MODULE-3

1. Concept of Interface

An **interface** in object-oriented programming is a reference type, similar to a class, that can contain only abstract methods (until Java 8) and constant (final) variables. It provides a way to achieve **abstraction** and **multiple inheritance** in languages like Java and C#.

An interface specifies *what* a class must do, not *how* it does it. A class that implements an interface must provide implementations for all its methods. This allows different classes to implement the same interface in different ways, supporting polymorphism.

For example:

```
interface Animal {
    void makeSound(); // abstract method
}
class Dog implements Animal {
    public void makeSound() {
        System.out.println("Bark");
    }
}
```

Key Features:

- No constructors (cannot be instantiated).
- Methods are public and abstract by default (until Java 8).
- Fields are public, static, and final by default.
- A class can implement multiple interfaces.

Interfaces help in achieving loose coupling and are widely used in software design patterns.

---

2. Private Interface Methods

Private interface methods were introduced in **Java 9**. These methods allow an interface to **reuse code** within **default and static methods**, but **cannot be accessed outside** the interface.

They help in improving code **modularity** and **readability** by avoiding code duplication. These methods are like helper methods inside the interface, ensuring that only the interface itself can call them.

Syntax Example:

```
interface MyInterface {
    private void log(String message) {
        System.out.println("Log: " + message);
    }

    default void doSomething() {
        log("Doing something...");
    }
}
```

Key Points:

- Cannot be abstract or static outside the interface.
- Cannot be inherited or overridden.
- Used internally by default or static methods to share common logic.

This feature makes interfaces more powerful and closer to traits in other languages.

---

3. Default Interface Methods

Default methods in interfaces were introduced in **Java 8**. They allow developers to add **method implementations** to interfaces **without breaking** the existing classes that implement the interface.

This supports **backward compatibility** and allows interfaces to evolve over time.

Syntax Example:

```
interface Vehicle {
    default void start() {
        System.out.println("Vehicle is starting");
    }
}
```

Key Features:

- Uses the `default` keyword.
- Implementing classes may override these methods.
- Helps avoid the need for utility/helper classes.

Use Case:
Suppose a new method is added to an interface. Without default methods, all implementing classes would have to implement it. With default methods, they can inherit the default behavior without any changes.

---

4. Concept of Static Methods in Interfaces

Static methods in interfaces were introduced in **Java 8**. These methods belong to the **interface itself**, not to the instance or the implementing class. They are used to define **utility or helper functions** related to the interface.

Syntax Example:

```
interface MathUtil {
    static int add(int a, int b) {
        return a + b;
    }
}
```

Usage:

```
int result = MathUtil.add(5, 3); // Called using interface name
```

Key Points:

- Cannot be overridden by implementing classes.
- Must be accessed using the interface name.
- Help in organizing utility methods logically within the interface context.

This feature encourages better code organization by associating relevant helper methods directly with the interface they serve.

MODULE-4

1. Concepts of Exception Handling

Definition:

Exception Handling in Java is a powerful mechanism that handles runtime errors, allowing the normal flow of the application to continue even when unexpected issues occur.

Key Terms:

- **Exception:** An object representing an error condition that disrupts program flow.
- **Throwable:** Superclass of all errors and exceptions in Java.

Types of Exceptions:

- **Checked Exceptions:** Checked at compile-time (e.g., IOException).
- **Unchecked Exceptions:** Occur at runtime (e.g., ArithmeticException).

How it Works:

Java uses **five keywords** to handle exceptions:

- `try`: Wraps the code that might throw an exception.
- `catch`: Catches and handles the exception.

- `throw`: Used to explicitly throw an exception.
- `throws`: Declares the exceptions a method may throw.
- `finally`: Executes code regardless of exception outcome.

Example:

```
try {
    int a = 5 / 0;
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero");
} finally {
    System.out.println("Cleanup code here.");
}
```

Purpose:

- Prevent program crashes.
- Provide meaningful messages to users.
- Ensure the application behaves gracefully.

2. Benefits of Exception Handling

Introduction:

Exception handling in Java is not just a mechanism to catch errors but also provides multiple advantages that help build robust, error-resilient applications.

Key Benefits:

1. **Maintains Program Flow:**
    - Java's exception handling mechanism allows the program to continue executing even after an error occurs.
    - Without it, the program would terminate abruptly.

2. **Separates Error Handling Code:**
    - Keeps normal logic separate from error-handling logic, improving readability and maintenance.

Example:

```
try {
    int a = 100 / 0;
} catch (ArithmeticException e) {
    System.out.println("Error: " + e);
}
```

    -
3. **Group and Categorize Errors:**
    - Java groups exceptions using a well-defined hierarchy. This makes it easy to catch broad or specific categories of errors.
4. **Encourages Fault Tolerance:**
    - Applications can recover gracefully from unexpected events, improving user experience.
5. **Supports Exception Propagation:**
    - With the `throws` keyword, methods can delegate exception handling to the calling method.
6. **Allows Custom Exceptions:**
    - Developers can create their own exceptions for domain-specific logic.

Example:

```
class AgeException extends Exception {
    AgeException(String message) { super(message); }
```

    - `}`

3. Exception Hierarchy

Overview:

Java's exception handling is built on a **hierarchical structure** beginning with the `Throwable` class, which has two main branches: **Error** and **Exception**.

Hierarchy Diagram:

```
Object
 └── Throwable
      ├── Error (not handled usually)
      │    ├── OutOfMemoryError
      │    └── StackOverflowError
      └── Exception (handled)
           ├── CheckedException (e.g., IOException)
           └── UncheckedException (e.g., NullPointerException)
```

Categories:

1. **Error:**
    - Indicates serious problems not intended to be caught (e.g., hardware failure).
    - Examples: `OutOfMemoryError`, `VirtualMachineError`.
2. **Exception:**
    - Represents conditions that an application might want to catch.
    - Subdivided into:
        - **Checked Exceptions:** Must be declared or handled.
            - Example: `SQLException`, `FileNotFoundException`.
        - **Unchecked Exceptions:** Inherit from `RuntimeException`. Not enforced by compiler.

        - Example: `ArithmeticException`, `NullPointerException`.

Purpose:

- Promotes organized exception handling.
- Enables catching general or specific exceptions.

4. Life Cycle of a Thread

Introduction:

A thread in Java undergoes several **states** during its lifetime. Java provides built-in thread support via the `Thread` class and `Runnable` interface.

Life Cycle Phases:

1. **New:**
    - Thread is created using `Thread t = new Thread();`.
2. **Runnable:**
    - Thread is ready to run, waiting for CPU.
    - Reached after calling `start()`.
3. **Running:**
    - Thread gets CPU and starts executing `run()` method.
4. **Blocked/Waiting:**
    - Thread is inactive, waiting for a resource or signal.
5. **Timed Waiting:**
    - Thread waits for a specified time (e.g., `sleep(1000)`).
6. **Terminated (Dead):**

- Thread completes execution or is forcefully stopped.

Diagram:

```
New → Runnable → Running
   ↘ Waiting/Timed Waiting ↘
                  ↘ Terminated
```

Example:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Running thread");
    }
}
```

5. Checked and Unchecked Exceptions

Introduction:

In Java, exceptions are categorized into two main types: **Checked** and **Unchecked**. This classification is based on whether the compiler checks the exception handling.

---

1. Checked Exceptions

- These are exceptions that are **checked at compile-time**.
- The compiler ensures that the programmer handles these exceptions using a `try-catch` block or declares them using the `throws` keyword.
- These are usually **external errors** that can be predicted and recovered from.

Common Checked Exceptions:

- `IOException`

- `SQLException`
- `FileNotFoundException`

Example:

```
import java.io.*;

class Example {
    public static void main(String[] args) {
        try {
            FileReader fr = new FileReader("file.txt"); // Checked
Exception
        } catch (FileNotFoundException e) {
            System.out.println("File not found!");
        }
    }
}
```

---

2. Unchecked Exceptions

- These are **not checked at compile-time**, but occur during **runtime**.
- These exceptions are **subclasses of `RuntimeException`**.
- They usually indicate **programming errors**, like logic issues or improper use of APIs.

Common Unchecked Exceptions:

- `NullPointerException`
- `ArithmeticException`
- `ArrayIndexOutOfBoundsException`

Example:

```
class Example {
    public static void main(String[] args) {
        int a = 10 / 0; // ArithmeticException
        System.out.println("This won't be printed");
    }
}
```

Differences Between Checked and Unchecked Exceptions:

Feature	Checked Exception	Unchecked Exception
Compile-time checking	Yes	No
Must handle or declare	Yes	No
Parent class	`Exception`	`RuntimeException`
Example	`IOException`	`NullPointerException`

Conclusion:

Proper distinction and handling of checked and unchecked exceptions ensures robust and fault-tolerant programs. Developers are advised to handle **checked exceptions** diligently and fix the root cause of **unchecked exceptions**.

6. Usage of try, catch, throw, throws, and finally (with Program)

Introduction:

Java provides a robust mechanism to handle exceptions using five main keywords:

- `try`
- `catch`

- `throw`
- `throws`
- `finally`

These keywords work together to catch, handle, and manage exceptions effectively, ensuring smooth program execution.

Explanation of Keywords:

1. **`try` block**
    - Contains code that might throw an exception.
    - Only one `try` block is allowed per exception handling unit.
2. **`catch` block**
    - Used to handle the exception thrown in the try block.
    - Multiple catch blocks can be used to handle different exception types.
3. **`throw` keyword**
    - Used to **explicitly throw** an exception (either built-in or custom).
    - Syntax: `throw new ExceptionType("message");`
4. **`throws` keyword**
    - Declares exceptions a method may throw.
    - Used in method signature.
5. **`finally` block**
    - Contains code that **always executes**, regardless of exception occurrence.

- Typically used for cleanup operations (e.g., closing files).

Example Program:

```
class Example {
    static void validateAge(int age) throws ArithmeticException {
        if (age < 18) {
            throw new ArithmeticException("Not eligible to vote");
        } else {
            System.out.println("Eligible to vote");
        }
    }

    public static void main(String[] args) {
        try {
            validateAge(16); // This will throw exception
        } catch (ArithmeticException e) {
            System.out.println("Caught Exception: " + e.getMessage());
        } finally {
            System.out.println("Validation completed.");
        }
    }
}
```

Output:

```
Caught Exception: Not eligible to vote
Validation completed.
```

Key Points:

- `throw` is used to **generate** an exception.
- `throws` is used to **declare** an exception.

- `finally` is **always executed**, even if an exception is not thrown or caught.
- `try-catch` ensures that the program does not terminate unexpectedly.

Conclusion:

Using these five keywords effectively enables structured and readable exception handling, promoting safe and error-free execution of Java programs.

7. Inter-Thread CommunicationIntroduction:

Inter-thread communication allows threads to **cooperate** by **sharing information** about their state, rather than polling or busy-waiting. Java provides built-in support via the `Object` methods: `wait()`, `notify()`, and `notifyAll()`.

Key Methods:

1. **`wait()`**
    - Called by a thread **owning** the object's monitor.
    - Causes the thread to **release the lock** and enter the **waiting** state until another thread calls `notify()` or `notifyAll()`.
2. **`notify()`**
    - Wakes up **one** randomly chosen thread that is waiting on the same object's monitor.
3. **`notifyAll()`**
    - Wakes up **all** threads waiting on the object's monitor; they compete for the lock.

Mechanism:

- Both threads must use a **shared object** as a communication channel.
- Invocations of `wait()`, `notify()`, and `notifyAll()` must occur **inside** a `synchronized` block or method.
- After `notify()`/`notifyAll()`, the awakened thread(s) must re-acquire the lock before proceeding.

Use Case Example (Producer–Consumer):

```
class Drop {
    private String message;
    private boolean empty = true;

    public synchronized String take() {
        while (empty) {
            try { wait(); } catch (InterruptedException e) {}
        }
        empty = true;
        notifyAll();
        return message;
    }

    public synchronized void put(String msg) {
        while (!empty) {
            try { wait(); } catch (InterruptedException e) {}
        }
        empty = false;
        message = msg;
        notifyAll();
    }
}

class Producer implements Runnable {
    private Drop drop;
    public Producer(Drop d) { drop = d; }
    public void run() {
```

```
        String[] msgs = { "A", "B", "C" };
        for (String m : msgs) drop.put(m);
    }
}

class Consumer implements Runnable {
    private Drop drop;
    public Consumer(Drop d) { drop = d; }
    public void run() {
        for (int i = 0; i < 3; i++)
            System.out.println("Consumed: " + drop.take());
    }
}

public class PCDemo {
    public static void main(String[] args) {
        Drop drop = new Drop();
        new Thread(new Producer(drop)).start();
        new Thread(new Consumer(drop)).start();
    }
}
```

Explanation:

- `Producer` calls `put()`, sets the message, then calls `notifyAll()`.
- `Consumer` waits in `take()` until `put()` calls `notifyAll()`.
- This ensures threads communicate smoothly without busy-waiting.

8. Synchronizing Threads

Introduction:

In Java, **synchronization** is used to **control access to shared resources** by multiple threads. It prevents **race conditions**, where two or more threads access shared data at the same time and produce inconsistent results.

Why Synchronization is Needed:

- Java is **multi-threaded** by nature.
- When two threads share a resource (e.g., a variable, method, or object), they might interfere with each other's operations.
- **Synchronization** ensures that only **one thread accesses** the resource at a time.

Synchronized Methods:

You can declare a method as `synchronized` so that only one thread can execute it at a time on the same object.

Example:

```
class Table {
    synchronized void printTable(int n) {
        for (int i = 1; i <= 5; i++) {
            System.out.println(n * i);
            try { Thread.sleep(400); } catch (Exception e) {}
        }
    }
}

class MyThread1 extends Thread {
    Table t;
    MyThread1(Table t) { this.t = t; }
    public void run() { t.printTable(5); }
}

class MyThread2 extends Thread {
    Table t;
    MyThread2(Table t) { this.t = t; }
    public void run() { t.printTable(100); }
}
```

```
public class SyncDemo {
    public static void main(String[] args) {
        Table obj = new Table();
        new MyThread1(obj).start();
        new MyThread2(obj).start();
    }
}
```

Output (synchronized):

```
5
10
15
20
25
100
200
300
400
500
```

Without synchronization, the output may interleave and become unreadable.

Synchronized Blocks:

Used when we want to synchronize **only part of a method**.

Syntax:

```
synchronized (object) {
    // critical section
}
```

Static Synchronization:

Used to synchronize static methods. Lock is on the **class**, not on the object.

java
CopyEdit

```
synchronized static void method() {
    // critical code
}
```

Summary:

- Use `synchronized` to avoid race conditions.
- Apply to methods or blocks depending on scope.
- Ensures data consistency and thread-safe access to shared resources.

9. Autoboxing

Introduction:

Autoboxing is the automatic conversion that the Java compiler makes between **primitive types** and their corresponding **wrapper classes**.
This feature was introduced in **Java 5** to simplify the code and make it more readable.

What is Autoboxing?

- It is the process of **converting a primitive data type into its corresponding wrapper class** object automatically.

Primitive	Wrapper Class
`int`	`Integer`
`char`	`Character`
`double`	`Double`

`float`	`Float`
`boolean`	`Boolean`

Example:

```
public class AutoBoxingExample {
    public static void main(String[] args) {
        int num = 10;

        // Autoboxing: primitive to object
        Integer obj = num;

        // Using wrapper class method
        System.out.println("Square: " + obj * obj);
    }
}
```

Output:

```
Square: 100
```

Here, `int num` is automatically converted to an `Integer` object by the compiler.

Unboxing:

The reverse process—**converting an object of a wrapper class back into a primitive**.

```
Integer obj = 100; // Autoboxing
int num = obj;     // Unboxing
```

Where Autoboxing is Used:

1. **Collections (like ArrayList):**

```
     ArrayList<Integer> list = new ArrayList<>();
list.add(10); // Autoboxing: int → Integer
int x = list.get(0); // Unboxing: Integer → int
```

1. **Method Calls:**
   Methods expecting wrapper objects can receive primitives.

2. **Generics:**
   Java generics work only with objects, so autoboxing allows primitives to be used seamlessly.

Advantages:

- Reduces boilerplate code.
- Improves readability.
- Makes Java more intuitive.

Caution:

- Can cause performance issues if used in large-scale loops or performance-critical code due to object creation.

Conclusion:

Autoboxing and unboxing help bridge the gap between primitives and objects in Java, especially when working with collections and generics.

10. Concept of `wait()`, `notify()`, and `notifyAll()` Methods (with Program)

Introduction:

The methods **`wait()`**, **`notify()`**, and **`notifyAll()`** are used for **inter-thread communication** in Java. They belong to the `Object` class and must be called from within a **synchronized context**.

Purpose:

They allow threads to:

- **Wait** (pause execution) until a certain condition is met.
- **Notify** other threads once that condition has been fulfilled.

Method Descriptions:

Method	Description
`wait()`	Causes the current thread to wait and release the lock until notified.
`notify()`	Wakes up **one** thread waiting on the object's monitor.
`notifyAll()`	Wakes up **all** threads waiting on the object's monitor.

Rules:

- Must be called inside a `synchronized` block or method.
- The thread must own the object's monitor; otherwise, it throws `IllegalMonitorStateException`.

Example Program – Producer-Consumer Using `wait()` & `notify()`

```
class Shared {
    int num;
    boolean valueSet = false;
```

```
    synchronized void put(int n) {
        while (valueSet) {
            try { wait(); } catch (Exception e) {}
        }
        num = n;
        System.out.println("Produced: " + num);
        valueSet = true;
        notify();
    }

    synchronized void get() {
        while (!valueSet) {
            try { wait(); } catch (Exception e) {}
        }
        System.out.println("Consumed: " + num);
        valueSet = false;
        notify();
    }
}

class Producer extends Thread {
    Shared s;
    Producer(Shared s) { this.s = s; }
    public void run() {
        for (int i = 1; i <= 5; i++) {
            s.put(i);
        }
    }
}

class Consumer extends Thread {
    Shared s;
    Consumer(Shared s) { this.s = s; }
    public void run() {
        for (int i = 1; i <= 5; i++) {
            s.get();
        }
```

```
    }
}

public class ThreadCommDemo {
    public static void main(String[] args) {
        Shared obj = new Shared();
        new Producer(obj).start();
        new Consumer(obj).start();
    }
}
```

Output:

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
...
```

When to Use:

- `wait()` → When a thread must pause until a condition is true.
- `notify()` → When a thread signals that the condition may now be true.
- `notifyAll()` → When multiple threads are waiting, and any one of them can proceed.

Conclusion:

These methods are essential for **cooperative multitasking**, allowing efficient and safe communication between threads while maintaining proper synchronization.

11. Thread Priorities

Introduction:

In Java, each thread is assigned a **priority** that helps the thread scheduler decide the order in which threads are executed. Threads with higher priority are generally executed before threads with lower priority.

Thread Priority Range:

- Java defines thread priorities in the range of **1 to 10**.
- Constants in `Thread` class:
    - `MIN_PRIORITY` = 1
    - `NORM_PRIORITY` = 5 (default)
    - `MAX_PRIORITY` = 10

How to Set and Get Thread Priority:

```
Thread t = new Thread();

t.setPriority(Thread.MAX_PRIORITY);  // Set highest priority

int p = t.getPriority();             // Get priority
```

How Priorities Affect Execution:

- The **thread scheduler** uses priorities to allocate CPU time.
- Threads with higher priority are usually given preference.
- However, **priority scheduling is platform-dependent** and not guaranteed to always run higher-priority threads first.

- Priorities help improve the responsiveness of important threads but should not be solely relied upon for program correctness.

Example:

```
class MyThread extends Thread {

    public void run() {

        System.out.println(getName() + " Priority: " + getPriority());

    }

}


public class ThreadPriorityDemo {

    public static void main(String[] args) {

        MyThread t1 = new MyThread();

        MyThread t2 = new MyThread();


        t1.setPriority(Thread.MIN_PRIORITY);  // 1

        t2.setPriority(Thread.MAX_PRIORITY);  // 10


        t1.start();

        t2.start();

    }

}
```

Important Points:

- **Default priority** of a new thread is the priority of the creating thread.
- Priority only affects **thread scheduling**, not the actual execution order.
- Use priorities carefully to avoid **starvation** of low priority threads.

Conclusion:

Thread priorities provide a way to hint to the JVM scheduler about the importance of threads but should be used judiciously alongside proper synchronization techniques.

MODULE-5

1. The Origins of Swing

Swing was introduced by Sun Microsystems as part of the Java Foundation Classes (JFC) in Java 1.2 (1998). It was developed to overcome the limitations of the Abstract Window Toolkit (AWT), which depended heavily on the native operating system's graphical user interface (GUI) components. AWT components were heavyweight and inconsistent across platforms because they used native peers. Swing, in contrast, provides a **lightweight, platform-independent GUI toolkit** written entirely in Java. This allows applications to have a consistent look and feel across all platforms, improved customizability, and richer UI components.

Key Points:

- Introduced in Java 1.2 as part of JFC.
- Replaced AWT's heavyweight components with lightweight Java components.
- Allows for greater customization and consistent cross-platform UI.
- Supports pluggable look and feel for UI flexibility.

2. Swing Is Built on the AWT

Swing builds on AWT's core architecture but replaces many native components with **lightweight components**. Swing uses AWT for its event handling, drawing surface, and window management but implements its components purely in Java, which means they are

rendered by Java rather than the native OS. This leads to greater control over appearance and behavior.

How Swing uses AWT:

- AWT provides the basic **windowing toolkit**, event queue, and input event processing.
- Swing components are Java classes extending from AWT's `Component` class but override painting methods.
- Swing uses AWT's **top-level containers** like `JFrame`, which in turn use native peers for window borders and decorations.

Benefits:

- Consistent UI appearance.
- Customizable and extendable components.
- Enables the "pluggable look and feel" mechanism.

3. Two Key Features of Swing: Lightweight Components and Pluggable Look and Feel

1. **Lightweight Components:**
    - Swing components are written purely in Java and do not rely on native operating system widgets.
    - They are called lightweight because they do not have a native "peer" (an associated OS widget).
    - This gives developers more control over their behavior and appearance.
    - Examples: `JButton`, `JLabel`, `JPanel`.
2. **Pluggable Look and Feel:**
    - Swing allows applications to change their appearance without altering the program logic.

- Developers can switch between different “look and feel” implementations such as:
  - Metal (default cross-platform),
  - Nimbus (modern, stylish UI),
  - System (matches OS native look like Windows or Mac).
- This is done via the `UIManager.setLookAndFeel()` method.
- It enhances user experience by allowing the GUI to blend with native OS aesthetics or a custom theme.

4. Different Types of Buttons Available in Java Swing with Program

Swing provides several types of buttons for different purposes:

- **JButton:** Standard push button. Triggers an action when clicked.
- **JToggleButton:** A button with two states, pressed or not pressed.
- **JCheckBox:** A box that can be checked or unchecked independently.
- **JRadioButton:** Used in groups to allow single selection among options.

Example:

```
import javax.swing.*;
import java.awt.*;

public class ButtonTypesExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Swing Buttons");
        frame.setLayout(new FlowLayout());
        frame.setSize(350, 150);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton button = new JButton("JButton");
        JToggleButton toggleButton = new
JToggleButton("JToggleButton");
        JCheckBox checkBox = new JCheckBox("JCheckBox");
```

```
        JRadioButton radioButton = new JRadioButton("JRadioButton");

        frame.add(button);
        frame.add(toggleButton);
        frame.add(checkBox);
        frame.add(radioButton);

        frame.setVisible(true);
    }
}
```

5. Differentiate Between Components and Containers in Swing

Feature	Component	Container
Definition	Basic UI element such as button, label, or text field.	Special type of component that holds and manages other components.
Function	Represents a single GUI widget.	Organizes and manages layout of child components.
Hierarchy	Usually leaf nodes in the GUI hierarchy.	Parent nodes that contain other components or containers.
Examples	`JButton`, `JLabel`, `JTextField`	`JFrame`, `JPanel`, `JDialog`
Role in UI	Displays interactive or display elements.	Provides structure and layout management.

Explanation:
Components are individual UI elements, while containers are used to hold multiple components and define their layout (like flow, grid, border). Containers can also be nested.

6. The Swing Packages

Swing APIs are grouped into several packages:

- **`javax.swing`**: Core Swing components like `JFrame`, `JButton`, `JLabel`, `JTextField`.
- **`javax.swing.event`**: Event classes and listeners specific to Swing, such as `ListSelectionListener` and `ChangeListener`.

- **`javax.swing.border`**: Classes for adding decorative borders to components (`LineBorder`, `EtchedBorder`).

- **`javax.swing.plaf`**: Provides the **Pluggable Look and Feel (PLAF)** architecture, interfaces, and classes to customize UI appearance.

- **`javax.swing.table` and `javax.swing.tree`**: Support for JTable and JTree components.

7. A Simple Swing Application

A basic Swing application requires creating a window and adding components.

```
import javax.swing.*;

public class SimpleSwingApp {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Simple Swing Application");
        JLabel label = new JLabel("Welcome to Swing!");

        frame.add(label);                       // Add label to frame
        frame.setSize(300, 150);                // Set frame size
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  // Exit
on close
        frame.setVisible(true);                  // Show frame
    }
}
```

Explanation:

- `JFrame` is a top-level window.

- `JLabel` displays simple text.

- `setVisible(true)` makes the window appear.

8. Containers with Example

Containers hold other components and manage their layout. Common containers are `JFrame` (top-level window) and `JPanel` (generic container).

Example:

```
import javax.swing.*;
import java.awt.*;


public class ContainerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Container Example");
        JPanel panel = new JPanel();
        panel.setLayout(new FlowLayout());

        JButton btn1 = new JButton("Button 1");
        JButton btn2 = new JButton("Button 2");

        panel.add(btn1);
        panel.add(btn2);
        frame.add(panel);                       // Add panel to frame

        frame.setSize(300, 150);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Explanation:

- `JPanel` organizes buttons with FlowLayout.
- Panel added to JFrame.
- Containers help in organizing complex GUIs by grouping components.

---

9. Java Program Using Swing Demonstrating JLabel, ImageIcon, and JTextField

```
import javax.swing.*;

import java.awt.*;


public class LabelIconTextFieldDemo {

    public static void main(String[] args) {

        JFrame frame = new JFrame("JLabel, ImageIcon & JTextField
Demo");

        frame.setLayout(new FlowLayout());

        frame.setSize(400, 200);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);


        ImageIcon icon = new ImageIcon("icon.png"); // Image in
project directory

        JLabel label = new JLabel("User Name:", icon, JLabel.LEFT);

        JTextField textField = new JTextField(15);
```

```
        frame.add(label);
        frame.add(textField);
        frame.setVisible(true);
    }
}
```

Explanation:

- `JLabel` shows text and an icon.
- `ImageIcon` loads an image.
- `JTextField` allows user input.

---

10. Java Program Simple Swing Application that Creates a Basic GUI Window Using JFrame

```
import javax.swing.*;

public class BasicSwingWindow {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Basic GUI Window");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
```

```
}
```

Explanation:

- Creates a basic window frame.
- No components inside, but window shows on screen.
- `EXIT_ON_CLOSE` terminates program when window is closed.

---

11. Concept of JList, JComboBox and JScrollPane with Java Program

- **JList:** Displays a list of items; allows selection(s).
- **JComboBox:** Drop-down menu to select one item.
- **JScrollPane:** Adds scrollbars when content exceeds visible space.

Program:

```
import javax.swing.*;

import java.awt.*;


public class ListComboScrollDemo {

    public static void main(String[] args) {

        JFrame frame = new JFrame("JList, JComboBox & JScrollPane
Demo");

        frame.setLayout(new FlowLayout());

        frame.setSize(400, 250);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        String[] items = {"Apple", "Banana", "Cherry", "Date",
"Elderberry"};


        JList<String> list = new JList<>(items);

        JScrollPane listScroll = new JScrollPane(list);

        listScroll.setPreferredSize(new Dimension(120, 80));


        JComboBox<String> comboBox = new JComboBox<>(items);


        frame.add(listScroll);

        frame.add(comboBox);

        frame.setVisible(true);

    }

}
```

Explanation:

- `JList` shows multiple items; `JScrollPane` ensures it's scrollable.
- `JComboBox` shows a drop-down menu for single selection.