Notes UNIT 3

Global Institute of Technology and management

Subject JAVA

Topics: Method Overriding In Java, Object Class in Java, Runtime Polymorphism in Java

Date 25/02/25

1. Method Overriding in Java

Method overriding is a feature in Java that allows a subclass to provide a specific implementation of a method that is already defined in its parent class.

Rules for Method Overriding:

1. The method must have the same name as in the parent class.
2. The method must have the same parameter list as in the parent class.
3. There must be an **inheritance relationship** (i.e., `extends` keyword is used).
4. The return type should be the same (or covariant return type in Java 5+).
5. **Static methods cannot be overridden**, but they can be re-declared (method hiding).

Example of Method Overriding:

```
class Parent {
    void display() {
        System.out.println("Display method in Parent class");
    }
}

class Child extends Parent {
    @Override
    void display() {  // Overriding the method
```

```
        System.out.println("Display method in Child class");
    }
}

public class MethodOverridingExample {
    public static void main(String args[]) {
        Parent obj = new Child(); // Upcasting
        obj.display(); // Calls the overridden method in Child class
    }
}
```

Output:

```
Display method in Child class
```

Explaination

Parent is a superclass that has a method display(), which prints "Display method in Parent class".
Child is a subclass of Parent (extends Parent).
It **overrides** the display() method to provide its own implementation.
The @Override annotation ensures that we are correctly overriding a method from the superclass.
A **reference variable of type Parent (obj) is assigned an object of Child**. This is **upcasting** (Parent obj = new Child();).
Since display() is overridden in Child, the overridden method **in the Child class is executed**, even though the reference type is Parent

2. Object Class in Java

In Java, **`Object` class is the parent class of all classes** (either directly or indirectly). It is located in `java.lang` package.

Important Methods in the Object Class:

Method	Description
`toString()`	Returns a string representation of the object.
`equals(Object obj)`	Compares two objects for equality.
`hashCode()`	Returns the hash code value of an object.
`clone()`	Creates a copy of the object (requires `Cloneable` interface).
`getClass()`	Returns the runtime class of the object.

Method	Description
`finalize()`	Called by the garbage collector before destroying the object.

Example of `toString()` and `equals()`:

```
class Person {
    String name;

    Person(String name) {
        this.name = name;
    }

    @Override
    public String toString() {  // Overriding toString() method
        return "Person Name: " + name;
    }

    @Override
    public boolean equals(Object obj) { // Overriding equals() method
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Person person = (Person) obj;
        return this.name.equals(person.name);
    }
}

public class ObjectClassExample {
    public static void main(String[] args) {
        Person p1 = new Person("John");
        Person p2 = new Person("John");

        System.out.println(p1.toString()); // Calls overridden toString()
        System.out.println(p1.equals(p2)); // Calls overridden equals()
    }
}
```

Output:

```
Person Name: John
True
```

Step-by-Step Explanation

- The `Person` class has an **instance variable** `name`.
- The **constructor** initializes `name` when a `Person` object is created.
- The `toString()` method is **overridden** from the `Object` class.
- Instead of printing the default object reference (`Person@hashcode`), it returns

`"Person Name: name"`, providing a meaningful representation of the object.

- **`this == obj`** → If both references point to the same object, return `true`.
- **`obj == null || getClass() != obj.getClass()`** → If `obj` is `null` or not of type `Person`, return `false`.
- **Typecasting (`Person person = (Person) obj;`)** → Converts `obj` into a `Person` object.
- **`this.name.equals(person.name)`** → Compares the `name` of both objects for equality.

3. Runtime Polymorphism in Java

Runtime Polymorphism (Dynamic Method Dispatch) is when the method that gets called is determined at **runtime**, not at compile time. This happens through **method overriding** and **upcasting**.

Example of Runtime Polymorphism:

```
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Cat meows");
    }
}

public class RuntimePolymorphismExample {
    public static void main(String[] args) {
        Animal a1 = new Dog(); // Upcasting
        Animal a2 = new Cat(); // Upcasting

        a1.makeSound(); // Calls Dog's makeSound()
        a2.makeSound(); // Calls Cat's makeSound()
    }
}
```

Output:

```
Dog barks
```

```
Cat meows
```

Compile-Time vs. Runtime Polymorphism

Feature	Compile-Time Polymorphism (Method Overloading)	Runtime Polymorphism (Method Overriding)
Definition	Multiple methods with the same name but different parameters	Subclass provides a specific implementation of a method in the parent class
Binding	Resolved at **compile-time**	Resolved at **runtime**
Achieved by	**Method Overloading**	**Method Overriding**
Upcasting Needed?	□ No	□ Yes (`Parent obj = new Child();`)
Flexibility	Less flexible	More flexible
Example	`sum(int a, int b)` and `sum(int a, int b, int c)`	`show()` overridden in a subclass

Key Differences in a Nutshell Method Overloading, Method Overriding

1. **Method Overloading** → Same method name, **different** parameters (within the same class).
2. **Method Overriding** → Same method name, **same** parameters (in parent and child classes).
3. **Overloading is compile-time polymorphism**, while **Overriding is runtime polymorphism**.
4. **Static methods can be overloaded but not overridden**.

Subject Java

Darte 27/02/2025

Topic

1. Interfaces in Object-Oriented Programming

An **interface** is a contract that defines a set of methods that a class must implement. It does not contain any implementation details; it only declares method signatures. Interfaces promote abstraction and multiple inheritance by allowing classes to implement multiple interfaces.

Key Features of Interfaces

- **Method Signatures Only:** No method bodies, just declarations.
- **Multiple Inheritance Support:** A class can implement multiple interfaces.
- **Loose Coupling:** Encourages separation of concerns in code design.
- **Encapsulation of Behavior:** Allows different classes to share behavior without enforcing an inheritance hierarchy.

Example of an Interface in Java

```
interface Animal {
    void makeSound();
}

class Dog implements Animal {
    public void makeSound() {
        System.out.println("Bark!");
    }
}

class Cat implements Animal {
    public void makeSound() {
        System.out.println("Meow!");
    }
}

public class InterfaceExample {
    public static void main(String[] args) {
        Animal dog = new Dog();
        Animal cat = new Cat();
        dog.makeSound();
        cat.makeSound();
    }
}
```

Output:

```
Bark!
Meow!
```

Code Explanation

1. **Interface Declaration (`Animal`)**
   - The `Animal` interface defines a contract with a single method `makeSound()`.
   - Any class implementing this interface must provide an implementation for `makeSound()`.
2. **Class Implementations (`Dog` and `Cat`)**
   - `Dog` and `Cat` classes implement the `Animal` interface.
   - Each class provides its own version of `makeSound()`.
3. **Main Method (`InterfaceExample`)**

- Objects of `Dog` and `Cat` are created but referenced by the `Animal` type.
- The overridden `makeSound()` method is called on each object.

Expected Output

```
Bark!
Meow!
```

Advantages of Using Interfaces Here

- **Encapsulation of Behavior**: Different animal types share a common interface but define their own behaviors.
- **Polymorphism**: The `makeSound()` method is called dynamically based on the actual object type.
- **Loose Coupling**: The `InterfaceExample` class does not depend on concrete implementations (`Dog` or `Cat`), making the code more flexible.

Nested Classes in Java

A **nested class** is a class that is defined within another class. It helps in logically grouping classes that are only used in one place, increasing encapsulation, and improving code readability.

Types of Nested Classes:

1. **Static Nested Class** – Defined using the `static` keyword.
2. **Non-static Inner Class** – A regular inner class that depends on an instance of the outer class.
3. **Method-local Inner Class** – Defined inside a method and is only accessible within that method.
4. **Anonymous Inner Class** – A class without a name that is instantiated in place.

Inner Classes in Java

An **inner class** is a class defined inside another class and has access to all of its enclosing class's members, including private members.

Example of a Non-Static Inner Class

```
class Outer {
    private String message = "Hello from Outer class!";
```

```
    class Inner {
        void display() {
            System.out.println(message);
        }
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.display();
    }
}
```

Static Nested Class Example

```
class Outer {
    static class StaticNested {
        void display() {
            System.out.println("Inside static nested class");
        }
    }

    public static void main(String[] args) {
        Outer.StaticNested nested = new Outer.StaticNested();
        nested.display();
    }
}
```

Anonymous Inner Classes

An **anonymous inner class** is a class that is declared and instantiated at the same time. It is useful when a class is needed only once.

Example Using an Interface

```
interface Greeting {
    void sayHello();
}

public class AnonymousInnerClassExample {
    public static void main(String[] args) {
        Greeting greeting = new Greeting() {
            public void sayHello() {
                System.out.println("Hello from Anonymous Inner Class!");
            }
        };
        greeting.sayHello();
    }
}
```

Example Using a Class

```
java
CopyEdit
abstract class Animal {
    abstract void makeSound();
}

public class AnonymousInnerExample {
    public static void main(String[] args) {
        Animal dog = new Animal() {
            void makeSound() {
                System.out.println("Woof Woof!");
            }
        };
        dog.makeSound();
    }
}
```

StringBuffer Class in Java

The `StringBuffer` class in Java is used for creating mutable (modifiable) string objects. It is thread-safe and provides better performance than `String` when frequent modifications are required.

Key Features of StringBuffer

1. **Mutable** – Strings can be modified.
2. **Thread-safe** – Synchronized methods ensure safety in a multi-threaded environment.
3. **Faster than String when performing many modifications.**

Common Methods in StringBuffer

Method	Description
`append(String s)`	Appends the specified string to the end.
`insert(int offset, String s)`	Inserts the string at the specified position.
`replace(int start, int end, String s)`	Replaces text between given indices.
`delete(int start, int end)`	Deletes characters from the given range.
`reverse()`	Reverses the string.
`capacity()`	Returns the current capacity.
`length()`	Returns the length of the string.

Example

```
public class StringBufferExample {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello");
```

```
        sb.append(" World");
        System.out.println(sb); // Hello World

        sb.insert(5, ",");
        System.out.println(sb); // Hello, World

        sb.replace(6, 12, "Java");
        System.out.println(sb); // Hello, Java

        sb.reverse();
        System.out.println(sb); // avaJ ,olleH

        System.out.println("Capacity: " + sb.capacity()); // Default is 16 +
length of string
    }
}
```

Conclusion

- **Nested Classes** provide a way to logically group classes.
- **Inner Classes** have direct access to outer class members.
- **Anonymous Inner Classes** allow defining and using a class at the same time.
- **StringBuffer** is a mutable and thread-safe alternative to `String` for frequent string modifications.

Multithreading in Java 🚀

What is Multithreading?

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

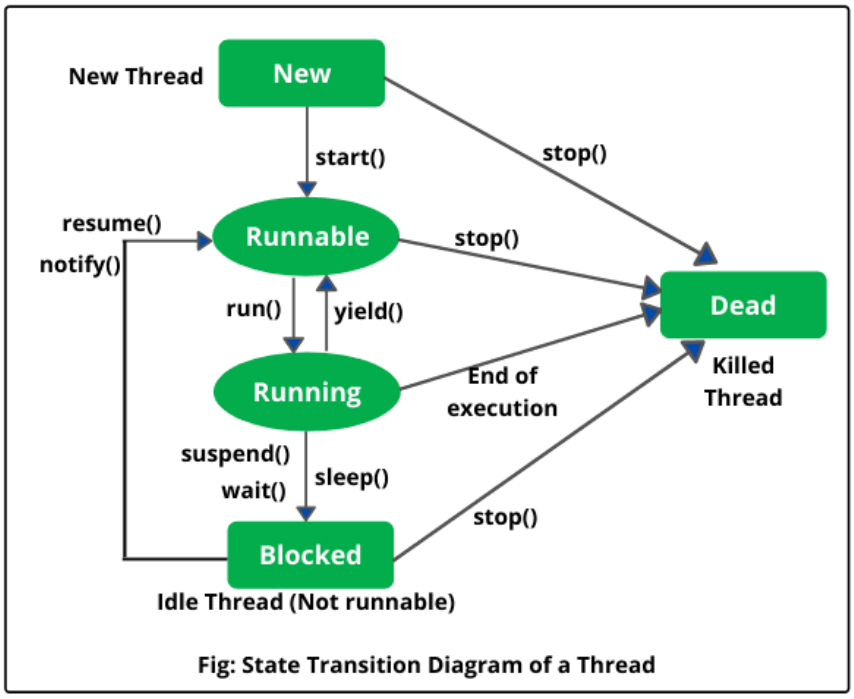


Fig: State Transition Diagram of a Thread

Web Browser (Handling Multiple Tasks Concurrently)

When you use a web browser, multiple threads work together:

- **Thread 1:** Handles user input (clicking, typing)
- **Thread 2:** Loads web pages
- **Thread 3:** Plays videos in a separate tab
- **Thread 4:** Downloads files in the background

Without multithreading, the browser would freeze while performing each task sequentially.

Threads can be created by using **two mechanisms** :

Extending the Thread class

Implementing the Runnable Interface

Thread creation by extending the Thread class
We create a class that extends the java.lang.Thread class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

---

1. Creating Threads in Java

Java provides two ways to create a thread:

A. Extending the `Thread` Class

```
class MyThread extends Thread {
    public void run() { // Override run() method
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + i);
            try { Thread.sleep(1000); } catch (InterruptedException e) {
e.printStackTrace(); }
        }
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.start(); // Starts thread 1
        t2.start(); // Starts thread 2
    }
}
```

 Each thread runs independently.

B. Thread creation by implementing the Runnable Interface

We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object

Implementing the `Runnable` Interface (Preferred Approach)

```
class MyRunnable implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + i);
        }
    }
}

public class RunnableExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        Thread t2 = new Thread(new MyRunnable());
```

```
        t1.start();
        t2.start();
    }
}
```

 Preferred approach since Java supports multiple inheritance with interfaces.

Thread Priority in Java

Thread priority determines the order in which threads are scheduled for execution. Each thread in Java is assigned a **priority value** between **1 (MIN_PRIORITY)** and **10 (MAX_PRIORITY)**. The default priority is **5 (NORM_PRIORITY)**.

Setting and Getting Thread Priority

```
class MyThread extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName() + " Priority: " +
Thread.currentThread().getPriority());
    }
}

public class ThreadPriorityExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();

        // Setting thread priority
        t1.setPriority(Thread.MIN_PRIORITY);  // Priority 1
        t2.setPriority(Thread.NORM_PRIORITY); // Priority 5 (Default)
        t3.setPriority(Thread.MAX_PRIORITY);  // Priority 10

        t1.start();
        t2.start();
        t3.start();
    }
}
```

Key Points:

- **Higher priority does not guarantee execution order**, it only increases the thread's chance of getting CPU time.
- **Thread scheduling is OS-dependent**, so actual execution order may vary.

Blocked States in Java

A thread can enter the **BLOCKED state** when it is waiting for a resource that another thread is holding. This typically happens when:

- A thread tries to enter a synchronized block/method, but another thread is already inside.
- A thread waits for a lock but another thread is holding it.

Thread States in Java

State	Description
NEW	Thread is created but not started (`new Thread()`).
RUNNABLE	Thread is ready to run but waiting for CPU time.
RUNNING	Thread is executing.
BLOCKED	Thread is waiting for a monitor lock (another thread is holding it).
WAITING	Thread is waiting indefinitely for another thread's signal.
TIMED_WAITING	Thread is waiting for a specified time (`sleep()`, `join()`).
TERMINATED	Thread has finished execution.

Example of Blocked State

```
class SharedResource {
    synchronized void accessResource() {
        System.out.println(Thread.currentThread().getName() + " is accessing
the resource...");
        try {
            Thread.sleep(3000); // Simulate processing
        } catch (InterruptedException e) {
            System.out.println(e);
        }
        System.out.println(Thread.currentThread().getName() + " has finished
execution.");
    }
}

class MyThread extends Thread {
    SharedResource resource;

    MyThread(SharedResource resource) {
```

```
        this.resource = resource;
    }

    public void run() {
        resource.accessResource();
    }
}

public class BlockedStateExample {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();

        Thread t1 = new MyThread(resource);
        Thread t2 = new MyThread(resource);

        t1.start();
        t2.start();
    }
}
```

Explanation:

1. `t1` starts execution and acquires the lock on `accessResource()`.
2. `t2` tries to enter `accessResource()`, but it gets **BLOCKED** because `t1` is still using the synchronized method.
3. `t2` remains in the **BLOCKED state** until `t1` completes execution and releases the lock.

Thread Synchronization in Java

Thread synchronization is a mechanism that ensures only one thread can access a shared resource at a time, preventing **race conditions** and **data inconsistency**.

Types of Synchronization

1. **Synchronized Methods** – The entire method is locked, allowing only one thread to execute it at a time.
2. **Synchronized Blocks** – Only a specific block of code is synchronized, improving performance by reducing the locked section.
3. **Static Synchronization** – Synchronization applied to static methods, ensuring only one thread accesses the method at the class level.

Synchronized Methods

- Declaring a method as `synchronized` ensures that only one thread executes it at a time.
- It locks the **entire method**, reducing concurrency but ensuring thread safety.

Synchronized Blocks

- Instead of locking the entire method, a **specific section** of code is synchronized.
- This allows better performance while still preventing race conditions.

Static Synchronization

- When a `static synchronized` method is used, the **class itself is locked**, meaning only one thread can access the method across all instances.
- Useful when shared data belongs to the class rather than an object.

Thread Blocking in Synchronization

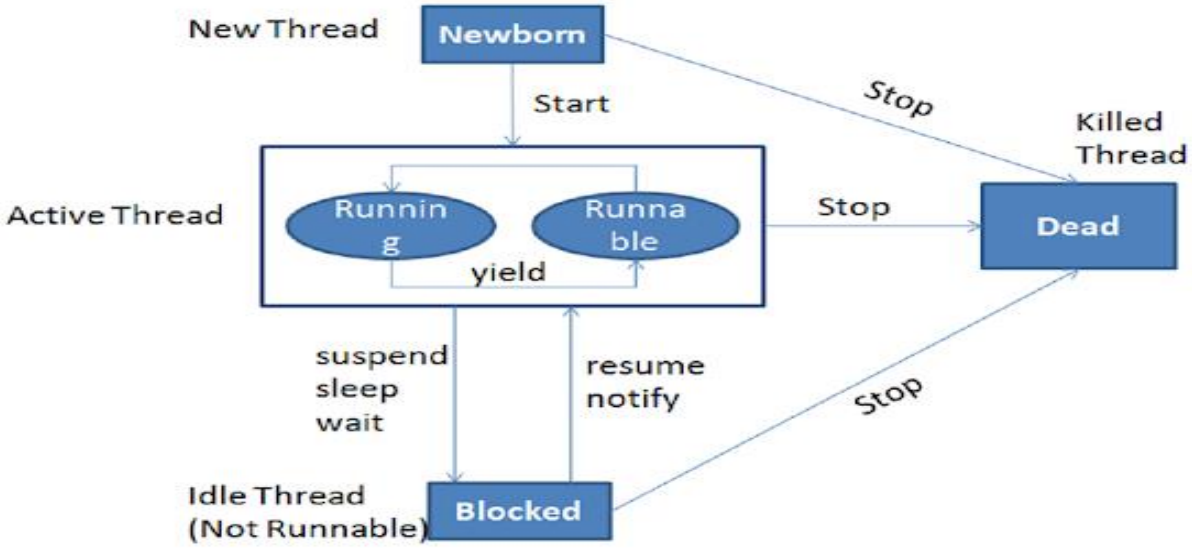
- If one thread is executing a synchronized method, other threads trying to access the same method must **wait** until the first thread releases the lock.
- This waiting thread enters the **BLOCKED state**.

  ############

- Synchronization helps in avoiding **data inconsistency** in multithreaded applications.
- Using **synchronized methods** or **synchronized blocks** ensures safe execution but may reduce performance.
- **Static synchronization** is useful when shared data is at the class level.

Thread Communication in Java

Thread communication in Java allows multiple threads to communicate efficiently while sharing a common resource. The **wait(), notify(), and notifyAll()** methods from the `Object` class play a crucial role in this process.



How to Stop a Thread in Java? Use volatile

Methods for Thread Communication

1. **wait()**
   - Causes the current thread to wait until another thread calls `notify()` or `notifyAll()`.
   - The thread releases the lock on the object and enters the **WAITING state**.
   - It must be called inside a **synchronized block or method**.
2. **notify()**
   - Wakes up **one** thread waiting on the same object's monitor.
   - The thread remains in the **runnable state**, waiting for the lock to be released.
3. **notifyAll()**
   - Wakes up **all** threads waiting on the object's monitor.
   - Only one thread will get the lock and execute at a time.

Working Mechanism

1. A thread calls `wait()`, releases the lock, and enters the **waiting state**.
2. Another thread calls `notify()` or `notifyAll()`, signaling a waiting thread to continue execution.
3. The notified thread moves to the **runnable state**, waiting to regain the lock.
4. Once it gets the lock, it resumes execution.

Key Points

- **`wait()`, `notify()`, and `notifyAll()` must be called within a synchronized block or method.**
- **Only objects have monitors, not primitive data types.**
- **Calling these methods outside a synchronized context results in `IllegalMonitorStateException`.**

When to Use?

- **Producer-Consumer Problem**: A producer thread produces data, and a consumer thread waits until data is available.
- **Inter-thread communication**: Ensuring multiple threads work in coordination without busy-waiting.

Cse 4th sem

Date 04/03/25

Swing Class Hierarchy in Java

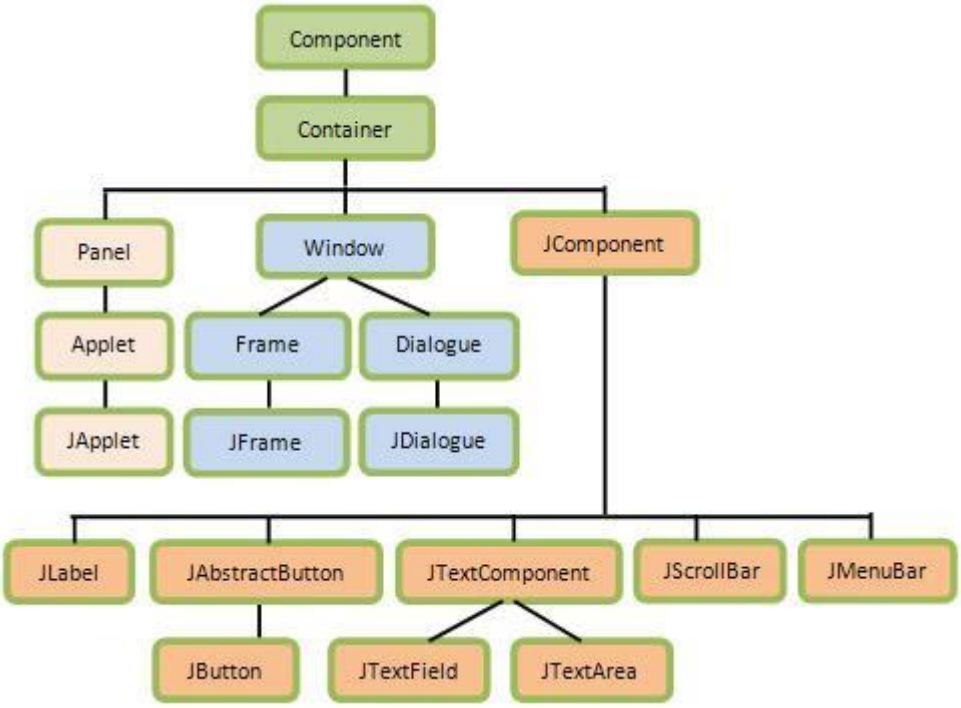
Swing is a part of Java's **Java Foundation Classes (JFC)** that provides a rich set of GUI components. It is built on top of AWT (Abstract Window Toolkit) but offers more flexible and lightweight components.

1. Overview of Swing Hierarchy

Swing components are part of the `javax.swing` package and are derived from **AWT (Abstract Window Toolkit)**. The hierarchy starts from `Object`, followed by `Component`, `Container`, and then Swing-specific components.

---

2. Swing Class Hierarchy Diagram



Below is a simplified hierarchy of Swing components:

```
java.lang.Object
    ├── java.awt.Component
    │       ├── java.awt.Container
    │       │       ├── javax.swing.JComponent
    │       │       │       ├── javax.swing.JLabel
    │       │       │       ├── javax.swing.JButton
    │       │       │       ├── javax.swing.JCheckBox
    │       │       │       ├── javax.swing.JRadioButton
    │       │       │       ├── javax.swing.JTextField
    │       │       │       ├── javax.swing.JTextArea
    │       │       │       ├── javax.swing.JList
    │       │       │       ├── javax.swing.JComboBox
    │       │       │       ├── javax.swing.JSlider
    │       │       │       ├── javax.swing.JScrollBar
    │       │       │       ├── javax.swing.JTable
    │       │       │       ├── javax.swing.JTree
    │       │       │       ├── javax.swing.JTabbedPane
    │       │       │       ├── javax.swing.JToolBar
    │       │       │       ├── javax.swing.JProgressBar
    │       │       │       ├── javax.swing.JDesktopPane
    │       │       │       ├── javax.swing.JInternalFrame
    │       │       │       ├── javax.swing.JLayeredPane
    │       │       │       ├── javax.swing.JSplitPane
    │       │       │       └── javax.swing.JScrollPane
    │       │       ├── javax.swing.JFrame
    │       │       ├── javax.swing.JDialog
    │       │       ├── javax.swing.JWindow
```

```
|     |     ├── javax.swing.JApplet
|     |     ├── javax.swing.JPanel
|     |     ├── javax.swing.JRootPane
|     |     ├── javax.swing.JViewport
```

---

3. Explanation of Key Swing Classes

3.1 `java.awt.Component`

- This is the base class for all UI elements (both AWT and Swing).
- Provides fundamental properties like size, position, visibility, etc.

3.2 `java.awt.Container`

- A subclass of `Component` that can hold other components.
- Examples: `JFrame`, `JPanel`, `JDialog`.

3.3 `javax.swing.JComponent`

- Base class for most Swing components.
- Provides advanced features like **tooltips, double buffering, and borders**.

---

4. Common Swing Components

4.1 Top-Level Containers

These are the main containers that hold other Swing components.

- **`JFrame`**: Standard window with a title bar.
- **`JDialog`**: Pop-up window for user interaction.
- **`JWindow`**: Window without a title bar.

4.2 Controls (User Input Components)

- **`JButton`**: Clickable button.
- **`JLabel`**: Display text or images.
- **`JTextField`**: Single-line text input.
- **`JTextArea`**: Multi-line text input.
- **`JCheckBox`**: Checkable box for multiple selections.
- **`JRadioButton`**: Allows selecting only one option from a group.
- **`JComboBox`**: Dropdown menu for selecting items.
- **`JList`**: Displays a list of selectable items.

4.3 Containers and Layout Components

- `JPanel`: Generic container to hold components.
- `JScrollPane`: Adds scrolling functionality.
- `JTabbedPane`: Provides tabbed navigation.
- `JSplitPane`: Divides a window into two resizable areas.

4.4 Advanced Components

- `JTable`: Displays tabular data.
- `JTree`: Displays hierarchical data.
- `JProgressBar`: Shows the progress of a task.
- `JSlider`: Allows selecting a numeric value using a slider.

Simple Swing GUI

```
import javax.swing.*;

import java.awt.*;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;


public class SimpleSwingGUI {

   public static void main(String[] args) {

       // Create the main frame (window)

       JFrame frame = new JFrame("Simple Swing GUI");

       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

       frame.setSize(300, 200);

       frame.setLayout(new FlowLayout());
```

```
        // Create a label
        JLabel label = new JLabel("Click the button!");

        // Create a button
        JButton button = new JButton("Click Me");

        // Add action listener to the button
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                label.setText("Button Clicked!");
            }
        });

        // Add components to the frame
        frame.add(label);
        frame.add(button);

        // Set frame visibility
        frame.setVisible(true);
```

```
    }
}
```

Output

```
-------------------------
| Simple Swing GUI       | <-- Window Title
|------------------------ |
| Click the button!      | <-- JLabel (Before Click)
| [ Click Me ]           | <-- JButton
 -------------------------
```

---

5. Conclusion

- Swing provides a powerful hierarchy for building graphical user interfaces.
- It offers lightweight and flexible components compared to AWT.
- The key classes include `JComponent`, `JFrame`, and various UI controls.

Difference Between AWT and Swing in Java

Both **AWT (Abstract Window Toolkit)** and **Swing** are used for creating Graphical User Interfaces (GUI) in Java. However, Swing is an improved version of AWT with more functionalities and a richer set of components.

---

1. Overview of AWT

- AWT is **part of Java's original GUI toolkit** (introduced in JDK 1.0).

- It is based on **native system components (heavyweight)**, meaning it depends on the underlying operating system.
- Limited flexibility and fewer components.

2. Overview of Swing

- Swing is part of **Java Foundation Classes (JFC)** (introduced in JDK 1.2).
- It is **lightweight** and does **not depend on the OS**, meaning it has a consistent look across platforms.
- Provides a **richer set of components** than AWT.

---

3. Key Differences Between AWT and Swing

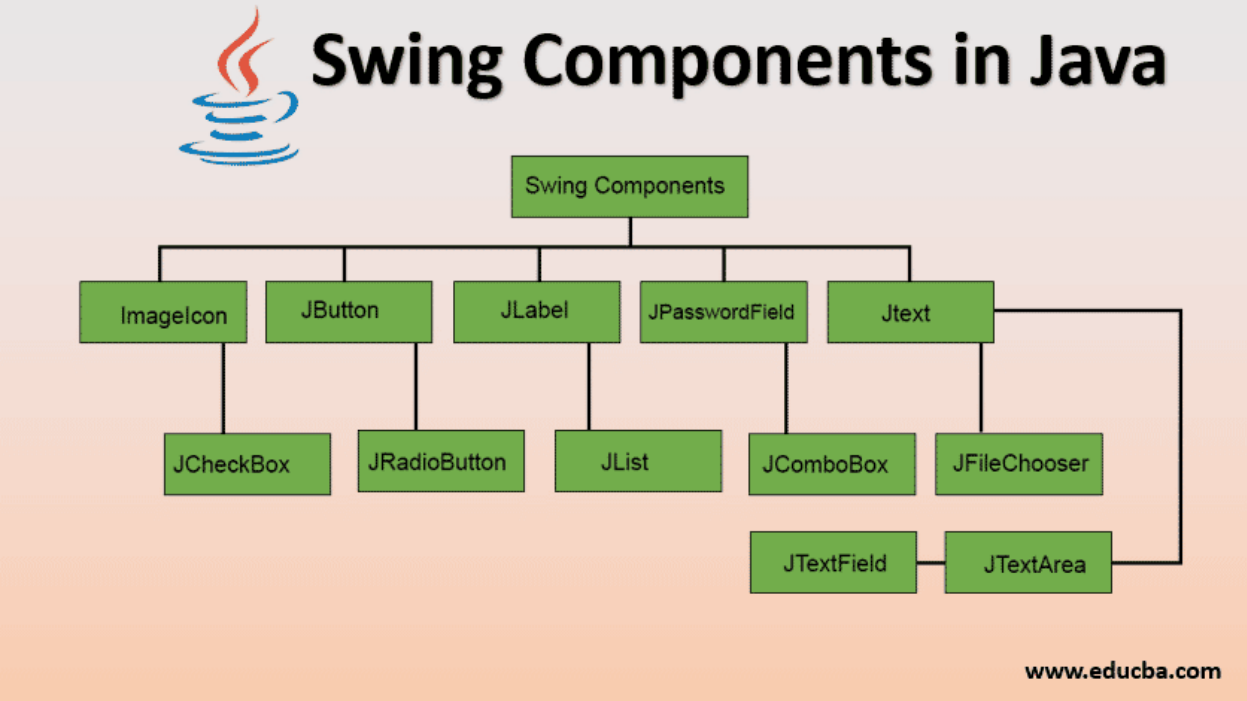
Feature	AWT (Abstract Window Toolkit)	Swing
Dependency	Uses **native OS components** (heavyweight).	Written in Java, does not depend on the OS (lightweight).
Look and Feel	UI appearance depends on the **operating system**.	Can have a **consistent look and feel** (supports themes like Metal, Nimbus).
Components	Basic UI components (e.g., `Button`, `TextField`, `Label`).	Rich UI components (e.g., `JButton`, `JTextField`, `JLabel`).
Customization	Limited customization options.	Highly customizable (supports icons, colors, fonts, etc.).
Performance	Slower because it interacts with OS-level components.	Faster because it does not rely on OS components.
Event Handling	Uses **old event model** (before Java 1.1).	Uses **advanced event model** (introduced in Java 1.1).
Extensibility	Difficult to extend and modify.	Easily extendable using `JComponent` class.
Pluggable Look and Feel	Not supported.	Supports different Look and Feel (e.g., Metal, Nimbus).
MVC Architecture	No proper MVC support.	Supports **Model-View-Controller (MVC)** architecture.
Support for Advanced Features	No support for tabbed panes, trees, tables, etc.	Supports advanced features like `JTabbedPane`, `JTree`, `JTable`, `JScrollPane`.

Containers, and UI Components in swing

What is a Component in Java Swing?

A **component** in Java Swing is an individual **graphical element** that is used to build a GUI (Graphical User Interface). Components include **buttons, text fields, labels, checkboxes, tables, menus, etc.**.

All Swing components are derived from the base class `javax.swing.JComponent`, which itself extends `java.awt.Component`. Abstract Window Toolkit.



Types of Containers:

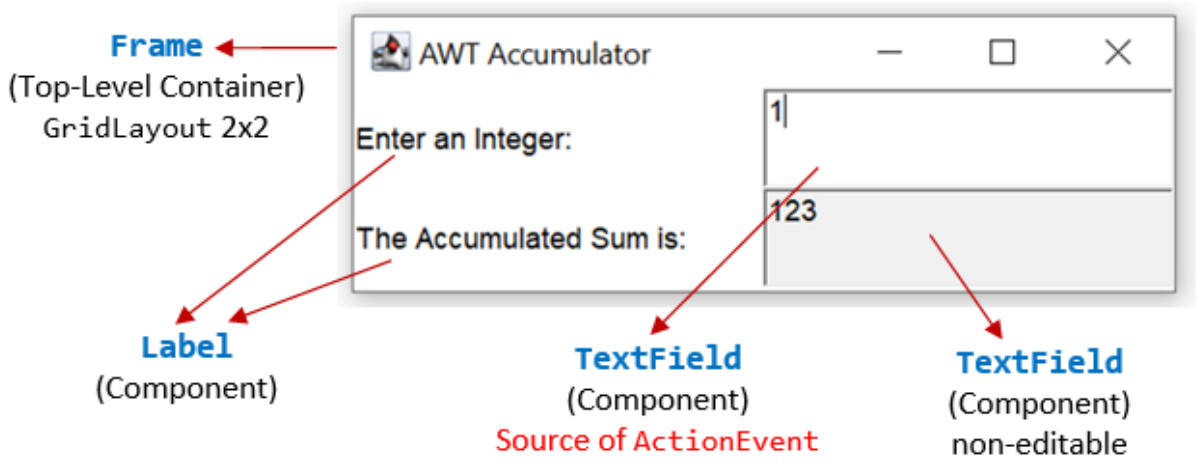
1. **Top-Level Containers** (Cannot be added inside other containers)
    - **JFrame** → Represents a window with title and close button.
    - **JDialog** → Popup window for messages or user input.
    - **JApplet** → Used for applets in a web browser (deprecated).
    - **JWindow** → Borderless window.

2. **Intermediate Containers** (Used to group components inside top-level containers)
    - **JPanel** → Used for grouping components inside a JFrame.
    - **JScrollPane** → Provides scrolling functionality to a component.
    - **JSplitPane** → Splits the UI into two resizable sections.
    - **JTabbedPane** → Provides a tabbed interface.
    - **JLayeredPane** → Manages layers of components.
    - **JDesktopPane** → Used for multiple internal frames (MDI applications).

---

2. UI Components in Swing

Swing provides several UI components for building interactive applications.



Common Swing Components:

1. **Buttons**
    - `JButton` → Simple clickable button.
    - `JToggleButton` → Button that can be toggled (pressed or unpressed).
    - `JCheckBox` → Checkbox for multiple selection.
    - `JRadioButton` → Used in groups for single selection.
2. **Text Input**
    - `JTextField` → Single-line text input.
    - `JPasswordField` → Input field for passwords (hidden characters).
    - `JTextArea` → Multi-line text input.
3. **Labels & Display**
    - `JLabel` → Displays text or images.
    - `JProgressBar` → Shows progress of a task.
    - `JToolTip` → Provides tooltips on hover.
4. **Menus & Toolbars**
    - `JMenuBar` → Menu bar at the top.

- **`JMenu`** → Menu inside a menu bar.
- **`JMenuItem`** → Menu option inside a menu.
- `JToolBar` → Toolbar with buttons/icons.

5. **List Components**
   - `JComboBox` → Drop-down list for selection.
   - `JList` → List box with multiple selections.
6. **Tables & Trees**
   - `JTable` → Displays tabular data.
   - `JTree` → Displays hierarchical data.
7. **Dialogs & Notifications**
   - `JOptionPane` → Pre-built dialogs (alerts, confirm dialogs, input dialogs).
   - `JFileChooser` → Dialog for file selection.
   - `JColorChooser` → Dialog for selecting colors.

Java Swing Program Example:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class SwingDemo {
    public static void main(String[] args) {
        // Creating a JFrame (Top-Level Container)
        JFrame frame = new JFrame("Swing Components Example");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new FlowLayout());

        // Creating a JPanel (Intermediate Container)
        JPanel panel = new JPanel();

        // Creating UI Components
        JLabel label = new JLabel("Enter your name:");
        JTextField textField = new JTextField(15);
        JButton button = new JButton("Submit");
        JLabel resultLabel = new JLabel("");

        // Action Listener for Button
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                String name = textField.getText();
                resultLabel.setText("Hello, " + name + "!");
            }
        });

        // Adding Components to Panel
```

```
        panel.add(label);
        panel.add(textField);
        panel.add(button);
        panel.add(resultLabel);

        // Adding Panel to Frame
        frame.add(panel);

        // Making the Frame Visible
        frame.setVisible(true);
    }
}
```

Step-by-Step Breakdown

1. Import Statements

```
import javax.swing.*;  // Imports Swing components (JFrame, JPanel, JLabel, JTextField, JButton, etc.)
import java.awt.*;  // Imports AWT components (layout managers)
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;  // Imports event handling classes for button click actions.
```

- `javax.swing.*` provides the GUI components.
- `java.awt.*` provides layout and graphical properties.
- `java.awt.event.*` provides event-handling classes.

---

2. Main Class and Method

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

- `public class SwingDemo` → Defines a class named **SwingDemo**.
- `public static void main(String[] args)` → The **main method** where execution starts.

---

3. Creating the Main Window (JFrame)

```
JFrame frame = new JFrame("Swing Components Example");
frame.setSize(400, 300);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setLayout(new FlowLayout());
```

- `JFrame frame = new JFrame("Swing Components Example");` → Creates a window (JFrame) with the title **"Swing Components Example"**.

- `frame.setSize(400, 300);` → Sets the **width (400px)** and **height (300px)**.
- `frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);` → Closes the application when the window is closed.
- `frame.setLayout(new FlowLayout());` → Uses **FlowLayout**, which arranges components from left to right.

---

4. Creating a Panel (Intermediate Container)

```
JPanel panel = new JPanel();
```

- `JPanel` is a container used to group multiple components.
- It helps in organizing components inside the **JFrame**.

---

5. Creating UI Components

```
JLabel label = new JLabel("Enter your name:");
JTextField textField = new JTextField(15);
JButton button = new JButton("Submit");
JLabel resultLabel = new JLabel("");
```

- `JLabel label = new JLabel("Enter your name:");` → Displays a **label** prompting the user.
- `JTextField textField = new JTextField(15);` → Creates a text field where the user can enter their name (**15 columns wide**).
- `JButton button = new JButton("Submit");` → Creates a **button** labeled "Submit".
- `JLabel resultLabel = new JLabel("");` → Creates an empty label where the greeting message will be displayed.

---

6. Adding Event Handling (Button Click)

```
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        String name = textField.getText();
        resultLabel.setText("Hello, " + name + "!");
    }
});
```

- `button.addActionListener(new ActionListener() { ... });` → Adds an event listener to detect button clicks.
- `public void actionPerformed(ActionEvent e) { ... }` → This method is called when the button is clicked.

- `String name = textField.getText();` → Retrieves the text entered by the user.
- `resultLabel.setText("Hello, " + name + "!");` → Updates the `resultLabel` with a greeting message.

---

7. Adding Components to the Panel

```
panel.add(label);
panel.add(textField);
panel.add(button);
panel.add(resultLabel);
```

- These lines **add components** to the `JPanel`, maintaining the structure.

---

8. Adding Panel to Frame and Making it Visible

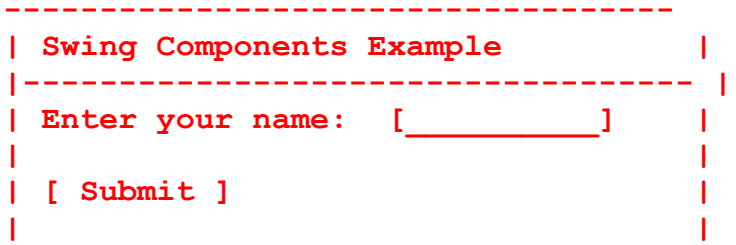
```
frame.add(panel);
frame.setVisible(true);
```

- `frame.add(panel);` → Adds the panel (which contains all components) to the **JFrame**.
- `frame.setVisible(true);` → Displays the window.

Final Output (GUI Window)

1. A window appears with:
   - A **label** saying **"Enter your name:"**.
   - A **text field** for user input.
   - A **"Submit" button**.
   - An **empty label** to display the result.
2. When the user types their name and clicks **"Submit"**, the label updates with **"Hello, [User's Name]!"**.

Example Screenshot (Expected UI)



```
| Hello, John! (Displayed after clicking) |
 ----------------------------------
```

Enhancements You Can Try

1. **Add More Components** (e.g., `JComboBox`, `JCheckBox`, `JRadioButton`).
2. **Change Layout** (`BorderLayout`, `GridLayout`, etc.).
3. **Use JOptionPane for Popups**.
4. **Style Components** (Change font, color, background).

AWT Components, Layout Manager Interface, and Default Layouts in Java

1. AWT Components

AWT (Abstract Window Toolkit) provides a set of UI components to build graphical user interfaces (GUI) in Java. Some important AWT components are:

Common AWT Components:

Component	Description
`Button`	Creates a clickable button.
`Label`	Displays a text label.
`TextField`	Allows single-line text input.
`TextArea`	Allows multi-line text input.
`Checkbox`	Creates a selectable checkbox.
`CheckboxGroup`	Used to create radio buttons.
`List`	Displays a list of selectable items.
`Choice`	A drop-down list for selecting an item.
`Canvas`	Allows custom graphics.

Component	Description
`Scrollbar`	Adds scroll functionality.
`Panel`	A container for organizing components.
`Frame`	A top-level window.
`Dialog`	A pop-up window for messages.

---

2. Layout Manager Interface

A **layout manager** is responsible for organizing components inside a container (e.g., `Frame`, `Panel`). Java provides the `LayoutManager` interface, which has different implementations for arranging components.

Types of Layout Managers in AWT:

1. **FlowLayout** – Places components in a row (default for `Panel`).
2. **BorderLayout** – Divides the container into five regions: `NORTH`, `SOUTH`, `EAST`, `WEST`, and `CENTER` (default for `Frame`).
3. **GridLayout** – Arranges components in a grid with equal-sized cells.
4. **CardLayout** – Allows multiple components, showing only one at a time like a stack of cards.
5. **GridBagLayout** – A flexible grid-based layout where cells can vary in size.
6. **Null Layout (Absolute Positioning)** – No layout manager; requires manual positioning using `setBounds()`.

---

3. Default Layouts in Java AWT

Container	Default Layout Manager
`Frame`	BorderLayout
`Panel`	FlowLayout
`Applet`	FlowLayout
`Dialog`	BorderLayout

Each layout has different behavior for arranging components, but we can change the default layout using the `setLayout()` method.

Key Points

- AWT provides **predefined components** to build GUI applications.
- The **LayoutManager interface** controls the arrangement of components.
- Each container has a **default layout** but can be changed.

Event Model, Listeners, and Event Handling in Java.

Date of lecture

19/03/2025

In Java, **event handling** is used to create interactive applications. The **AWT Event Model** allows handling user interactions like button clicks, mouse movements, and keyboard actions.

---

1. Java Event Model

Java follows a **delegation-based event model**, which consists of:

1. **Event Source** – The component that generates an event (e.g., `Button`, `TextField`).
2. **Event Object** – Represents the event, such as `ActionEvent`, `MouseEvent`, `KeyEvent`, etc.
3. **Event Listener** – The interface that listens for and processes events.

Example Workflow:

1. A user **clicks a button** → An **event is generated** (`ActionEvent`).
2. The event is sent to a **registered listener** (`ActionListener`).
3. The listener executes the **callback method** (`actionPerformed()`).

---

2. Event Listeners in Java

A **listener** is an interface that handles specific events. Java provides several listener interfaces, including:

Listener Interface	Method to Implement	Event Type
`ActionListener`	`actionPerformed(ActionEvent e)`	Button clicks, menu selections
`KeyListener`	`keyPressed(KeyEvent e)`, `keyReleased(KeyEvent e)`, `keyTyped(KeyEvent e)`	Keyboard events
`MouseListener`	`mouseClicked(MouseEvent e)`, `mousePressed(MouseEvent e)`, `mouseReleased(MouseEvent e)`, `mouseEntered(MouseEvent e)`, `mouseExited(MouseEvent e)`	Mouse clicks and movements
`MouseMotionListener`	`mouseDragged(MouseEvent e)`, `mouseMoved(MouseEvent e)`	Mouse motion events
`ItemListener`	`itemStateChanged(ItemEvent e)`	Checkbox, choice selection
`WindowListener`	`windowOpened(WindowEvent e)`, `windowClosing(WindowEvent e)`, `windowClosed(WindowEvent e)`, etc.	Window actions

---

3. Event Handling in Java (Example with Button Click)

Below is an example of **event handling** using an `ActionListener` to handle a button click event.

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

public class EventHandlingExample extends Frame implements ActionListener {
    // Creating components
    Button button;
```

```
    // Constructor
    public EventHandlingExample() {
        // Set layout
        setLayout(new FlowLayout());

        // Create button
        button = new Button("Click Me");

        // Register listener
        button.addActionListener(this);

        // Add button to frame
        add(button);

        // Set frame properties
        setSize(300, 200);
        setVisible(true);
    }

    // Implement actionPerformed method
    public void actionPerformed(ActionEvent e) {
        button.setLabel("Clicked!");
    }

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

Output Explanation:

1. A **button** is displayed inside the window.
2. When the user **clicks the button**, an `ActionEvent` is generated.
3. The `actionPerformed()` method executes and **changes the button label to "Clicked!"**.

---

4. Key Points in Event Handling:

- Events are **generated by user interactions** (clicks, key presses, etc.).
- **Listeners handle** these events by implementing methods like `actionPerformed()`.
- **Listeners must be registered** using methods like `addActionListener()`.
- Java provides **predefined listener interfaces** for handling different event types.

Need for Packages and Associating Classes to Packages in Java

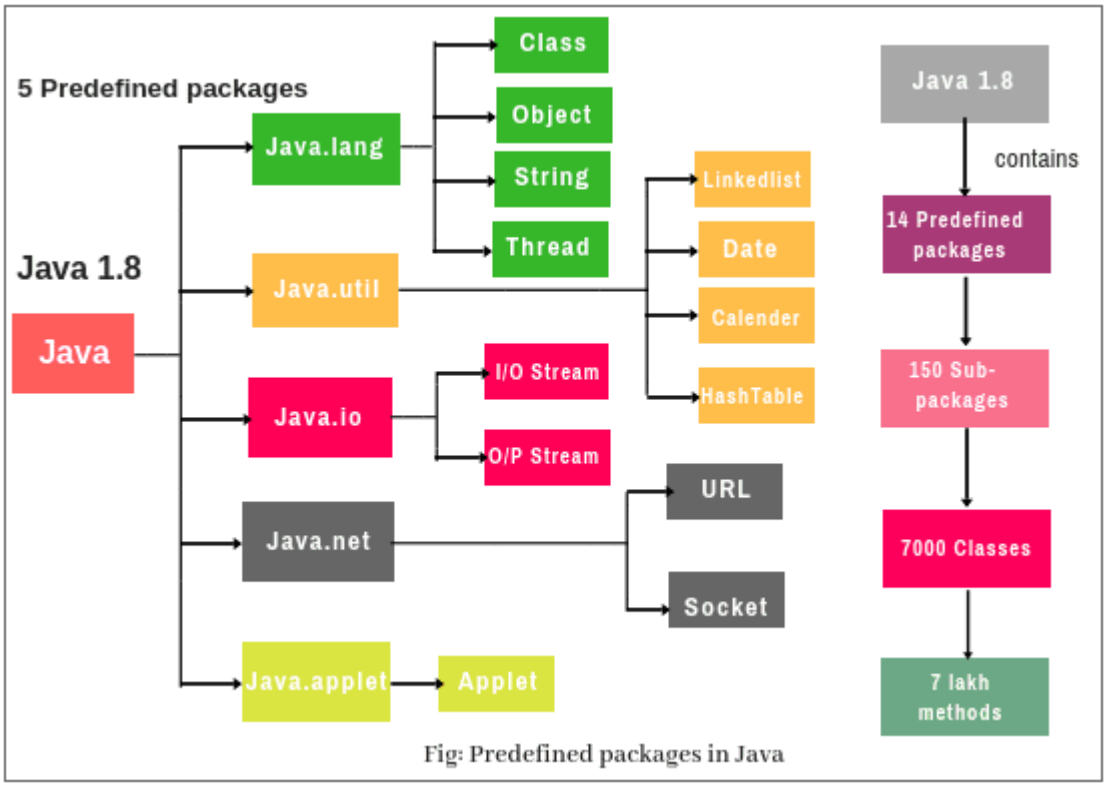


Fig: Predefined packages in Java

1. What is a Package in Java?

A **package** in Java is a way to group related **classes and interfaces** together, similar to folders in a file system. It helps in **organizing code, avoiding name conflicts, and enhancing reusability**.

---

2. Need for Packages in Java

Packages provide several advantages:

1. **Code Organization** – Helps keep related classes grouped together.
2. **Avoids Name Conflicts** – Prevents class name clashes when multiple developers work on a project.
3. **Encapsulation** – Allows controlling access levels using **public, protected, private, and default** access modifiers.
4. **Reusability** – Once a package is created, it can be reused in multiple programs.
5. **Easy Maintenance** – Divides code into modular sections, making it easier to debug and manage.
6. **Security** – Allows defining restricted access to certain classes or methods.

---

3. Associating Classes to Packages

To create and associate a class with a package:

Step 1: Declare a Package in a Java File

```
package mypackage; // Declaring package

public class MyClass {
    public void display() {
        System.out.println("Hello from MyClass in mypackage!");
    }
}
```

Step 2: Compile the Java File

Use the `-d` option to store the compiled class in a directory corresponding to the package.

```
javac -d . MyClass.java
```

This will create a folder `mypackage` and place the `.class` file inside it.

Step 3: Use the Package in Another Class

```
import mypackage.MyClass;  // Importing the package

public class TestPackage {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.display();
    }
}
```

Step 4: Compile and Run the Program

```
javac TestPackage.java
java TestPackage
```

Output:

```
Hello from MyClass in mypackage!
```

---

4. Types of Packages in Java

1. **Built-in Packages** – Java provides predefined packages such as:
    - `java.util` → Contains utility classes like `ArrayList`, `Scanner`.
    - `java.io` → Used for file handling.
    - `java.awt` → Provides GUI components.

- `javax.swing` → Advanced GUI features.

2. **User-defined Packages** – Created by developers to organize custom classes.

---

5. Access Modifiers and Packages

Modifier	Same Package	Different Package (Subclass)	Different Package (Non-Subclass)
`public`	✅ Accessible	✅ Accessible	✅ Accessible
`protected`	✅ Accessible	✅ Accessible (via Inheritance)	❌ Not Accessible
Default (No Modifier)	✅ Accessible	❌ Not Accessible	❌ Not Accessible
`private`	❌ Not Accessible	❌ Not Accessible	❌ Not Accessible

---

4. Key Takeaways

1. **Use `public`** when a class/method should be accessible everywhere.
2. **Use `private`** for encapsulation (restrict access to within the class).
3. **Use `protected`** when the member should be accessible to subclasses.
4. **Use default access** when access should be limited to the same package.