

- The **event source** generates an event.
- The **event listener** receives and handles it.
- The source **delegates** the handling of the event to the listener.

Diagram (Conceptual)

User Action → Event Source → Event Object → Event Listener
(Click) (Button) (ActionEvent) (ActionListener)

Steps to Handle an Event

1. **Implement the listener interface**
 - Create a class that implements an appropriate listener interface (e.g., ActionListener).
2. **Register the listener with the event source**
 - Use a method like addActionListener().
3. **Override the listener method**
 - Define what should happen when the event occurs.

Example: Button Click Event

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

class ButtonExample extends Frame implements ActionListener {
    Button b;

    ButtonExample() {
        b = new Button("Click Me");
        b.setBounds(100, 100, 80, 30);
        b.addActionListener(this); // Step 2: Register listener
        add(b);

        setSize(300, 300);
        setLayout(null);
    }

    public void actionPerformed(ActionEvent e) {
        System.out.println("Button Clicked!");
    }
}
```

```

        setVisible(true);
    }

// Step 3: Implement the listener method
public void actionPerformed(ActionEvent e) {
    System.out.println("Button Clicked!");
}

public static void main(String[] args) {
    new ButtonExample(); // Step 1: Create frame
}
}

```

Explanation:

- Button → Event Source
- ActionEvent → Event Object
- ActionListener → Event Listener
- addActionListener(this) → Registers the listener

Event Classes

All event classes inherit from `java.util.EventObject`.
 Most GUI-related events extend from `java.awt.AWTEvent`.

Event Class	Description
ActionEvent	Generated by buttons, menu items, etc.
MouseEvent	Generated by mouse actions (click, press, move)
KeyEvent	Generated by keyboard actions
ItemEvent	Generated by selecting/deselecting items (like checkbox)
TextEvent	Generated when text in a text field changes
WindowEvent	Generated by window actions (open, close, minimize)
FocusEvent	Generated when a component gains or loses focus

Event Listener Interfaces

Each event type has a corresponding listener interface.

Listener Interface	Methods	Used For
ActionListener	actionPerformed(ActionEvent e)	Buttons, menu items
ItemListener	itemStateChanged(ItemEvent e)	Checkboxes, lists
MouseListener	mouseClicked(), mousePressed(), mouseReleased(), mouseEntered(), mouseExited()	Mouse actions
MouseMotionListener	mouseDragged(), mouseMoved()	Mouse movement
KeyListener	keyPressed(), keyReleased(), keyTyped()	Keyboard input
WindowListener	windowOpened(), windowClosing(), etc.	Window events
FocusListener	focusGained(), focusLost()	Focus change
TextListener	textValueChanged(TextEvent e)	Text field changes

Adapter Classes

For interfaces with **multiple methods**, implementing all methods can be tedious. To make it easier, Java provides **adapter classes** with **empty implementations** of all interface methods.

You can **extend an adapter class** and override only the methods you need.

Interface	Adapter Class
MouseListener	MouseAdapter
KeyListener	KeyAdapter
WindowListener	WindowAdapter
FocusListener	FocusAdapter

MouseListener	MouseAdapter
KeyListener	KeyAdapter
WindowListener	WindowAdapter
FocusListener	FocusAdapter

Example:

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

class MouseExample extends Frame {
    MouseExample() {
        addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                System.out.println("Mouse clicked at " + e.getX() + ", " + e.getY());
            }
        });
        setSize(300, 300);
        setVisible(true);
    }

    public static void main(String[] args) {
        new MouseExample();
    }
}
```

Handling Multiple Listeners

Multiple listeners can be registered to a single event source.
Each listener will be notified in the order of registration.

Example:

```
button.addActionListener(listener1);
button.addActionListener(listener2);
```

Custom Events

You can also create **user-defined (custom)** events by extending EventObject and defining your own listener interface.

Example:

```
class MyEvent extends EventObject {  
    MyEvent(Object source) {  
        super(source);  
    }  
}
```