

LIVE 

Advance Java Seminar

Date – 10th & 11th Nov 2023

Organized By :-

UR ENGINEERING FRIEND

#1 stop Destination for Diploma & Degree

Day 1

1) Abstract Windowing Toolkit

- What is AWT ?
- How to create Frames ?
- Adding controls to frame
- Applets and its controls
- Layout manager
- Menubars & menus
- Dialog Box

2) Swing

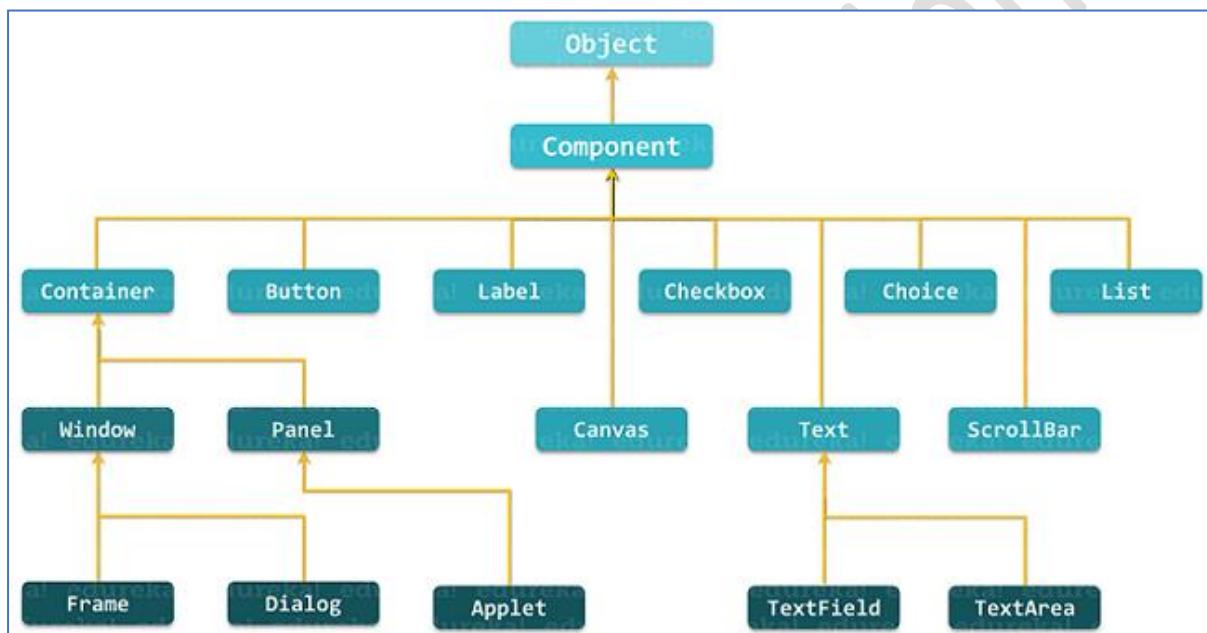
- What is Swing ?
- How to create Frames In swing ?
- Swing Advanced controls
- MVC Architecture

3) Event Handling

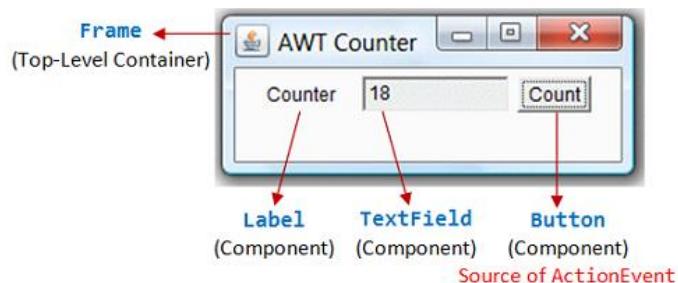
- What is an Event ?
- Event delegation model
- Event sources
- Event Listeners
- Event Classes
- Mouse Event
- Key Events
- Adapter Class

1) Abstract Windowing Toolkit

AWT stands for Abstract Window Toolkit, and it is a set of classes and APIs (Application Programming Interfaces) provided by Java for creating graphical user interfaces (GUIs) in Java applications. AWT was one of the first GUI frameworks in Java and is part of the Java Foundation Classes (JFC).



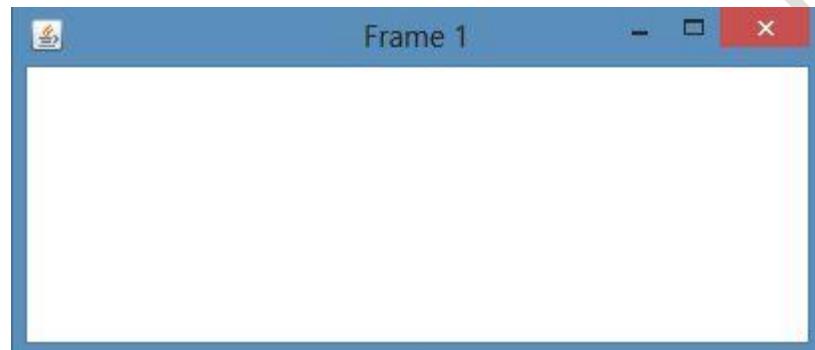
AWT Controls :-



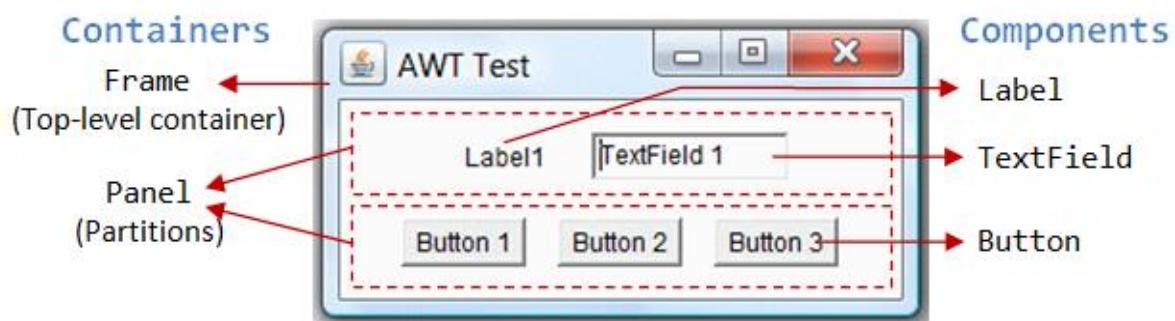
How to Create Container ? (Frame)

By extending the
frame class

Using instance of
frame class



Adding Controls to Frame



Applets

In Java, an applet is a small Java program that is embedded and run within a web page. Applets were a popular way to add interactive content to web pages in the early days of the internet. They provided a way to create graphical user interfaces (GUIs) and perform various tasks within a web browser.

Key features of Java applets include:

- 1. Platform Independence:** Like other Java programs, applets are platform-independent, meaning they can run on any system that has a Java Virtual Machine (JVM) installed.
- 2. Embedding in HTML:** Applets are typically embedded in HTML (Hypertext Markup Language) pages using the `<applet>` tag. The HTML page acts as a container for the applet.
- 3. Graphics and User Interface:** Applets can create graphical user interfaces using the Abstract Window Toolkit (AWT) or Swing components. They can draw graphics, handle user input, and respond to events.
- 4. Security Restrictions:** Applets run in a restricted environment known as the "sandbox" to prevent potentially harmful actions. They have limited access to the local system to ensure security.
- 5. Lifecycle Methods:** Applets have lifecycle methods such as `init()`, `start()`, `stop()`, and `destroy()`, which are called at different stages of the applet's life. These methods allow developers to initialize resources, start and stop animations, and perform cleanup tasks.

How to create an Applet ?

```
import java.applet.Applet;
import java.awt.Graphics;

public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello, this is a simple applet!", 20, 20);
    }
}
```

Adding Controls to Applet

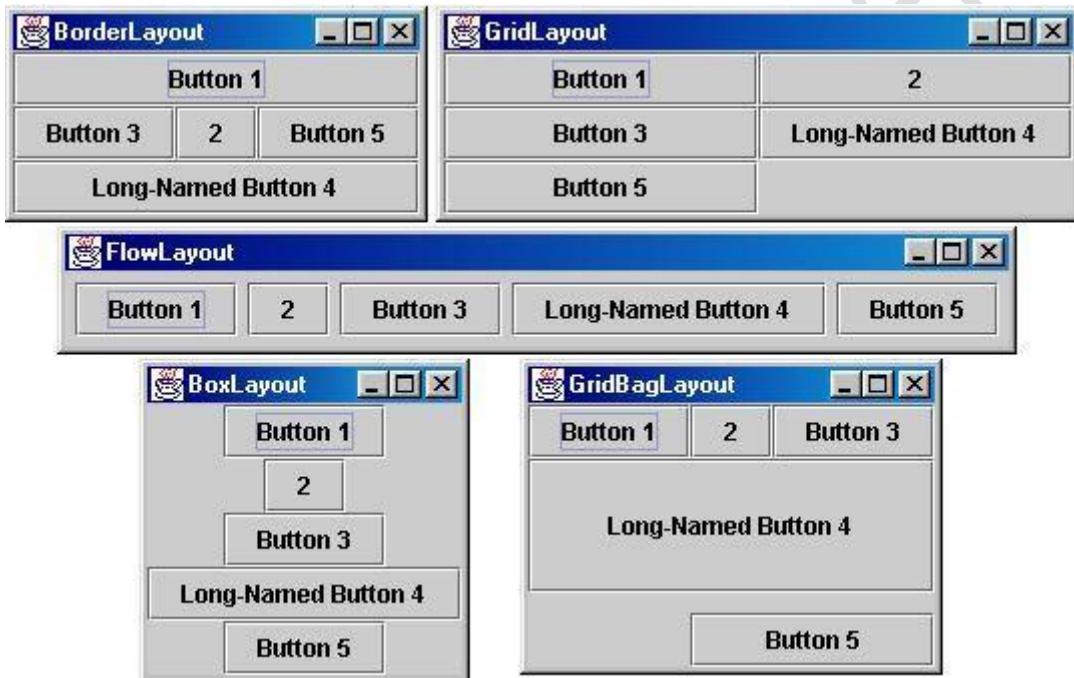
```
public void init() {
    // Create a Button
    clickButton = new Button("Click me");

    // Add an ActionListener to the Button
    clickButton.addActionListener(this);

    // Add the Button to the applet
    add(clickButton);
}
```

Layout Managers

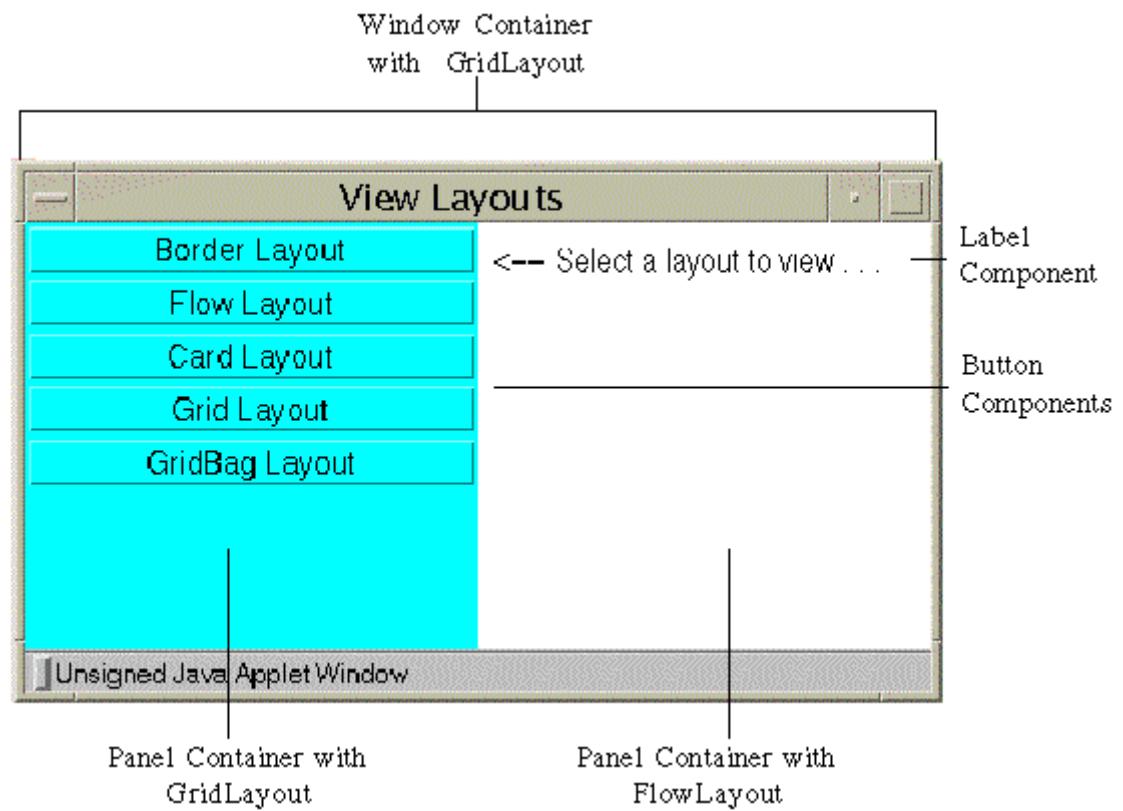
In AWT (Abstract Window Toolkit) in Java, layout managers are objects that control the placement and sizing of components within a container. A container is a component that can hold and organize other components, such as panels, frames, and applets. Layout managers are used to define how components are arranged and resized when the container is resized.



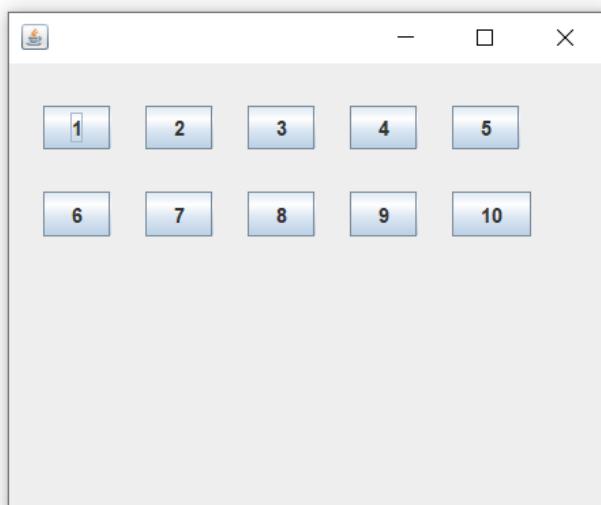
- Layout managers simplify the process of designing user interfaces by handling the details of component placement and resizing. Choosing the appropriate layout manager depends on the desired structure and organization of components within the container.
- Developers can also combine multiple layout managers within nested containers to achieve more complex layouts.

AWT provides several layout managers to assist with the organization of components.

The main AWT layout managers include:



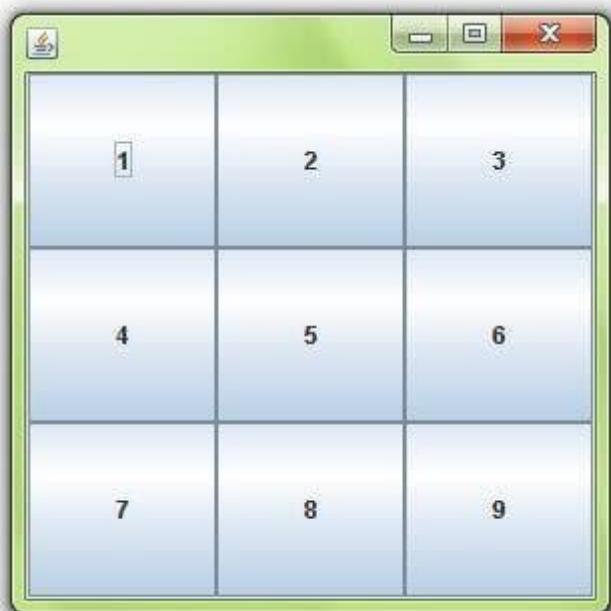
a) Flow Layout



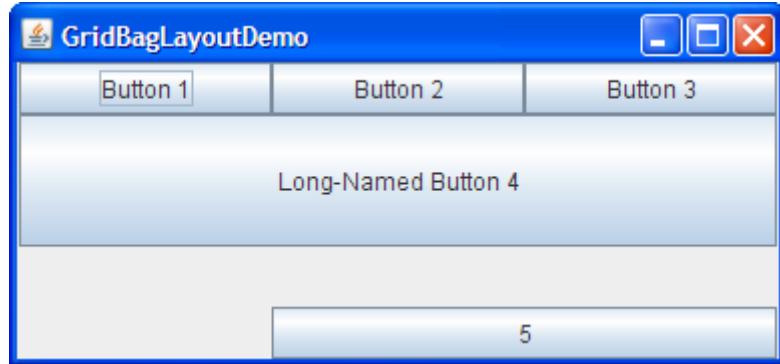
b) Border Layout



c) Grid Layout



d) Gridbag Layout



e) Card Layout



Menu Bars in AWT

In AWT (Abstract Window Toolkit) in Java, the primary class used to create menu bars is `MenuBar`.

- The `MenuBar` class provides constructors to create instances of menu bars. Additionally, you'll use other classes such as `Menu` and `MenuItem` to construct the actual menus and menu items within the menu bar.
- **Below are the constructors for creating `MenuBar`:**

1. Default Constructor:

- The default constructor creates an empty menu bar.

```
MenuBar menuBar = newMenuBar();
```

2. None-Default Constructor:

- You can create a `MenuBar` with an initial set of menus using the constructor that takes a `Menu` object as a parameter.

```
MenuBar menuBar = newMenuBar();
```

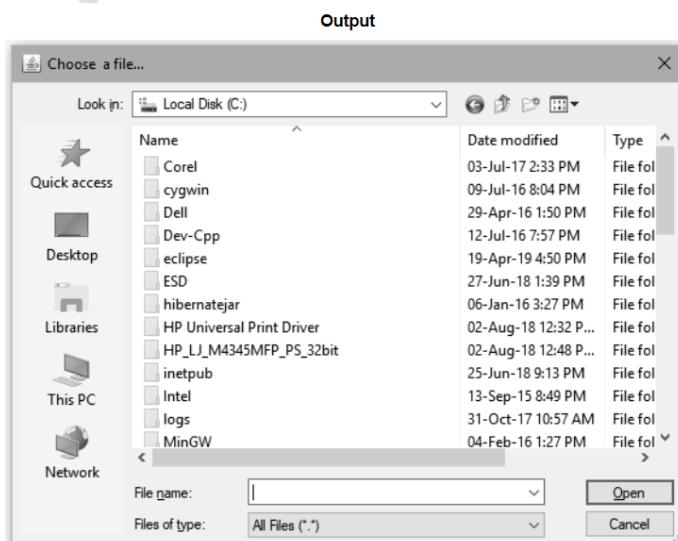
```
Menu fileMenu = new Menu("File");
```

```
Menu editMenu = new Menu("Edit");
```

```
menuBar.add(fileMenu);
```

```
menuBar.add(editMenu);
```

File Dialog Box



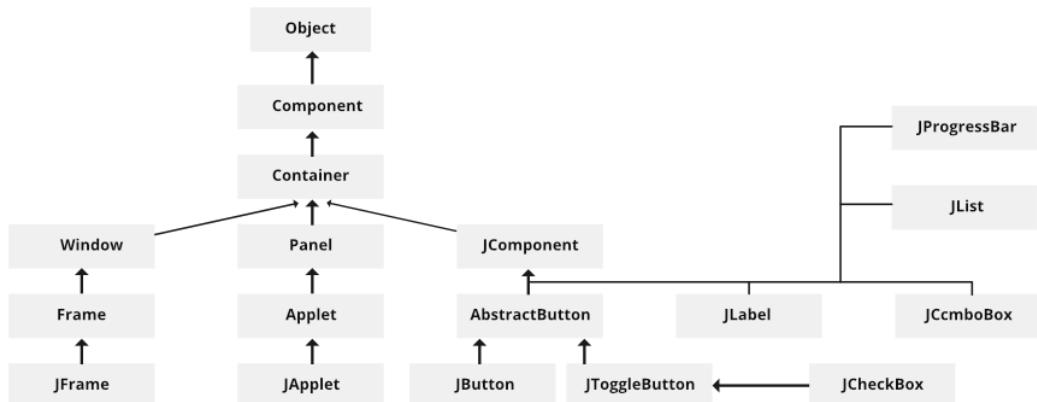
2) Swing

What is Swing ?

Swing is a GUI (Graphical User Interface) toolkit for Java that provides a set of lightweight, platform-independent components for building graphical user interfaces.

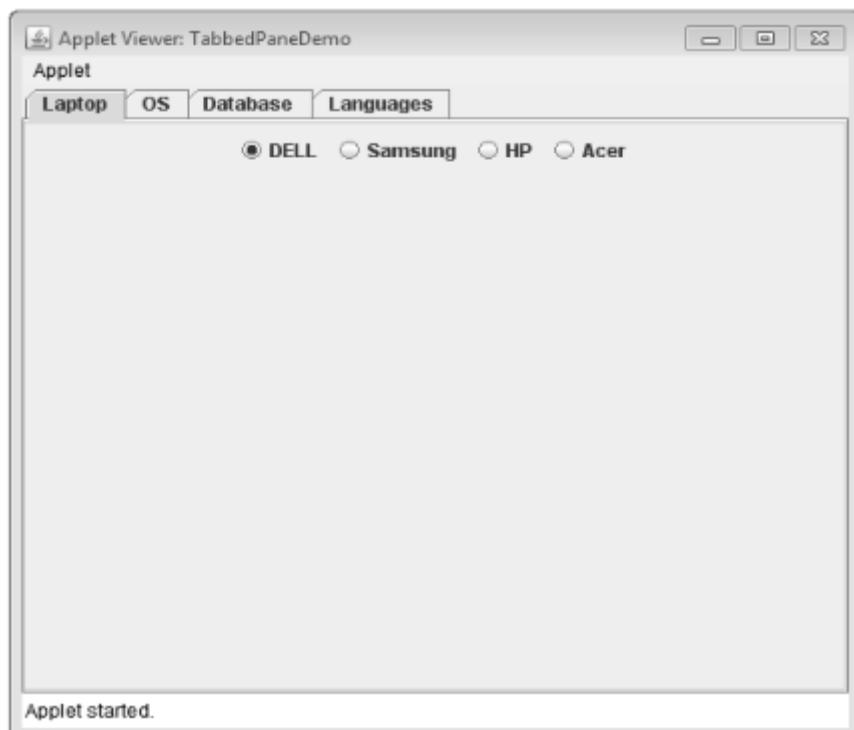
- It is part of the Java Foundation Classes (JFC) and is an extension of the original Abstract Window Toolkit (AWT).
- Unlike AWT, which relies on the native components of the underlying operating system, Swing components are entirely written in Java, making them consistent across different platforms.

Basic controls in Swing

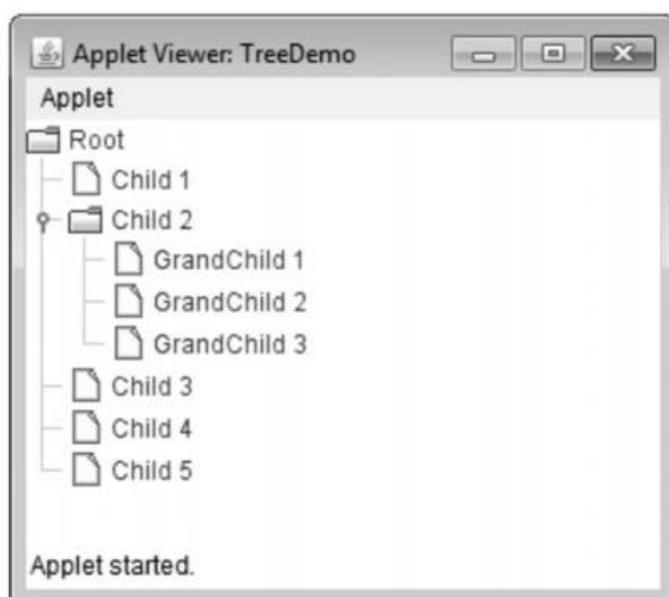


Swing Advanced Components

Tabbed Pane



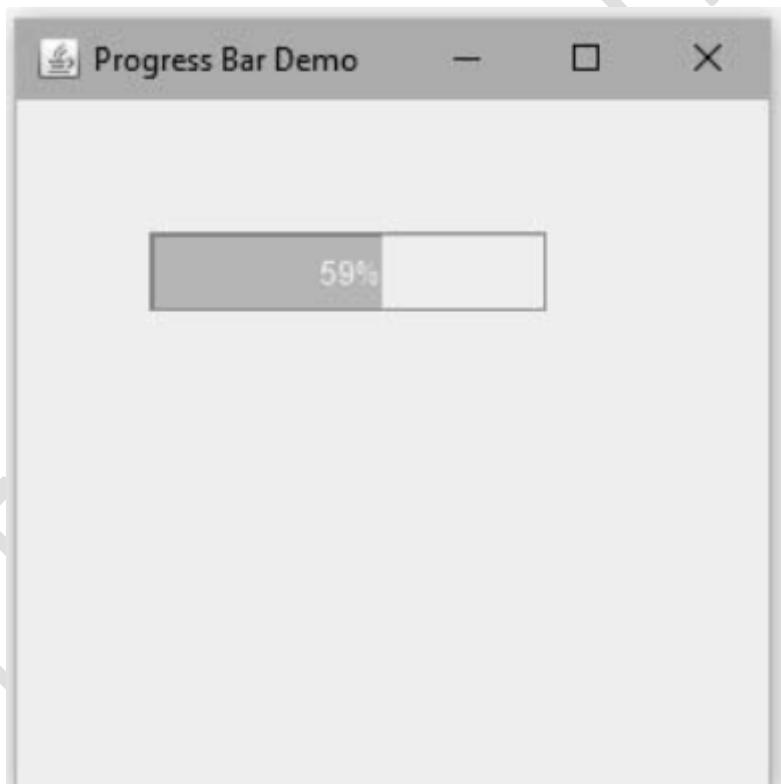
Trees



Table



Progress Bar



MVC Architecture

The Model-View-Controller (MVC) architecture is a design pattern commonly used in software development, including Swing applications in Java. MVC separates an application into three main components: Model, View, and Controller. This separation promotes modularity, code organization, and maintainability.

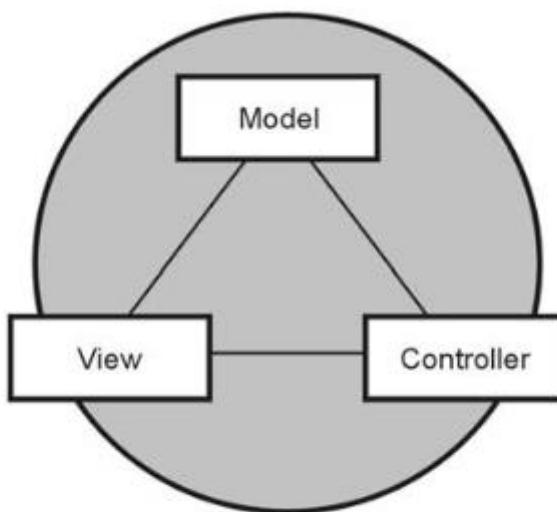


Fig. 2.5.1 MVC design pattern

Here's how MVC is typically applied in Swing:

1. Model:

- The Model represents the application's data and business logic. It is responsible for managing the state of the application and responding to requests for information or updates.
- In a Swing application, the Model is often implemented using plain Java classes or custom classes that encapsulate the application's data and behavior.

- The Model notifies registered observers (typically the View components) about changes in the data through the Observer pattern.

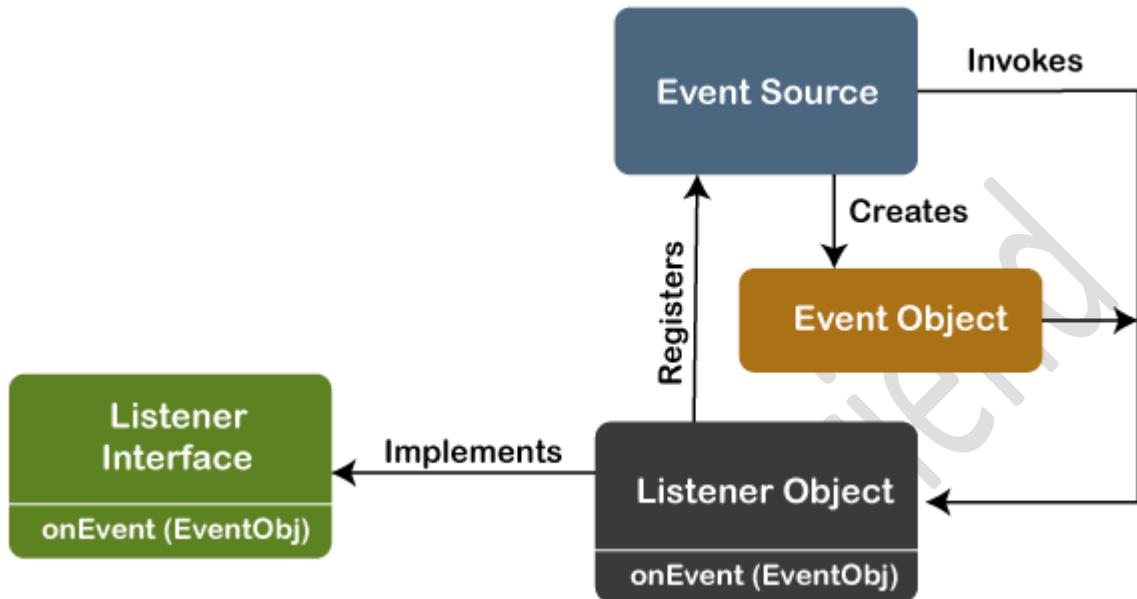
2. View:

- The View represents the presentation and display of the application's user interface. It is responsible for rendering the data to the user and capturing user input.
- In Swing, the View is typically implemented using Swing components such as JFrame, JPanel, JLabel, etc. These components are responsible for displaying the data provided by the Model.
- Views are observers of the Model. They update their display in response to changes in the Model's state.

3. Controller:

- The Controller acts as an intermediary between the Model and the View. It receives user input from the View, processes it, and updates the Model accordingly.
- In Swing, controllers are often implemented as event listeners. They respond to user actions (button clicks, menu selections, etc.) and invoke methods on the Model to update the data.
- The Controller also updates the View based on changes in the Model. It retrieves data from the Model and updates the display components in the View.

3) Event Handling



Event Delegation Model

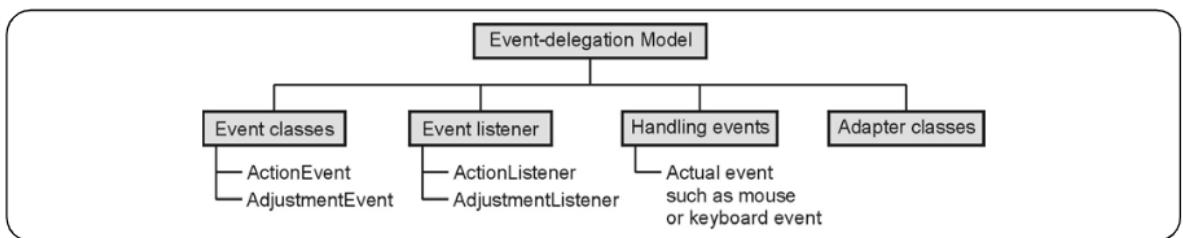


Fig. 3.1.1 Components of event delegation model

Event :-

An event is a notification or signal that something has occurred. It encapsulates information about the occurrence, such as the type of event and any relevant data.

- Examples of events include button clicks, key presses, mouse movements, and so on.

Event Source :-

An object, often a graphical user interface (GUI) component or a part of a software system, that has the capability to generate events.

- Examples of event sources in Java include buttons, text fields, checkboxes, and other user interface components. Non-UI examples could include timers, sensors, or other components that trigger events based on specific conditions.

Event Listener :-

An event listener is an object or a method that is designed to handle and respond to specific types of events. It is registered with an event source, and when the source generates an event, the corresponding event listener is notified.

- In Java, event listeners often implement specific listener interfaces provided by the Java API, such as ActionListener, MouseListener, or KeyListener.

Event Handling :-

Event handling is the process of designing code (event listeners) that responds to events generated by event sources.

- Event handlers are typically defined to perform specific actions or execute specific code when a particular event occurs.

Event Classes :-

In Java, the `java.awt.event` package provides a set of event classes and listener interfaces that are commonly used for handling events in GUI (Graphical User Interface) applications.

Here is a list of various event classes defined in the `java.awt.event` package:

1. ActionEvent:

- Represents an event generated by a user action, such as a button click or a menu item selection.
- Commonly associated with the ActionListener interface.

2. ActionListener:

- An interface for handling ActionEvents. Classes that implement this interface can respond to user actions, such as button clicks.

3. ItemEvent:

- Represents an event generated by an item selection or deselection, for example, in a Checkbox or Choice component.
- Associated with the ItemListener interface.

4. ItemListener:

- An interface for handling ItemEvents. Classes that implement this interface can respond to changes in the state of items, such as checkboxes or choice items.

5. KeyEvent:

- Represents an event generated by a key press or release.
- Commonly used with the KeyListener interface.

6. KeyListener:

- An interface for handling KeyEvent events. Classes that implement this interface can respond to keyboard events.

7. MouseEvent:

- Represents an event generated by a mouse action, such as a mouse click, movement, or drag.
- Commonly used with the MouseListener and MouseMotionListener interfaces.

8. MouseListener:

- An interface for handling mouse events, including clicks, releases, and enters/exits.
- Classes implementing this interface can respond to mouse-related events

9. MouseMotionEvent:

- Represents an event generated by a mouse movement.
- Used with the MouseMotionListener interface.

10. MouseMotionListener:

- An interface for handling mouse motion events. Classes that implement this interface can respond to mouse movement.

Event Classes With Methods (Must Learn)

3.2.5 TextEvent Class

Description	This event indicates that object's text is changed.
Constructor	<code>TextEvent(Object source, int id)</code> It constructs a TextEvent object.
Methods	<code>String paramString()</code> : Returns a parameter string identifying this text event.
Fields	<ul style="list-style-type: none">• <code>TEXT_FIRST</code> : The first number in the range of ids used for text events.• <code>TEXT_LAST</code> : The last number in the range of ids used for text events.• <code>TEXT_VALUE_CHANGED</code> : This event id indicates that object's text changed.

3.2.4 MouseEvent Class

Description	<p>This event occurs when mouse action occurs in a component. It reacts for both mouse event and mouse motion event.</p> <ul style="list-style-type: none"> • Mouse Events <ul style="list-style-type: none"> ○ A mouse button is pressed ○ A mouse button is released ○ A mouse button is clicked (pressed and released) ○ The mouse cursor enters a component ○ The mouse cursor exits a component • Mouse Motion Events <ul style="list-style-type: none"> ○ The mouse is moved ○ The mouse is dragged
Methods	<p>1) int getButton() : It returns which of the mouse button has changed the state.</p> <p>2) int getClickCount() : It returns number of mouse clicks.</p> <p>3) Point getLocationOnScreen() : Returns the absolute x, y position at that event.</p> <p>4) Point getPoint() : Returns the x,y position of the event relative to the source component.</p> <p>5) int getX() : Returns the horizontal x position of the event relative to the source component.</p> <p>6) int getY() : Returns the vertical y position of the event relative to the source component.</p>

3.2.3 KeyEvent Class

Description	<p>This event is generated when key on the keyboard is pressed or released.</p>
Constructors	<p>KeyEvent(Component source, int type, long t, int modifiers, int code)</p> <p>The <i>source</i> is a reference to the component that generates the event.</p> <p>The <i>type</i> specifies the type of event.</p> <p>The <i>t</i> denotes the system time at which the event occurs.</p> <p>The <i>modifiers</i> represent the modifier keys such as ALT, CNTRL, SHIFT that are pressed when an event occurs.</p>



Methods	<p>The <i>code</i> represents the virtual keycode such as VK_UP, VK_ESCAPE and so on.</p> <ul style="list-style-type: none"> • char getKeyChar() : It returns the character when a key is pressed.
Constants	<ul style="list-style-type: none"> • KEY_PRESSED : This event is generated when the key is pressed. • KEY_RELEASED : This event is generated when the key is released. • KEY_TYPED : This event is generated when the key is typed.

3.2.2 ItemEvent Class

Description	This event gets caused when an item is selected or deselected.
Constructors	ItemEvent(ItemSelectable, int, Object, int) Constructs a ItemSelectEvent object with the specified ItemSelectable source, type, item and item select state.
Methods	1) getItem() : It returns the item where the event occurred. 2) getItemSelectable() : Returns the ItemSelectable object where this event originated. 3) getStateChange() : Returns the state change type which generated the event.
Constants	1) ITEM_FIRST : Marks the first integer id for the range of item 2) ITEM_LAST : Marks the last integer id for the range of item 3) ITEM_STATE_CHANGED : The item state changed event type 4) SELECTED : The item selected state change type 5) DESELECTED : The item de-selected state change type

3.2.1 ActionEvent Class

Description	When an event gets generated due to pressing of button, or by selecting menu item or by selecting an item, this event occurs.
Constructors	ActionEvent(Object source, int id, string command, long when, int modifier) The <i>source</i> indicates the object due to which the event is generated. The <i>id</i> which is used to identify the type of event. The <i>command</i> is a string that specifies the command that is associated with the event. The <i>when</i> denotes the time of event. The <i>modifier</i> indicates the modifier keys such as ALT, CNTRL, SHIFT that are pressed when an event occurs.
Methods	<ul style="list-style-type: none">• String getActionCommand() : This method is useful for obtaining the <i>command</i> string which is specified during the generation of event.• int getModifiers() : This method returns the value which indicates the type of key being pressed at the time of event.• long getWhen() : It returns the time at which the event occurs.
Constants	There are four constants that are used to indicate the modifier keys being pressed. These constants are CTRL_MASK, SHIFT_MASK, META_MASK and ALT_MASK.

3.2.6 WindowEvent Class

Description	This is an event that indicates that a window has changed its status.
Constructor	<code>WindowEvent(Window source, int id)</code> : Constructs a WindowEvent object. <code>WindowEvent(Window source, int id, int oldState, int newState)</code> : Constructs a WindowEvent object with the specified previous and new window states.
Methods	<ol style="list-style-type: none">1) <code>int getNewState()</code> : It returns the new state of the window.2) <code>int getOldState()</code> : It returns the previous state of the window.3) <code>Window getOppositeWindow()</code> : Returns the other Window involved in this focus or activation change.4) <code>Window getWindow()</code> : Returns the originator of the event.
Fields	<ul style="list-style-type: none">• <code>WINDOW_ACTIVATED</code> : The window-activated event type.• <code>WINDOW_CLOSED</code> : The window closed event.• <code>WINDOW_DEACTIVATED</code> : The window-deactivated event type.• <code>WINDOW_FIRST</code> : The first number in the range of ids used for window events.• <code>WINDOW_LAST</code> : The last number in the range of ids used for window events.• <code>WINDOW_OPENED</code> : The window opened event.• <code>WINDOW_STATE_CHANGED</code> : The window-state-changed event type.

Adapter Class

In Java event handling, an adapter class is a convenience class that provides empty implementations (default or "no-op" implementations) of all methods defined in a particular listener interface. Adapter classes are useful when you want to create an object that responds to events but doesn't need to provide a concrete implementation for all the methods of the listener interface.

- The adapter classes are part of the `java.awt.event` package and typically have names ending with "Adapter." These classes are designed to be extended, and you can override only the methods you are interested in, instead of implementing all methods in the listener interface.

Here are some commonly used adapter classes in Java event handling:

1. ActionAdapter (for ActionListener):

- Provides empty implementations for the `actionPerformed` method of the ActionListener interface.

2. ItemAdapter (for ItemListener):

- Provides empty implementations for the `itemStateChanged` method of the ItemListener interface.

3. KeyAdapter (for KeyListener):

- Provides empty implementations for the `keyPressed`, `keyReleased`, and `keyTyped` methods of the KeyListener interface.

4. MouseAdapter (for MouseListener):

- Provides empty implementations for the `mouseClicked`, `mouseEntered`, `mouseExited`, `mousePressed`, and `mouseReleased` methods of the MouseListener interface.

5. MouseMotionAdapter (for MouseMotionListener):

- Provides empty implementations for the `mouseDragged` and `mouseMoved` methods of the MouseMotionListener interface.

6. MouseWheelAdapter (for MouseWheelListener):

- Provides an empty implementation for the `mouseWheelMoved` method of the MouseWheelListener interface.

7. FocusAdapter (for FocusListener):

- Provides empty implementations for the `focusGained` and `focusLost` methods of the FocusListener interface.

8. WindowAdapter (for WindowListener):

- Provides empty implementations for all methods of the WindowListener interface.

End of Day 1

UR ENGINEERING FRIEND

#1 stop Destination for Diploma & Degree

Day 2

4) Networking Basics

- Define socket & server socket
- Reserved sockets
- Proxy servers
- Internet addressing
- Inet Address
- TCP/ IP & UDP
- URL & URI Class
- Datagrams

5) Interacting with databases

- Define JDBC & ODBC
- JDBC architecture
- JDBC drivers
- Connectivity with database
- Driver interfaces

6) Servlets

- Define servlets
- Life cycle of servlet
- Servlet API
- Cookies

4) Networking Basics

Networking - In Java programming, "networking" refers to the ability of a Java program to communicate with other programs or devices over a network, such as the internet. Java provides a rich set of APIs (Application Programming Interfaces) for networking that allow developers to create networked applications.

Here are some key concepts and classes related to networking in Java:

1. `java.net` package:

- The `java.net` package provides classes for networking, including both low-level and high-level abstractions.
- Key classes include `Socket` and `ServerSocket` for creating client and server sockets, respectively.
- The `URL` class is used for working with URLs, and `URLConnection` can be used to establish a connection to a remote resource.

2. `Socket` and `ServerSocket` classes:

- `Socket` is a class that represents a client-side socket, and `ServerSocket` is used on the server side to listen for incoming connections.
- These classes facilitate communication between a client and a server using TCP/IP sockets.

3. `URL` and `URLConnection` classes:

- `URL` is a class that represents a Uniform Resource Locator, and it can be used to open a connection to a specified resource on the internet.
- `URLConnection` provides a higher-level interface for working with URLs, allowing you to read from and write to the resource.

4. `InetAddress` class:

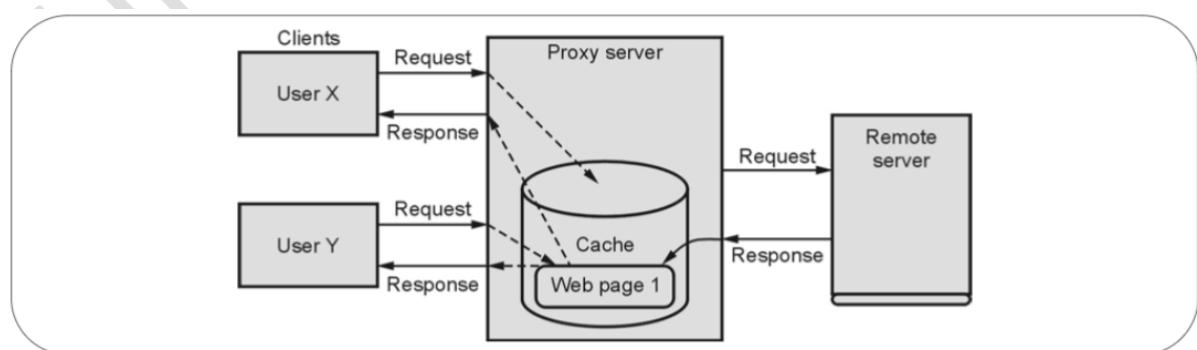
- The `InetAddress` class represents an IP address. It can be used to get the IP address of a host name or the host name of an IP address.

5. `DatagramSocket` and `DatagramPacket` classes:

- These classes are used for UDP (User Datagram Protocol) communication, which is a connectionless protocol. `DatagramSocket` represents a socket for sending and receiving datagram packets, and `DatagramPacket` represents the data packet.

Proxy Servers

A proxy server acts as an intermediary between a client (such as a web browser) and a destination server (such as a website). It serves several purposes, including enhancing security, privacy, and performance.



Inet Addressing

The `InetAddress` class in Java is part of the `java.net` package and is used for handling IP addresses and host names. It provides methods for performing operations related to network addressing, such as converting between host names and IP addresses. The class is particularly useful in network programming scenarios.

Here are some key aspects and methods of the `InetAddress` class:

1. Creating Instances:

- You can create an instance of `InetAddress` using static factory methods, such as `getLocalHost()` for the local machine or `getByName(String host)` to obtain an `InetAddress` instance for a specified host name or IP address.

```
InetAddress localHost = InetAddress.getLocalHost();
```

```
InetAddress googleAddress = InetAddress.getByName("www.google.com");
```

2. Getting Information:

- Once you have an `InetAddress` instance, you can retrieve information about the host, such as the host name and IP address.

```
System.out.println("Local Hostname: " + localHost.getHostName());
```

```
System.out.println("Local IP Address: " + localHost.getHostAddress());
```

3. Resolving Host Names:

- The `getByName` method can be used to obtain an `InetAddress` instance for a given host name. If the host name cannot be resolved, a `UnknownHostException` is thrown.

```
try {  
  
    InetAddress address = InetAddress.getByName("www.example.com");  
  
    System.out.println("IP Address: " + address.getHostAddress());  
  
} catch (UnknownHostException e) {  
  
    e.printStackTrace();  
  
}
```

4. Static Methods:

In addition to instance methods, the `InetAddress` class provides some static methods, such as `getAllByName(String host)` to retrieve all IP addresses associated with a given host name.

```
try {  
  
    InetAddress[] addresses =  
        InetAddress.getAllByName("www.example.com");  
  
    for (InetAddress address : addresses) {  
  
        System.out.println("IP Address: " + address.getHostAddress());  
  
    }  
  
} catch (UnknownHostException e) {  
  
    e.printStackTrace();
```

}

TCP , IP & UDP

TCP (Transmission Control Protocol), IP (Internet Protocol), and UDP (User Datagram Protocol) are fundamental protocols of the Internet protocol suite (TCP/IP suite). They are essential for network communication and play distinct roles in ensuring reliable and efficient data transmission.

1. TCP (Transmission Control Protocol):

TCP is a connection-oriented protocol that provides reliable, stream-oriented communication between two devices over a network.

Key Features:

- **Reliability:** TCP ensures reliable data delivery by establishing a connection, acknowledging received data, and retransmitting lost or corrupted packets.
- **Ordered Delivery:** It guarantees that data is delivered to the application layer in the order it was sent.
- **Flow Control:** TCP includes mechanisms to control the flow of data between sender and receiver, preventing congestion and ensuring efficient transmission.

2. IP (Internet Protocol):

IP is responsible for addressing and routing packets of data so they can travel across networks and arrive at the correct destination.

Key Features

- **Addressing:** IP assigns unique IP addresses to devices on a network, allowing them to be identified.
- **Routing:** IP is responsible for determining the best path for data packets to travel from the source to the destination across a network.
- **Connectionless:** IP is a connectionless protocol, meaning it doesn't establish a connection before sending data. Each packet is treated independently.

3. UDP (User Datagram Protocol):

UDP is a connectionless, lightweight protocol that provides low-latency communication, suitable for scenarios where some data loss is acceptable.

Key Features

- **Low Overhead:** UDP has less overhead compared to TCP because it lacks the connection setup, acknowledgment, and flow control mechanisms. This makes it faster but less reliable.
- **Unordered Delivery:** Unlike TCP, UDP does not guarantee the order of packet delivery, and it doesn't retransmit lost packets.
- **Broadcast and Multicast Support:** UDP supports broadcast and multicast communication, making it suitable for scenarios where one-to-many or many-to-many communication is needed.

URI & URL Class

In Java, the `URI` (Uniform Resource Identifier) and `URL` (Uniform Resource Locator) classes are part of the `java.net` package and are used for handling uniform resource identifiers and locators, respectively. Both classes are essential for working with web-related operations, such as forming, parsing, and manipulating URIs and URLs.

URI (Uniform Resource Identifier):

The `URI` class represents a uniform resource identifier, which is a string of characters that uniquely identifies a particular resource. A URI can be a URL or a URN (Uniform Resource Name).

Key Characteristics:

- **Immutability:** Once created, a `URI` object is immutable, meaning its value cannot be changed.

Example :-

```
import java.net.URI;  
import java.net.URISyntaxException;  
  
public class URIExample {  
    public static void main(String[] args) {  
        try {  
            // Creating a URI  
            URI uri = new URI("https://www.example.com/path?query=param");  
        } catch (URISyntaxException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
// Accessing URI components  
System.out.println("Scheme: " + uri.getScheme());  
System.out.println("Host: " + uri.getHost());  
System.out.println("Path: " + uri.getPath());  
System.out.println("Query: " + uri.getQuery());  
} catch (URISyntaxException e) {  
    e.printStackTrace();  
}  
}  
}
```

URL (Uniform Resource Locator):

The `URL` class represents a uniform resource locator, which is a specific type of URI that provides the means to locate and retrieve a resource on the internet.

Key Characteristics:

- **Mutable:** Unlike `URI`, a `URL` object is mutable, meaning its components can be modified after creation.

Example

```
import java.net.MalformedURLException;  
import java.net.URL;  
  
public class URLExample {
```

```
public static void main(String[] args) {  
    try {  
        // Creating a URL  
        URL url = new  
        URL("https://www.example.com/path?query=param");  
  
        // Accessing URL components  
        System.out.println("Protocol: " + url.getProtocol());  
        System.out.println("Host: " + url.getHost());  
        System.out.println("Path: " + url.getPath());  
        System.out.println("Query: " + url.getQuery());  
    } catch (MalformedURLException e) {  
        e.printStackTrace();  
    }  
}
```

Datagrams

In Java, "datagrams" typically refer to the concept of datagram communication in networking, specifically associated with the User Datagram Protocol (UDP). In networking, a datagram is an independent, self-contained message sent over the network without establishing a persistent connection between the sender and receiver.

1. `DatagramPacket` Class:

- The `DatagramPacket` class in Java is part of the `java.net` package and represents a datagram packet.
- It is used to send or receive datagrams through a `DatagramSocket`.
- A `DatagramPacket` contains the data to be sent or received, information about the sender or receiver, and the length of the data.

```
// Example of creating a DatagramPacket for sending data
byte[] sendData = "Hello, Server!".getBytes();
InetAddress serverAddress = InetAddress.getByName("localhost");
int serverPort = 12345;
DatagramPacket    sendPacket    =    new    DatagramPacket(sendData,
sendData.length, serverAddress, serverPort);
```

2. `DatagramSocket` Class:

- The `DatagramSocket` class is used for creating a UDP socket for sending and receiving datagrams.
- It can be used for both client and server applications.
- Unlike TCP, UDP is connectionless, and each datagram is independent of others.

```
// Example of creating a DatagramSocket for sending and receiving datagrams
DatagramSocket datagramSocket = new DatagramSocket();
```

3. Sending Datagram:

- To send a datagram, you use the `send` method of the `DatagramSocket` class, passing in a `DatagramPacket` containing the data and destination information.

```
// Sending a datagram  
datagramSocket.send(sendPacket);
```

4. Receiving Datagram:

- To receive a datagram, you use the `receive` method of the `DatagramSocket` class, which fills a `DatagramPacket` with the received data.

```
// Receiving a datagram  
byte[] receiveData = new byte[1024];  
DatagramPacket receivePacket = new DatagramPacket(receiveData,  
receiveData.length);  
datagramSocket.receive(receivePacket);
```

5) Interacting with Databases

JDBC & ODBC

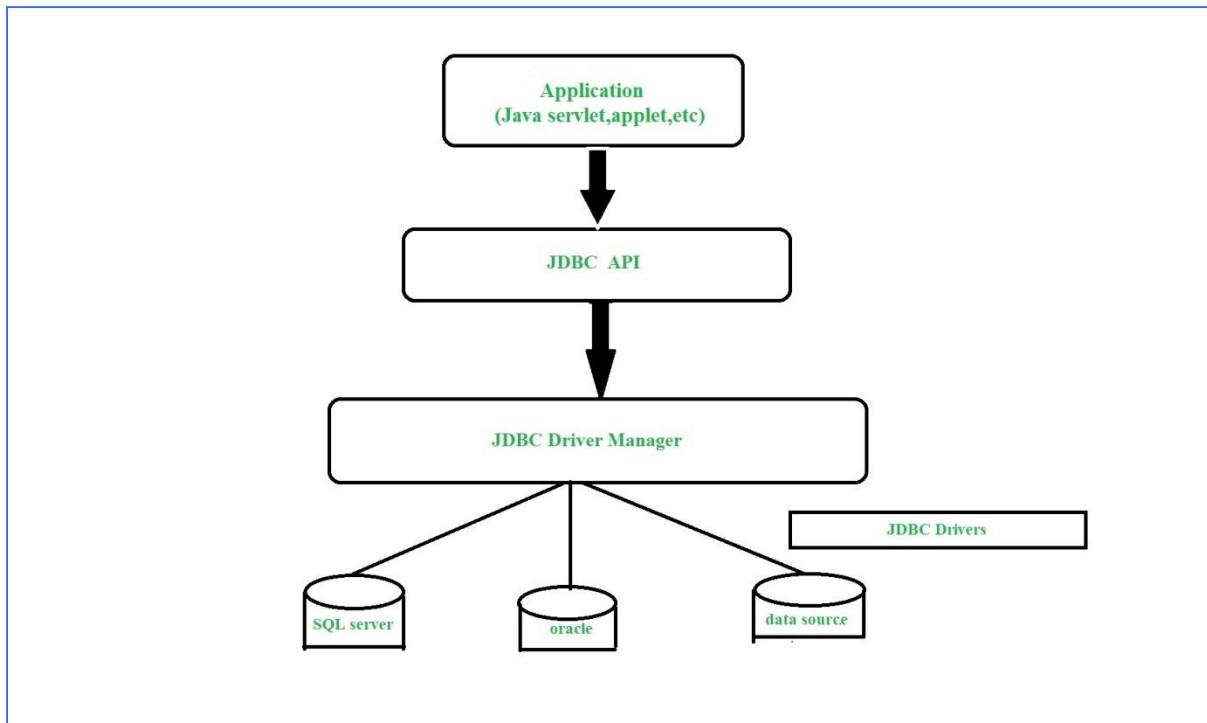
JDBC (Java Database Connectivity):

JDBC, which stands for Java Database Connectivity, is a Java API that enables Java applications to interact with databases. It provides a standard interface for connecting to relational databases and executing SQL queries. JDBC allows Java programs to perform database operations such as querying, updating, inserting, and deleting data.

ODBC (Open Database Connectivity):

ODBC, or Open Database Connectivity, is a standard interface for connecting and interacting with databases. It is not specific to Java but is a generic interface used by various programming languages. ODBC allows applications to communicate with different database management systems (DBMS) using a common set of functions.

JDBC Architecture



The JDBC architecture consists of two-tier and three-tier processing models to access a database. They are as described below:

1. **Two-tier model:** A java application communicates directly to the data source. The JDBC driver enables the communication between the application and the data source. When a user sends a query to the data source, the answers for those queries are sent back to the user in the form of results. The data source can be located on a different machine on a network to which a user is connected. This is known as a **client/server configuration**, where the user's machine acts as a client, and the machine has the data source running acts as the server.
2. **Three-tier model:** In this, the user's queries are sent to middle-tier services, from which the commands are again sent to the data source. The results are sent back to the middle tier, and from there to the user.

This type of model is found very useful by management information system directors.

JDBC Drivers

JDBC (Java Database Connectivity) drivers are platform-specific implementations that provide a standardized interface for Java applications to connect to different types of databases. There are four types of JDBC drivers, each with its own characteristics and use cases:

1. Type 1: JDBC-ODBC Bridge Driver (Bridge Driver):

The Type 1 driver, also known as the JDBC-ODBC Bridge driver, relies on an ODBC (Open Database Connectivity) driver to connect to the database.

Pros:

- Provides a bridge between the JDBC API and ODBC, allowing Java applications to interact with any ODBC-compliant database.
- Platform-independent at the Java level.

Cons:

- Requires the ODBC driver to be installed on the client machine.
- Performance may not be as efficient as other types due to the additional layer.

2. Type 2: Native-API Driver (Partially Java Driver):

The Type 2 driver uses a database-specific native library to connect to the database. It communicates directly with the database using the vendor's client-side API.

Pros:

- Provides better performance than the Type 1 driver because it communicates directly with the database using native code.
- Platform-specific native code optimized for the target database.

Cons:

- Requires the database-specific native library to be installed on the client machine.
- Not entirely platform-independent, as it relies on native code.
-

3. Type 3: Network Protocol Driver (Middleware Driver):

The Type 3 driver, also known as the Network Protocol driver or Middleware driver, communicates with a middleware server, which in turn connects to the database. The middleware server converts JDBC calls into a database-specific protocol.

Pros:

- Database connectivity is achieved through a standardized middleware server, making it database-independent at the client side.
- Platform-independent at the Java level.

Cons:

- Requires the installation and configuration of middleware on both the client and server sides.
- Performance may be slower than Type 2 due to the additional layer.

4. Type 4: Thin or Direct-to-database Driver (Pure Java Driver):

The Type 4 driver, also known as the Thin driver or Direct-to-database driver, is a pure Java implementation that communicates directly with the database using the database-specific protocol.

Pros:

- Platform-independent as it is implemented entirely in Java.
- No need for a separate installation or configuration on the client side.

Cons:

- Requires a separate driver for each database, as it communicates directly with the database's protocol.
- Performance is generally good, but it may vary depending on the specific database.

JDBC Interfaces

The JDBC (Java Database Connectivity) API provides a set of interfaces and classes for Java applications to interact with relational databases. These interfaces define a standard for database connectivity, allowing developers to write database-independent code. Here are some key JDBC interfaces:

1. Driver:

The `Driver` interface provides a standard mechanism for managing database drivers. Each JDBC driver must implement this interface to allow the `DriverManager` to identify and select the appropriate driver.

- It has methods like `connect` to establish a connection to the database and `acceptsURL` to determine if the driver can connect to a given URL.

```
public interface Driver {  
  
    Connection connect(String url, Properties info) throws SQLException;  
  
    boolean acceptsURL(String url) throws SQLException;  
  
    // ... other methods  
  
}
```

2. Connection:

The `Connection` interface represents a connection to a database. It provides methods for creating statements, managing transactions, and controlling various aspects of the connection.

- Key methods include `createStatement`, `prepareStatement`, `commit`, `rollback`, and `close`.

```
public interface Connection extends Wrapper, AutoCloseable {  
  
    Statement createStatement() throws SQLException;  
  
    PreparedStatement prepareStatement(String sql) throws SQLException;  
  
    void commit() throws SQLException;  
  
    void rollback() throws SQLException;  
  
    // ... other methods  
  
}
```

3. Statement:

The `Statement` interface represents a SQL statement to be executed against a database. It provides methods for executing queries, updates, and stored procedures.

- It has three sub-interfaces: `Statement` (for general SQL statements), `PreparedStatement` (precompiled SQL statements), and `CallableStatement` (for executing stored procedures).

```
public interface Statement extends Wrapper, AutoCloseable {  
  
    ResultSet executeQuery(String sql) throws SQLException;  
  
    int executeUpdate(String sql) throws SQLException;  
  
    // ... other methods  
  
}
```

4. ResultSet:

The `ResultSet` interface represents the result set of a query. It provides methods for navigating through the rows of the result set and retrieving data.

- It has methods like `next` to move to the next row, `getString` to retrieve string values, and various other methods for different data types.

```
public interface ResultSet extends Wrapper, AutoCloseable {  
  
    boolean next() throws SQLException;  
  
    String getString(int columnIndex) throws SQLException;
```

```
// ... other methods  
}
```

5. PreparedStatement:

The `PreparedStatement` interface extends `Statement` and represents a precompiled SQL statement. It allows for parameterized queries and improves performance by reusing execution plans.

- It provides methods for setting parameters using placeholders and executing the statement.

```
public interface PreparedStatement extends Statement {  
  
    void setString(int parameterIndex, String x) throws SQLException;  
  
    // ... other methods  
}
```

6. CallableStatement:

The `CallableStatement` interface extends `PreparedStatement` and is used for executing stored procedures. It provides methods for registering output parameters and handling result sets.

```
public interface CallableStatement extends PreparedStatement {  
  
    void registerOutParameter(int parameterIndex, int sqlType) throws  
SQLException;
```

```
// ... other methods  
}
```

Ur Engineering Friend

6) Servlets

Servlets are Java-based components that are used to create dynamic web applications. They are part of the Java Platform, Enterprise Edition (Java EE) and provide a way to extend the capabilities of web servers to handle requests, process data, and generate dynamic responses for clients (typically web browsers).

Servlets operate on the server side of a web application and are responsible for handling various tasks such as processing user input, interacting with databases, and generating dynamic content. They are often employed in conjunction with web servers like Apache Tomcat.

Key characteristics and features of servlets include:

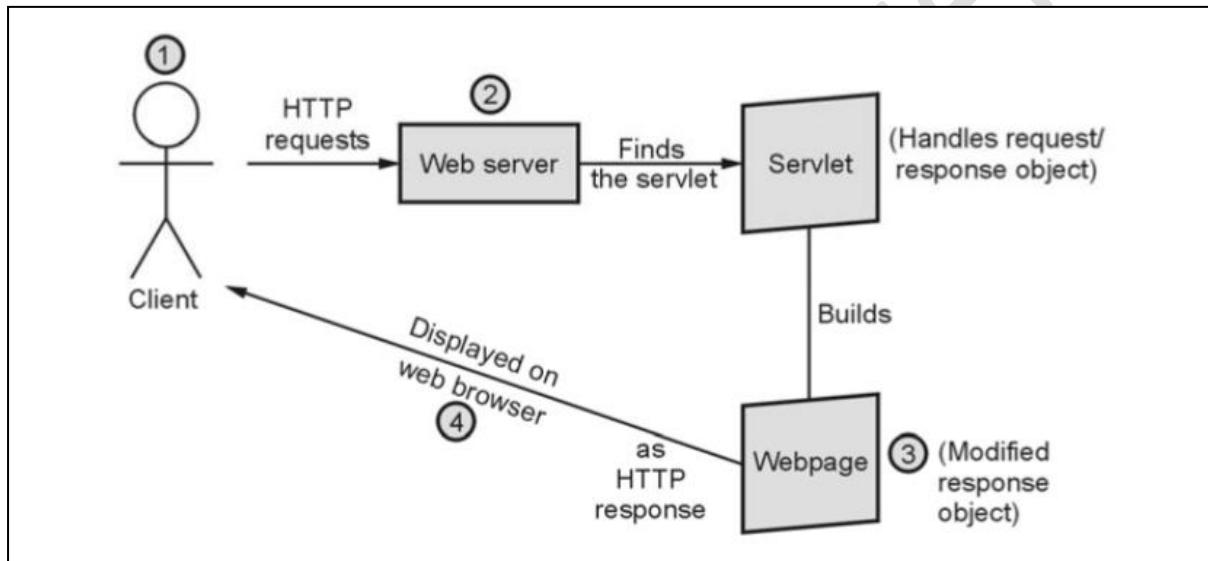
1. Platform Independence: Servlets are written in Java, making them platform-independent. Once compiled, a servlet can run on any system that supports the Java Virtual Machine (JVM).

2. Server-Side Processing: Servlets handle requests and generate responses on the server side, allowing for dynamic content generation and interaction with databases.

3. Lifecycle Management: Servlets follow a lifecycle that includes initialization, service request handling, and destruction. Developers can override specific methods to control the behavior of the servlet at different stages of its lifecycle.

4. HTTP Support: Servlets are designed to work with the HTTP protocol, making them suitable for building web applications. They can respond to HTTP requests such as GET and POST and process parameters sent by clients.

5. Extensibility: Servlets can be extended to support various functionalities by implementing specific interfaces or by extending existing classes. This extensibility allows developers to tailor servlet behavior to meet the requirements of their applications.



Lifecycle of Servlet

The entire life cycle of a Servlet is managed by the **Servlet container** which uses the **javax.servlet.Servlet** interface to understand the Servlet object and manage it. So, before creating a Servlet object, let's first understand the life cycle of the Servlet object which is actually understanding how the Servlet container manages the Servlet object.

Stages of the Servlet Life Cycle:

The Servlet life cycle mainly goes through four stages,

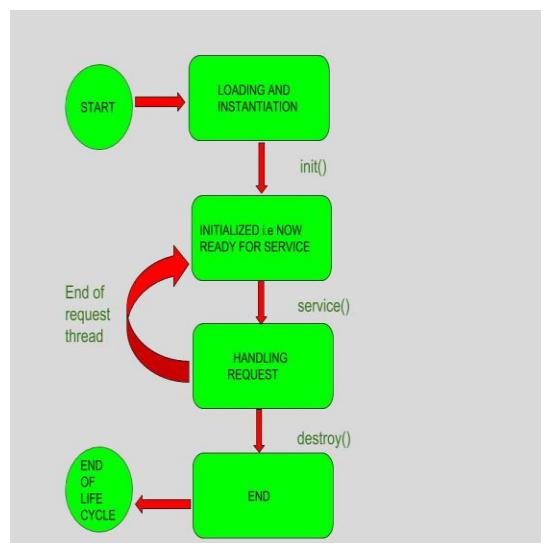
- **Loading a Servlet.**
- **Initializing the Servlet.**
- **Request handling.**
- **Destroying the Servlet.**

1. **Loading a Servlet:** The first stage of the Servlet lifecycle involves loading and initializing the Servlet by the Servlet container. The Web container or Servlet Container can load the Servlet at either of the following two stages :

- Initializing the context, on configuring the Servlet with a zero or positive integer value.
- If the **Servlet** is not preceding stage, it may delay the loading process until the Web container determines that this Servlet is needed to service a request.

The Servlet container performs two operations in this stage :

- **Loading :** Loads the Servlet class.
- **Instantiation :** Creates an instance of the Servlet. To create a new instance of the Servlet, the container uses the no-argument constructor.



2. **Initializing a Servlet:** After the Servlet is instantiated successfully, the Servlet container initializes the instantiated Servlet object. The container initializes the Servlet object by invoking the **Servlet.init(ServletConfig)** method which accepts ServletConfig object reference as parameter.

3. **Handling request:** After initialization, the Servlet instance is ready to serve the client requests. The Servlet container performs the following operations when the Servlet instance is located to service a request :

4. **Destroying a Servlet:** When a Servlet container decides to destroy the Servlet, it performs the following operations,

- It allows all the threads currently running in the service method of the Servlet instance to complete their jobs and get released.
- After currently running threads have completed their jobs, the Servlet container calls the **destroy()** method on the Servlet instance.

After the **destroy()** method is executed, the Servlet container releases all the references of this Servlet instance so that it becomes eligible for garbage collection.

Servlet Life Cycle Methods

There are three life cycle methods of a Servlet :

- **init()**
- **service()**
- **destroy()**

Servlet API

- **javax.servlet**
- **javax.servlet.http**

Servlet Interfaces

The Servlet API in Java provides a set of interfaces and classes for building web applications.

Here are some key interfaces in the Servlet API:

1. Servlet Interface:

- The `Servlet` interface defines the methods that all servlets must implement.
- Methods include `init()`, `service()`, `destroy()`, and `getServletConfig()`.

Method	Description
void init(ServletConfig s)	This method is called for initialising the servlet. The initialisation parameter is obtained from the ServletConfig interface.
void destroy()	This method is called when the servlet has to be unloaded.
ServletConfig getServletConfig()	This method is used to obtain the initialisation parameters.
void service(ServletRequest req, ServletResponse res)	This method is used to implement the service that a servlet should provide. The clients request is processed and a response is given.
String getServletInfo()	This method is used to obtain description of the servlet.

2. `ServletConfig` Interface:

- The `ServletConfig` interface provides a servlet with its configuration information.
- This information is typically specified in the web deployment descriptor (web.xml).

Method	Description
String getServletName()	This method is used to obtain the name of the servlet.
String getInitParameter(String p)	This method returns the value of the parameter p
Enumeration getInitParameterNames()	This method returns the names of initialisation parameters.
ServletContext getServletContext()	This method returns the context for the servlet.

3. `ServletContext` Interface:

- The `ServletContext` interface represents the web application and provides a way for servlets to interact with the container.
- It is obtained from the `ServletConfig` or directly from the `ServletRequest`.
- Methods include `getRealPath(String path)`, `getResource(String path)`, and `getRequestDispatcher(String path)`.

Method	Description
Object getAttribute(String attribute_name)	The value of the attribute attribute_name in the current session is returned.
void setAttribute(String attribute_name, object value)	The attribute_name is passed to the object value.
String getServerInfo()	This method returns the information about the server.
String log(String str)	Writes the str to the servlet log.
String getMimeType(String file)	It returns the MIME type of the file.

4. `ServletRequest` Interface:

- The `ServletRequest` interface represents a client request to the servlet.
- It provides methods for accessing parameters, attributes, input streams, and other request-related information.

Method	Description
Object getAttribute(String attribute_name)	The value of the attribute attribute_name in the current session is returned.
int getContentlength()	It returns the content size of the request.
String getContentType()	It returns the type of the request.
getInputStream()	This method is used to read the binary data from the request.
getProtocol()	Returns the name of the protocol used.

5. ServletResponse Interface:

- The `ServletResponse` interface represents the response that a servlet sends to a client.
- It provides methods for setting response headers, obtaining output streams, and other response-related operations.
- Methods include `setContentType(String type)`, `getWriter()`, and `setStatus(int sc)`.

Method	Description
String getCharacterEncoding()	This method returns the character encoding.
ServletOutputStream getOutputStream()	This method returns outputstream which is used to write the data for responding.
PrintWriter getWriter()	This method is used to write the character data to the response.
void setContentLength(int size)	This method sets the length of the content equal to the size.
void setContentType(String Type)	This methods sets the type of the content.

Cookies

Cookies are small pieces of data stored on the client's machine by a web browser. They are often used to store information about the user or the state of the application. In Java, when working with servlets, you can use the `Cookie` class to create and manage cookies. The `Cookie` class is part of the `javax.servlet.http` package.

Constructors:

1. `Cookie(String name, String value)`:

- This constructor creates a new cookie with the specified name and value.

Example:

```
Cookie myCookie = new Cookie("username", "john_doe");
```

2. `Cookie(String name, String value, String path)`:

- This constructor creates a new cookie with the specified name, value, and path.
- The path indicates the scope of the cookie within the web application.

Example:

```
Cookie myCookie = new Cookie("username", "john_doe", "/myapp");
```

Sr. No.	Method	Purpose
1.	public string getName()	It returns the name of the cookie.
2.	public String getValue()	It returns the value of the cookie.
3.	public string setName()	It sets or changes the name of the cookie.
4.	public String setValue()	It sets or changes the value of the cookie.
5.	public void addCookie(Cookie c)	The cookie is added in the response object of HttpServletResponse interface.
6.	public Cookie[] getCookies()	All the cookies can be returned using this method with the help of HttpServletRequest interface.

End of Day 2

UR ENGINEERING FRIEND

#1 stop Destination for Diploma & Degree